# ARM® Architecture Reference Manual

## ARMv8, for ARMv8-A architecture profile

**Beta**

ARM®

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
ARM DDI 0487A.h (ID092915)

# ARM Architecture Reference Manual
## ARMv8, for ARMv8-A architecture profile

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.

### Release Information

The following releases of this document have been made.

**Release history**

| Date | Issue | Confidentiality | Change |
|---|---|---|---|
| 30 April 2013 | A.a-1 | Confidential-Beta Draft | Beta draft of first issue, limited circulation |
| 12 June 2013 | A.a-2 | Confidential-Beta Draft | Second beta draft of first issue, limited circulation |
| 04 September 2013 | A.a | Non-Confidential Beta | Beta release. |
| 24 December 2013 | A.b | Non-Confidential Beta | Second beta release. |
| 18 July 2014 | A.c | Non-Confidential Beta | Third beta release. |
| 09 October 2014 | A.d | Non-Confidential Beta | Fourth beta release. |
| 17 December 2014 | A.e | Non-Confidential Beta | Fifth beta release. |
| 25 March 2015 | A.f | Non-Confidential Beta | Sixth beta release. |
| 10 July 2015 | A.g | Non-Confidential Beta | Seventh beta release. |
| 30 September 2015 | A.h | Non-Confidential Beta | Eighth beta release. |

### Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM Limited (“ARM”). **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated**.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, ARM makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version shall prevail.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to ARM’s customers is not intended to create or refer to any partnership relationship with any other company. ARM may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement specifically covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the ARM trademark usage guidelines http://www.arm.com/about/trademarks/guidelines/index.php.

This document is Non-Confidential but any disclosure by you is subject to you providing the recipient the conditions set out in this notice and procuring the acceptance by the recipient of the conditions set out in this notice.

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
ARM Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20327

In this document, where the term ARM is used to refer to the company it means “ARM or any of its subsidiaries as appropriate”.

**Note**

- The term ARM can refer to versions of the ARM architecture, for example ARMv8 refers to version 8 of the ARM architecture. The context makes it clear when the term is used in this way.
- This document describes only the ARMv8-A architecture profile. For the behaviors required by the ARMv7-A and ARMv7-R architecture profiles, see the *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

#### Web Address

`http://www.arm.com`

### *Status and limitations of issue A.h*

This issue A.h of the ARMv8 Architecture Reference Manual contains many improvements and corrections. However, as indicated by its overall beta status, it remains work-in-progress. ARM is aware of the following issues that will be addressed in the next issue:

- Compared to issue A.g:
  - We are in the process of improving the introductory descriptions of the A64 instructions in Chapter C7 *A64 Advanced SIMD and Floating-point Instruction Descriptions*. However, this work is incomplete and many instructions still have only a minimal introduction.
  - We are in the process of replacing pseudocode inserts in parts B to I of the manual with links to the appropriate functions in Chapter J1 *ARMv8 Pseudocode*. This work is incomplete.

    **Note**

    Chapter J1 was previously an Appendix.

  - Appendix K11 *ARM Pseudocode Definition* has been updated and restructured extensively, to describe the ARMv8 version of the ARM pseudocode. These changes have had very limited review.
- We are working to improve the descriptions of register reset behavior in the register descriptions throughout this manual. This remains work-in-progress for the debug System registers and memory-mapped registers.

  **Note**

  Except for non-debug System registers, the register descriptions might not give a clear indication of when any stated reset values apply.
- There have been additions to the Performance Monitors Extension described in Chapter D5 *The Performance Monitors Extension*. These additions include the definition of additional events, described in Chapter D5, and additional recommended IMPLEMENTATION DEFINED events, described in Appendix K3 *Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events*. These additions have had only very limited review.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Contents
# ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile


**Preface**
About this manual ..... xvi
Using this manual ..... xviii
Conventions ..... xxiii
Additional reading ..... xxv
Feedback ..... xxvi

## Part A ARMv8 Architecture Introduction and Overview

**Chapter A1 Introduction to the ARMv8 Architecture**
A1.1 About the ARM architecture ..... A1-30
A1.2 Architecture profiles ..... A1-32
A1.3 ARMv8 architectural concepts ..... A1-33
A1.4 Supported data types ..... A1-36
A1.5 Floating-point and Advanced SIMD support ..... A1-46
A1.6 Cryptographic Extension ..... A1-52
A1.7 The ARM memory model ..... A1-53

## Part B The AArch64 Application Level Architecture

**Chapter B1 The AArch64 Application Level Programmers' Model**
B1.1 About the Application level programmers' model ..... B1-58
B1.2 Registers in AArch64 Execution state ..... B1-59
B1.3 Software control features and EL0 ..... B1-64




*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Chapter B2** **The AArch64 Application Level Memory Model**
B2.1 Address space ... B2-68
B2.2 Memory type overview ... B2-69
B2.3 Caches and memory hierarchy ... B2-70
B2.4 Alignment support ... B2-75
B2.5 Endian support ... B2-77
B2.6 Atomicity in the ARM architecture ... B2-80
B2.7 Memory ordering ... B2-83
B2.8 Memory types and attributes ... B2-93
B2.9 Mismatched memory attributes ... B2-104
B2.10 Synchronization and semaphores ... B2-107

**Part C** **The AArch64 Instruction Set**

**Chapter C1** **The A64 Instruction Set**
C1.1 Introduction ... C1-120
C1.2 Structure of the A64 assembler language ... C1-121
C1.3 Address generation ... C1-126
C1.4 Instruction aliases ... C1-129

**Chapter C2** **About the A64 Instruction Descriptions**
C2.1 Understanding the A64 instruction descriptions ... C2-132
C2.2 Conventions used in AArch64 instruction and System register descriptions ... C2-135

**Chapter C3** **A64 Instruction Set Overview**
C3.1 Branches, Exception generating, and System instructions ... C3-138
C3.2 Loads and stores ... C3-142
C3.3 Data processing - immediate ... C3-153
C3.4 Data processing - register ... C3-158
C3.5 Data processing - SIMD and floating-point ... C3-165

**Chapter C4** **A64 Instruction Set Encoding**
C4.1 A64 instruction index by encoding ... C4-186
C4.2 Data processing - immediate ... C4-187
C4.3 Branches, exception generating and system instructions ... C4-191
C4.4 Loads and stores ... C4-196
C4.5 Data processing - register ... C4-218
C4.6 Data processing - SIMD and floating point ... C4-227

**Chapter C5** **The A64 System Instruction Class**
C5.1 The System instruction class encoding space ... C5-266
C5.2 Special-purpose registers ... C5-289
C5.3 A64 system instructions for cache maintenance ... C5-342
C5.4 A64 system instructions for address translation ... C5-358
C5.5 A64 system instructions for TLB maintenance ... C5-371

**Chapter C6** **A64 Base Instruction Descriptions**
C6.1 Introduction ... C6-426
C6.2 Register size ... C6-427
C6.3 Use of the PC ... C6-428
C6.4 Use of the stack pointer ... C6-429
C6.5 Condition flags and related instructions ... C6-430
C6.6 Alphabetical list of instructions ... C6-431

**Chapter C7** **A64 Advanced SIMD and Floating-point Instruction Descriptions**
C7.1 Introduction ... C7-836


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

C7.2 About the SIMD and floating-point instructions ..... C7-837
C7.3 Alphabetical list of floating-point and Advanced SIMD instructions ..... C7-839

# Part D The AArch64 System Level Architecture

## Chapter D1 The AArch64 System Level Programmers' Model

D1.1 Exception levels ..... D1-1514
D1.2 Exception terminology ..... D1-1515
D1.3 Execution state ..... D1-1517
D1.4 Security state ..... D1-1518
D1.5 Virtualization ..... D1-1520
D1.6 Registers for instruction processing and exception handling ..... D1-1523
D1.7 Process state, PSTATE ..... D1-1529
D1.8 Program counter and stack pointer alignment ..... D1-1531
D1.9 Reset ..... D1-1533
D1.10 Exception entry ..... D1-1537
D1.11 Exception return ..... D1-1552
D1.12 The Exception level hierarchy ..... D1-1556
D1.13 Synchronous exception types, routing and priorities ..... D1-1563
D1.14 Asynchronous exception types, routing, masking and priorities ..... D1-1569
D1.15 Configurable instruction enables and disables, and trap controls ..... D1-1575
D1.16 System calls ..... D1-1610
D1.17 Mechanisms for entering a low-power state ..... D1-1611
D1.18 Self-hosted debug ..... D1-1616
D1.19 The Performance Monitors Extension ..... D1-1618
D1.20 Interprocessing ..... D1-1619
D1.21 The effect of implementation choices on the programmers' model ..... D1-1631

## Chapter D2 AArch64 Self-hosted Debug

D2.1 About debug exceptions ..... D2-1637
D2.2 The debug exception enable controls ..... D2-1640
D2.3 Routing debug exceptions ..... D2-1641
D2.4 Enabling debug exceptions from the current Exception level and Security state ..... D2-1643
D2.5 The effect of powerdown on debug exceptions ..... D2-1645
D2.6 Summary of the routing and enabling of debug exceptions ..... D2-1646
D2.7 Pseudocode description of debug exceptions ..... D2-1648
D2.8 Software Breakpoint Instruction exceptions ..... D2-1650
D2.9 Breakpoint exceptions ..... D2-1653
D2.10 Watchpoint exceptions ..... D2-1671
D2.11 Vector Catch exceptions ..... D2-1685
D2.12 Software Step exceptions ..... D2-1686
D2.13 Synchronization and debug exceptions ..... D2-1701

## Chapter D3 The AArch64 System Level Memory Model

D3.1 About the memory system architecture ..... D3-1704
D3.2 Address space ..... D3-1705
D3.3 Mixed-endian support ..... D3-1706
D3.4 Cache support ..... D3-1707
D3.5 External aborts ..... D3-1727
D3.6 Memory barrier instructions ..... D3-1729
D3.7 Pseudocode description of general memory system instructions ..... D3-1730

## Chapter D4 The AArch64 Virtual Memory System Architecture

D4.1 About the Virtual Memory System Architecture (VMSA) ..... D4-1742
D4.2 The VMSAv8-64 address translation system ..... D4-1745
D4.3 VMSAv8-64 translation table format descriptors ..... D4-1799


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

D4.4 Access controls and memory region attributes ..... D4-1808
D4.5 MMU faults ..... D4-1824
D4.6 Translation Lookaside Buffers (TLBs) ..... D4-1833
D4.7 TLB maintenance requirements and the TLB maintenance instructions ..... D4-1837
D4.8 Caches in a VMSA implementation ..... D4-1850

**Chapter D5 The Performance Monitors Extension**
D5.1 About the Performance Monitors ..... D5-1856
D5.2 Accuracy of the Performance Monitors ..... D5-1859
D5.3 Behavior on overflow ..... D5-1861
D5.4 Attributability ..... D5-1863
D5.5 Effect of EL3 and EL2 ..... D5-1864
D5.6 Event filtering ..... D5-1866
D5.7 Performance Monitors and Debug state ..... D5-1868
D5.8 Counter enables ..... D5-1869
D5.9 Counter access ..... D5-1870
D5.10 Events, Event numbers and mnemonics ..... D5-1872
D5.11 Performance Monitors Extension registers ..... D5-1893

**Chapter D6 The Generic Timer in AArch64 state**
D6.1 About the Generic Timer in AArch64 state ..... D6-1898

**Chapter D7 AArch64 System Register Descriptions**
D7.1 About the AArch64 System registers ..... D7-1908
D7.2 General system control registers ..... D7-1913
D7.3 Debug registers ..... D7-2165
D7.4 Performance Monitors registers ..... D7-2236
D7.5 Generic Timer registers ..... D7-2276

**Part E The AArch32 Application Level Architecture**

**Chapter E1 The AArch32 Application Level Programmers' Model**
E1.1 About the Application level programmers' model ..... E1-2310
E1.2 Additional information about the programmers' model in AArch32 state ..... E1-2311
E1.3 Advanced SIMD and floating-point instructions ..... E1-2325
E1.4 Conceptual coprocessor support ..... E1-2353
E1.5 Exceptions ..... E1-2354

**Chapter E2 The AArch32 Application Level Memory Model**
E2.1 Address space ..... E2-2356
E2.2 Memory type overview ..... E2-2357
E2.3 Caches and memory hierarchy ..... E2-2358
E2.4 Alignment support ..... E2-2363
E2.5 Endian support ..... E2-2365
E2.6 Atomicity in the ARM architecture ..... E2-2368
E2.7 Memory ordering ..... E2-2372
E2.8 Memory types and attributes ..... E2-2382
E2.9 Mismatched memory attributes ..... E2-2392
E2.10 Synchronization and semaphores ..... E2-2395

**Part F The AArch32 Instruction Sets**

**Chapter F1 The AArch32 Instruction Sets Overview**
F1.1 Support for instructions in different versions of the ARM architecture ..... F1-2408
F1.2 Unified Assembler Language ..... F1-2409
F1.3 Branch instructions ..... F1-2411


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

F1.4 Data-processing instructions ..... F1-2412
F1.5 PSTATE and banked register access instructions ..... F1-2420
F1.6 Load/store instructions ..... F1-2421
F1.7 Load/store multiple instructions ..... F1-2424
F1.8 Miscellaneous instructions ..... F1-2425
F1.9 Exception-generating and exception-handling instructions ..... F1-2426
F1.10 Coprocessor instructions ..... F1-2427
F1.11 Advanced SIMD and floating-point load/store instructions ..... F1-2428
F1.12 Advanced SIMD and floating-point register transfer instructions ..... F1-2430
F1.13 Advanced SIMD data-processing instructions ..... F1-2431
F1.14 Floating-point data-processing instructions ..... F1-2439

**Chapter F2 About the T32 and A32 Instruction Descriptions**
F2.1 Format of instruction descriptions ..... F2-2442
F2.2 Standard assembler syntax fields ..... F2-2446
F2.3 Conditional execution ..... F2-2447
F2.4 Shifts applied to a register ..... F2-2449
F2.5 Memory accesses ..... F2-2452
F2.6 Encoding of lists of general-purpose registers and the PC ..... F2-2453
F2.7 About the T32 and A32 instruction encodings ..... F2-2454
F2.8 Additional pseudocode support for instruction descriptions ..... F2-2461

**Chapter F3 T32 Base Instruction Set Encoding**
F3.1 Top level T32 instruction set encoding ..... F3-2464
F3.2 16-bit T32 instruction encoding ..... F3-2465
F3.3 32-bit T32 instruction encoding ..... F3-2476

**Chapter F4 A32 Base Instruction Set Encoding**
F4.1 Top level A32 instruction set encoding ..... F4-2508
F4.2 Data-processing and miscellaneous instructions ..... F4-2509
F4.3 Load/Store Word, Unsigned Byte (immediate, literal) ..... F4-2525
F4.4 Load/Store Word, Unsigned Byte (register) ..... F4-2526
F4.5 Media instructions ..... F4-2527
F4.6 Branch, branch with link, and block data transfer ..... F4-2535
F4.7 Coprocessor instructions, and Supervisor Call ..... F4-2537
F4.8 Unconditional instructions ..... F4-2541

**Chapter F5 T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings**
F5.1 Overview ..... F5-2546
F5.2 Advanced SIMD and floating-point instruction syntax ..... F5-2547
F5.3 Register encoding ..... F5-2551
F5.4 Advanced SIMD data-processing instructions ..... F5-2553
F5.5 Floating-point data-processing instructions ..... F5-2565
F5.6 Advanced SIMD and floating-point register load/store instructions ..... F5-2568
F5.7 Advanced SIMD element or structure load/store instructions ..... F5-2569
F5.8 8, 16, and 32-bit transfers accessing the SIMD and floating-point register file F5-2572
F5.9 64-bit transfers accessing the SIMD and floating-point register file ..... F5-2573

**Chapter F6 T32 and A32 Base Instruction Set Instruction Descriptions**
F6.1 Alphabetical list of T32 and A32 base instruction set instructions ..... F6-2576
F6.2 Encoding and use of Banked register transfer instructions ..... F6-3202

**Chapter F7 T32 and A32 Advanced SIMD and floating-point Instruction Descriptions**
F7.1 Alphabetical list of floating-point and Advanced SIMD instructions ..... F7-3208


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Part G The AArch32 System Level Architecture

## Chapter G1 The AArch32 System Level Programmers' Model


G1.1 About the AArch32 System level programmers' model ..... G1-3746
G1.2 Exception levels ..... G1-3747
G1.3 Exception terminology ..... G1-3748
G1.4 Execution state ..... G1-3750
G1.5 Instruction Set state ..... G1-3752
G1.6 Security state ..... G1-3753
G1.7 Virtualization ..... G1-3756
G1.8 AArch32 PE modes, and general-purpose and Special-purpose registers ..... G1-3758
G1.9 Process state, PSTATE ..... G1-3767
G1.10 Instruction set states ..... G1-3772
G1.11 Handling exceptions that are taken to an Exception level using AArch32 ..... G1-3774
G1.12 Exception return to an Exception level using AArch32 ..... G1-3796
G1.13 Asynchronous exception behavior for exceptions taken from AArch32 state . G1-3801
G1.14 AArch32 state exception descriptions ..... G1-3811
G1.15 Reset into AArch32 state ..... G1-3830
G1.16 Mechanisms for entering a low-power state ..... G1-3834
G1.17 The conceptual coprocessor interface and system control ..... G1-3839
G1.18 Advanced SIMD and floating-point support ..... G1-3841
G1.19 Configurable instruction enables and disables, and trap controls ..... G1-3846


## Chapter G2 AArch32 Self-hosted Debug


G2.1 About debug exceptions ..... G2-3883
G2.2 The debug exception enable controls ..... G2-3886
G2.3 Routing debug exceptions ..... G2-3887
G2.4 Enabling debug exceptions from the current Privilege level and Security state ..... G2-3889
G2.5 The effect of powerdown on debug exceptions ..... G2-3891
G2.6 Summary of permitted routing and enabling of debug exceptions ..... G2-3892
G2.7 Pseudocode description of debug exceptions ..... G2-3894
G2.8 Software Breakpoint Instruction exceptions ..... G2-3895
G2.9 Breakpoint exceptions ..... G2-3898
G2.10 Watchpoint exceptions ..... G2-3923
G2.11 Vector Catch exceptions ..... G2-3937
G2.12 Synchronization and debug exceptions ..... G2-3945


## Chapter G3 The AArch32 System Level Memory Model


G3.1 About the memory system architecture ..... G3-3948
G3.2 Address space ..... G3-3949
G3.3 Mixed-endian support ..... G3-3950
G3.4 AArch32 cache and branch predictor support ..... G3-3952
G3.5 System register support for IMPLEMENTATION DEFINED memory features G3-3975
G3.6 External aborts ..... G3-3976
G3.7 Memory barrier instructions ..... G3-3978
G3.8 Pseudocode description of general memory system instructions ..... G3-3979


## Chapter G4 The AArch32 Virtual Memory System Architecture


G4.1 Execution privilege, Exception levels, and AArch32 Privilege levels ..... G4-3990
G4.2 About VMSAv8-32 ..... G4-3992
G4.3 The effects of disabling address translation stages on VMSAv8-32 behavior G4-4000
G4.4 Translation tables ..... G4-4004
G4.5 The VMSAv8-32 Short-descriptor translation table format ..... G4-4009
G4.6 The VMSAv8-32 Long-descriptor translation table format ..... G4-4018
G4.7 Memory access control ..... G4-4037
G4.8 Memory region attributes ..... G4-4046
G4.9 Translation Lookaside Buffers (TLBs) ..... G4-4058



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

G4.10 TLB maintenance requirements ..... G4-4061
G4.11 Caches in VMSAv8-32 ..... G4-4074
G4.12 VMSAv8-32 memory aborts ..... G4-4078
G4.13 Exception reporting in a VMSAv8-32 implementation ..... G4-4091
G4.14 Address translation instructions ..... G4-4110
G4.15 About the System registers for VMSAv8-32 ..... G4-4116
G4.16 Organization of the CP14 registers in VMSAv8-32 ..... G4-4138
G4.17 Organization of the CP15 registers in VMSAv8-32 ..... G4-4141
G4.18 Functional grouping of VMSAv8-32 System registers ..... G4-4161
G4.19 Pseudocode description of VMSAv8-32 memory system operations ..... G4-4183

**Chapter G5 The Generic Timer in AArch32 state**
G5.1 About the Generic Timer in AArch32 state ..... G5-4202

**Chapter G6 AArch32 System Register Descriptions**
G6.1 About the AArch32 System registers ..... G6-4212
G6.2 General system control registers ..... G6-4213
G6.3 Debug registers ..... G6-4648
G6.4 Performance Monitors registers ..... G6-4737
G6.5 Generic Timer registers ..... G6-4780

## Part H External Debug

**Chapter H1 Introduction to External Debug**
H1.1 Introduction to external debug ..... H1-4816
H1.2 External debug ..... H1-4817

**Chapter H2 Debug State**
H2.1 About Debug state ..... H2-4820
H2.2 Halting the PE on debug events ..... H2-4821
H2.3 Entering Debug state ..... H2-4828
H2.4 Behavior in Debug state ..... H2-4832
H2.5 Exiting Debug state ..... H2-4858

**Chapter H3 Halting Debug Events**
H3.1 Introduction to Halting debug events ..... H3-4862
H3.2 Halting Step debug events ..... H3-4864
H3.3 Halt Instruction debug event ..... H3-4874
H3.4 Exception Catch debug event ..... H3-4875
H3.5 External Debug Request debug event ..... H3-4878
H3.6 OS Unlock Catch debug event ..... H3-4879
H3.7 Reset Catch debug events ..... H3-4880
H3.8 Software Access debug event ..... H3-4881
H3.9 Synchronization and Halting debug events ..... H3-4882

**Chapter H4 The Debug Communication Channel and Instruction Transfer Register**
H4.1 Introduction ..... H4-4886
H4.2 DCC and ITR registers ..... H4-4887
H4.3 DCC and ITR access modes ..... H4-4889
H4.4 Flow control of the DCC and ITR registers ..... H4-4893
H4.5 Synchronization of DCC and ITR accesses ..... H4-4897
H4.6 Interrupt-driven use of the DCC ..... H4-4902
H4.7 Pseudocode description of the operation of the DCC and ITR registers ..... H4-4903

**Chapter H5 The Embedded Cross-Trigger Interface**
H5.1 About the Embedded Cross-Trigger (ECT) ..... H5-4908
H5.2 Basic operation on the ECT ..... H5-4910


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

H5.3 Cross-triggers on a PE in an ARMv8 implementation ..... H5-4914
H5.4 Description and allocation of CTI triggers ..... H5-4915
H5.5 CTI registers programmers' model ..... H5-4919
H5.6 Examples ..... H5-4920

**Chapter H6 Debug Reset and Powerdown Support**
H6.1 About Debug over powerdown ..... H6-4924
H6.2 Power domains and debug ..... H6-4925
H6.3 Core power domain power states ..... H6-4926
H6.4 Emulating low-power states ..... H6-4928
H6.5 Debug OS Save and Restore sequences ..... H6-4930
H6.6 Reset and debug ..... H6-4935

**Chapter H7 The Sample-based Profiling Extension**
H7.1 Sample-based profiling ..... H7-4938

**Chapter H8 About the External Debug Registers**
H8.1 Relationship between external debug and System registers ..... H8-4944
H8.2 Supported access sizes ..... H8-4945
H8.3 Synchronization of changes to the external debug registers ..... H8-4946
H8.4 Memory-mapped accesses to the external debug interface ..... H8-4950
H8.5 External debug interface register access permissions ..... H8-4952
H8.6 External debug interface registers ..... H8-4956
H8.7 Cross-trigger interface registers ..... H8-4961
H8.8 External debug register resets ..... H8-4963

**Chapter H9 External Debug Register Descriptions**
H9.1 Introduction ..... H9-4966
H9.2 Debug registers ..... H9-4967
H9.3 Cross-Trigger Interface registers ..... H9-5054

**Part I Memory-mapped Components of the ARMv8 Architecture**

**Chapter I1 System Level Implementation of the Generic Timer**
I1.1 About the Generic Timer specification ..... I1-5098
I1.2 Memory-mapped counter module ..... I1-5099
I1.3 Counter module control and status register summary ..... I1-5102
I1.4 Memory-mapped timer components ..... I1-5104
I1.5 The CNTBaseN and CNTEL0BaseN frames ..... I1-5105
I1.6 The CNTCTLBase frame ..... I1-5107
I1.7 Providing a complete set of counter and timer features ..... I1-5108
I1.8 Gray-count scheme for timer distribution scheme ..... I1-5110

**Chapter I2 Recommended Memory-mapped Interfaces to the Performance Monitors**
I2.1 About the memory-mapped views of the Performance Monitors registers ..... I2-5112

**Chapter I3 Memory-Mapped System Control Register Descriptions**
I3.1 About the memory-mapped system control register descriptions ..... I3-5118
I3.2 Performance Monitors memory-mapped registers summary ..... I3-5119
I3.3 Performance Monitors memory-mapped register descriptions ..... I3-5121
I3.4 Generic Timer memory-mapped registers overview ..... I3-5172
I3.5 Generic Timer memory-mapped register descriptions ..... I3-5173


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Part J Architectural Pseudocode


## Chapter J1 ARMv8 Pseudocode

J1.1 Pseudocode for AArch64 operations ..... J1-5214
J1.2 Pseudocode for AArch32 operation ..... J1-5273
J1.3 Shared pseudocode ..... J1-5344


# Part K Appendixes


## Appendix K1 Architectural Constraints on UNPREDICTABLE behaviors

K1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors ..... K1-5422
K1.2 Constraints on AArch64 state UNPREDICTABLE behaviors ..... K1-5501

## Appendix K2 Recommended External Debug Interface

K2.1 About the recommended external debug interface ..... K2-5516
K2.2 PMUEVENT bus ..... K2-5519
K2.3 Recommended authentication interface ..... K2-5520
K2.4 Management registers and CoreSight compliance ..... K2-5523

## Appendix K3 Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events

K3.1 ARM recommendations for IMPLEMENTATION DEFINED event numbers ... K3-5530
K3.2 Summary of events for exceptions taken to an Exception Level using AArch64 ..... K3-5544

## Appendix K4 Recommendations for reporting memory attributes on an interconnect

K4.1 ARM recommendations for reporting memory attributes on an interconnect .. K4-5548

## Appendix K5 ARMv8 Changes to the T32 and A32 Instruction Sets

K5.1 The A32 and T32 instruction sets ..... K5-5550
K5.2 Partial deprecation of IT ..... K5-5551
K5.3 New A32 and T32 Load-Acquire/Store-Release instructions ..... K5-5552
K5.4 New A32 and T32 scalar floating-point instructions ..... K5-5553
K5.5 New A32 and T32 Advanced SIMD floating-point instructions ..... K5-5556
K5.6 New A32 and T32 instructions provided by the Cryptographic Extension ..... K5-5558
K5.7 New A32 and T32 System instructions ..... K5-5559
K5.8 CRC32 instructions ..... K5-5561

## Appendix K6 Legacy Instruction Syntax for AArch32 Instruction Sets

K6.1 Legacy Instruction Syntax ..... K6-5564

## Appendix K7 Address translation examples

K7.1 AArch64 Address translation examples ..... K7-5572
K7.2 AArch32 Address translation examples ..... K7-5585

## Appendix K8 Example OS Save and Restore Sequences

K8.1 Save Debug registers ..... K8-5596
K8.2 Restore Debug registers ..... K8-5598

## Appendix K9 Recommended Upload and Download Processes for External Debug

K9.1 Using memory access mode in AArch64 state ..... K9-5602

## Appendix K10 Barrier Litmus Tests

K10.1 Introduction ..... K10-5606
K10.2 Load-Acquire, Store-Release and barriers ..... K10-5609



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

K10.3 Load-Acquire Exclusive, Store-Release Exclusive and barriers ..... K10-5616
K10.4 Using a mailbox to send an interrupt ..... K10-5621
K10.5 Cache and TLB maintenance instructions and barriers ..... K10-5622
K10.6 ARMv7 compatible approaches for ordering, using DMB and DSB barriers . K10-5633

**Appendix K11 ARM Pseudocode Definition**
K11.1 About the ARM pseudocode ..... K11-5648
K11.2 Pseudocode for instruction descriptions ..... K11-5649
K11.3 Data types ..... K11-5651
K11.4 Operators ..... K11-5656
K11.5 Statements and control structures ..... K11-5662
K11.6 Built-in functions ..... K11-5668
K11.7 Miscellaneous helper procedures and functions ..... K11-5671
K11.8 ARM pseudocode definition index ..... K11-5674

**Appendix K12 Pseudocode Index**
K12.1 Pseudocode operators and keywords ..... K12-5678
K12.2 Pseudocode index ..... K12-5679

**Appendix K13 Registers Index**
K13.1 Introduction and register disambiguation ..... K13-5708
K13.2 Alphabetical index of AArch64 registers and system instructions ..... K13-5713
K13.3 Functional index of AArch64 registers and system instructions ..... K13-5722
K13.4 Alphabetical index of AArch32 registers and system instructions ..... K13-5732
K13.5 Functional index of AArch32 registers and system instructions ..... K13-5740
K13.6 Alphabetical index of memory-mapped registers ..... K13-5750
K13.7 Functional index of memory-mapped registers ..... K13-5755

**Glossary**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Preface

This preface introduces the *ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*. It contains the following sections:


- *About this manual* on page xvi.
- *Using this manual* on page xviii.
- *Conventions* on page xxiii.
- *Additional reading* on page xxv.
- *Feedback* on page xxvi.


**Note**

This document describes only the ARMv8-A architecture profile. For the behaviors required by the ARMv7-A and ARMv7-R architecture profiles, see the *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## About this manual

This manual describes the ARM® architecture v8, ARMv8. The architecture describes the operation of an ARMv8-A *Processing element (PE)*, and this manual includes descriptions of:

- The two Execution states, AArch64 and AArch32.
- The instruction sets:
  - In AArch32 state, the A32 and T32 instruction sets, that are compatible with earlier versions of the ARM architecture.
  - In AArch64 state, the A64 instruction set.
- The states that determine how a PE operates, including the current Exception level and Security state, and in AArch32 state the PE mode.
- The Exception model.
- The interprocessing model, that supports transitioning between AArch64 state and AArch32 state.
- The memory model, that defines memory ordering and memory management. This manual covers a single architecture profile, ARMv8-A, that defines a *Virtual Memory System Architecture* (VMSA).
- The programmers' model, and its interfaces to System registers that control most PE and memory system features, and provide status information.
- The Advanced SIMD and floating-point instructions, that provide high-performance:
  - Single-precision and double-precision floating-point operations.
  - Conversions between double-precision, single-precision, and half-precision floating-point values.
  - Integer, single-precision floating-point, and in A64, double-precision vector operations in all instruction sets.
  - Double-precision floating-point vector operations in the A64 instruction set.
- The security model, that provides two security states to support secure applications.
- The virtualization model, that support the virtualization of Non-secure operation.
- The Debug architecture, that provides software access to debug features.

This manual gives the assembler syntax for the instructions it describes, meaning that it describes instructions in textual form. However, this manual is not a tutorial for ARM assembler language, nor does it describe ARM assembler language, except at a very basic level. To make effective use of ARM assembler language, read the documentation supplied with the assembler being used.

This manual is organized into parts:

**Part A** Provides an introduction to the ARMv8-A architecture, and an overview of the AArch64 and AArch32 Execution states.

**Part B** Describes the application level view of the AArch64 Execution state, meaning the view from EL0. It describes the application level view of the programmers' model and the memory model.

**Part C** Describes the A64 instruction set, that is available in the AArch64 Execution state. The descriptions for each instruction also include the precise effects of each instruction when executed at EL0, described as *unprivileged* execution, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate ARM machine code.

**Part D** Describes the system level view of the AArch64 Execution state. It includes details of the System registers, most of which are not accessible from EL0, and the system level view of the programmers' model and the memory model. This part includes the description of self-hosted debug.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Part E** Describes the application level view of the AArch32 Execution state, meaning the view from the EL0. It describes the application level view of the programmers' model and the memory model.

**Note**

In AArch32 state, execution at EL0 is execution in User mode.

**Part F** Describes the T32 and A32 instruction sets, that are available in the AArch32 Execution state. These instruction sets are backwards-compatible with earlier versions of the ARM architecture. This part describes the precise effects of each instruction when executed in User mode, described as *unprivileged* execution or execution at EL0, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate ARM machine code.

**Note**

User mode is the only mode where software execution is unprivileged.

**Part G** Describes the system level view of the AArch32 Execution state, that is generally compatible with earlier versions of the ARM architecture. This part includes details of the System registers, most of which are not accessible from EL0, and the conceptual coprocessor interface to those registers. It also describes the system level view of the programmers' model and the memory model.

**Part H** Describes the Debug architecture for external debug. This provides configuration, breakpoint and watchpoint support, and a *Debug Communications Channel* (DCC) to a debug host.

**Part I** Describes additional features of the architecture that are not closely coupled to a *processing element* (PE), and therefore are accessed through memory-mapped interfaces. Some of these features are OPTIONAL.

**Part J** Provides pseudocode that describes various features of the ARMv8 architecture.

**Part K, Appendixes**

Provide additional information. Some appendixes give information that is not part of the ARMv8 architectural requirements. The cover page of each appendix indicates its status.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Using this manual

The information in this manual is organized into parts, as described in this section.

## Part A, Introduction and Architecture Overview

Part A gives an overview of the ARMv8-A architecture profile, including its relationship to the other ARM PE architectures. It introduces the terminology used to describe the architecture, and gives an overview of the Executions states, AArch64 and AArch32. It contains the following chapter:

**Chapter A1** ***Introduction to the ARMv8 Architecture***

Read this for an introduction to the ARMv8 architecture.

## Part B, The AArch64 Application Level Architecture

Part B describes the application level view of the architecture in AArch64 state. It contains the following chapters:

**Chapter B1** ***The AArch64 Application Level Programmers' Model***

Read this for an application level description of the programmers' model for software executing in AArch64 state. It describes execution at EL0 when EL0 is using AArch64 state.

**Chapter B2** ***The AArch64 Application Level Memory Model***

Read this for an application level description of the memory model for software executing in AArch64 state. It describes the memory model for execution in EL0 when EL0 is using AArch64 state. It includes information about ARM memory types, attributes, and memory access controls.

## Part C, The A64 Instruction Set

Part C describes the A64 instruction set, that is used in AArch64 state. It contains the following chapters:

**Chapter C1** ***The A64 Instruction Set***

Read this for a description of the A64 instruction set and common instruction operation details.

**Chapter C2** ***About the A64 Instruction Descriptions***

Read this to understand the format of the A64 instruction descriptions.

**Chapter C3** ***A64 Instruction Set Overview***

Read this for an overview of the individual A64 instructions, that are divided into five functional groups.

**Chapter C4** ***A64 Instruction Set Encoding***

Read this for a description of the A64 instruction set encoding.

**Chapter C5** ***The A64 System Instruction Class***

Read this for a description of the AArch64 system instructions and register descriptions, and the system instruction class encoding space.

**Chapter C6** ***A64 Base Instruction Descriptions***

Read this for information on key aspects of the A64 base instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

**Chapter C7** ***A64 Advanced SIMD and Floating-point Instruction Descriptions***

Read this for information on key aspects of the A64 Advanced SIMD and floating-point instructions and for descriptions of the individual instructions, which are listed in alphabetical order.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Part D, The AArch64 System Level Architecture

Part D describes the AArch64 the system level view of the architecture. It contains the following chapters:

**Chapter D1** ***The AArch64 System Level Programmers' Model***

Read this for a description of the AArch64 system level view of the programmers' model.

**Chapter D2** ***AArch64 Self-hosted Debug***

Read this for an introduction to, and a description of, self-hosted debug in AArch64 state.

**Chapter D3** ***The AArch64 System Level Memory Model***

Read this for a description of the AArch64 system level view of the general features of the memory system.

**Chapter D4** ***The AArch64 Virtual Memory System Architecture***

Read this for a system level view of the AArch64 Virtual Memory System Architecture (VMSA), the memory system architecture of an ARMv8 implementation that is executing in AArch64 state.

**Chapter D5** ***The Performance Monitors Extension***

Read this for a description of an implementation of the ARM Performance Monitors, that are an optional non-invasive debug component.

**Chapter D6** ***The Generic Timer in AArch64 state***

Read this for a description of an implementation of the AArch64 view of the ARM Generic Timer.

**Chapter D7** ***AArch64 System Register Descriptions***

Read this for an introduction to, and description of, each of the AArch64 System registers.

### Part E, The AArch32 Application Level Architecture

Part E describes the AArch32 application level view of the architecture. It contains the following chapters:

**Chapter E1** ***The AArch32 Application Level Programmers' Model***

Read this for an application level description of the programmers' model for software executing in AArch32 state. It describes execution at EL0 when EL0 is using AArch32 state.

**Chapter E2** ***The AArch32 Application Level Memory Model***

Read this for an application level description of the memory model for software executing in AArch32 state. It describes the memory model for execution in EL0 when EL0 is using AArch32 state. It includes information about ARM memory types, attributes, and memory access controls.

### Part F, The AArch32 Instruction Sets

Part F describes the T32 and A32 instruction sets, that are used in AArch32 state. It contains the following chapters:

**Chapter F1** ***The AArch32 Instruction Sets Overview***

Read this for an overview of the T32 and A32 instruction sets.

**Chapter F2** ***About the T32 and A32 Instruction Descriptions***

Read this to understand the format of the T32 and A32 instruction descriptions.

**Chapter F3** ***T32 Base Instruction Set Encoding***

Read this for an introduction to the T32 instruction set and a description of how the T32 instruction set uses the ARM programmers' model.

**Chapter F4** ***A32 Base Instruction Set Encoding***

Read this for a description of the A32 base instruction set encoding.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Chapter F5** ***T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings***

Read this for an overview of the T32 and A32 Advanced SIMD and floating-point instruction sets.

**Chapter F6** ***T32 and A32 Base Instruction Set Instruction Descriptions***

Read this for a description of each T32 and A32 base instruction.

**Chapter F7** ***T32 and A32 Advanced SIMD and floating-point Instruction Descriptions***

Read this for a description of each T32 and A32 Advanced SIMD and floating-point instruction.

## Part G, The AArch32 System Level Architecture

Part G describes the AArch32 system level view of the architecture. It contains the following chapters:

**Chapter G1** ***The AArch32 System Level Programmers' Model***

Read this for a description of the AArch32 system level view of the programmers' model for execution in an Exception level that is using AArch32.

**Chapter G2** ***AArch32 Self-hosted Debug***

Read this for an introduction to, and a description of, self-hosted debug in AArch64 state.

**Chapter G3** ***The AArch32 System Level Memory Model***

Read this for a system level view of the general features of the memory system.

**Chapter G4** ***The AArch32 Virtual Memory System Architecture***

Read this for a description of the AArch32 Virtual Memory System Architecture (VMSA).

**Chapter G5** ***The Generic Timer in AArch32 state***

Read this for a description of an implementation of the AArch32 view of the ARM Generic Timer.

**Chapter G6** ***AArch32 System Register Descriptions***

Read this for a description of each of the AArch32 System registers.

## Part H, External Debug

Part H describes the architecture for external debug. It contains the following chapters:

**Chapter H1** ***Introduction to External Debug***

Read this for an introduction to external debug, and a definition of the scope of this part of the manual.

**Chapter H2** ***Debug State***

Read this for a description of debug state, which the PE might enter as the result of a Halting debug event.

**Chapter H3** ***Halting Debug Events***

Read this for a description of the external debug events referred to as Halting debug events.

**Chapter H4** ***The Debug Communication Channel and Instruction Transfer Register***

Read this for a description of the communication between a debugger and the PE debug logic using the Debug Communications Channel and the Instruction Transfer register.

**Chapter H5** ***The Embedded Cross-Trigger Interface***

Read this for a description of the embedded cross-trigger interface.

**Chapter H6** ***Debug Reset and Powerdown Support***

Read this for a description of reset and powerdown support in the Debug architecture.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Chapter H7** ***The Sample-based Profiling Extension***

Read this for a description of the Sample-based Profiling Extension that is an OPTIONAL extension to an ARMv8 implementation.

**Chapter H8** ***About the External Debug Registers***

Read this for some additional information about the external debug registers.

**Chapter H9** ***External Debug Register Descriptions***

Read this for a description of each external debug register.

## Part I, Memory-mapped Components of the ARMv8 Architecture

Part I describes the memory-mapped components in the architecture. It contains the following chapters:

**Chapter I1** ***System Level Implementation of the Generic Timer***

Read this for a definition of a system level implementation of the Generic Timer.

**Chapter I2** ***Recommended Memory-mapped Interfaces to the Performance Monitors***

Read this for a description of the recommended memory-mapped and external debug interfaces to the Performance Monitors.

**Chapter I3** ***Memory-Mapped System Control Register Descriptions***

Read this for a description of each memory-mapped system control register.

## Part J, Architectural Pseudocode

Part J contains pseudocode that describes various features of the ARM architecture. It contains the following chapter:

**Chapter J1** ***ARMv8 Pseudocode***

Read this for the pseudocode definitions that describe various features of the ARMv8 architecture, for operation in AArch64 state and in AArch32 state.

## Part K, Appendixes

This manual contains the following appendixes:

**Appendix K1** ***Architectural Constraints on UNPREDICTABLE behaviors***

Read this for a description of the architecturally-required constraints on UNPREDICTABLE behaviors in the ARMv8 architecture, including AArch32 behaviors that were UNPREDICTABLE in previous versions of the architecture.

**Appendix K2** ***Recommended External Debug Interface***

Read this for a description of the recommended external debug interface.

**Note**

This description is not part of the ARM architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

**Appendix K3** ***Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events***

Read this for a description of ARM recommendations for the use of the IMPLEMENTATION DEFINED event numbers.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Note**

This description is not part of the ARM architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

**Appendix K4 *Recommendations for reporting memory attributes on an interconnect***

Read this for the ARM recommendations about how the architectural memory attributes are reported on an interconnect.

**Appendix K5 *ARMv8 Changes to the T32 and A32 Instruction Sets***

Read this for a summary of the changes that are introduced to the T32 and A32 instruction sets in ARMv8.

**Appendix K6 *Legacy Instruction Syntax for AArch32 Instruction Sets***

Read this for information about the pre-UAL syntax of the AArch32 instruction sets, which can still be valid for the A32 instruction set.

**Appendix K7 *Address translation examples***

Read this for examples of translation table lookups using the translation regimes described in Chapter D4 *The AArch64 Virtual Memory System Architecture* and Chapter G4 *The AArch32 Virtual Memory System Architecture*.

**Appendix K8 *Example OS Save and Restore Sequences***

Read this for software examples that perform the OS Save and Restore sequences for an ARMv8 debug implementation.

**Note**

Chapter H6 *Debug Reset and Powerdown Support* describes the OS Save and Restore mechanism.

**Appendix K9 *Recommended Upload and Download Processes for External Debug***

Read this for information about implementing and using the ARM architecture.

**Note**

This description is not part of the ARM architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

**Appendix K10 *Barrier Litmus Tests***

Read this for examples of the use of barrier instructions provided by the ARMv8 architecture.

**Note**

This description is not part of the ARM architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

**Appendix K11 *ARM Pseudocode Definition***

Read this for definitions of the AArch32 pseudocode.

**Appendix K12 *Pseudocode Index***

Read this for an index of the pseudocode.

**Appendix K13 *Registers Index***

Read this for an alphabetic and functional index of AArch32 and AArch64 registers, and memory-mapped registers.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Conventions

The following sections describe conventions that this book can use:

- *Typographic conventions*.
- *Signals*.
- *Numbers*.
- *Pseudocode descriptions*.
- *Assembler syntax descriptions* on page xxiv.

## Typographic conventions

The typographical conventions are:

***italic*** Introduces special terminology, and denotes citations.

**bold** Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace` Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, and are defined in the *Glossary*.

**Colored text** Indicates a link. This can be:

- A URL, for example `http://infocenter.arm.com`.
- A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, *Assembler syntax descriptions* on page xxiv.
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term, for example *Simple sequential execution* or SCTLR.

## Signals

In general this specification does not define hardware signals, but it does include some signal examples and recommendations. The signal conventions are:

**Signal level** The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

**Lower-case n** At the start or end of a signal name denotes an active-LOW signal.

## Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a `monospace` font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

## Pseudocode descriptions

This manual uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in `monospace` font, and is described in Appendix K11 *ARM Pseudocode Definition*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler syntax descriptions

This manual contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font, and use the conventions described in *Structure of the A64 assembler language* on page C1-121, Appendix K11 *ARM Pseudocode Definition*, and *Pseudocode operators and keywords* on page K12-5678.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Additional reading

This section lists relevant publications from ARM and third parties.

See the Infocenter http://infocenter.arm.com, for access to ARM documentation.

## ARM publications

- *ARM® AMBA® 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1*, (ARM IHI 0032B).
- *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *ARM® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2* (ARM IHI 0031).
- *ARM® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
- *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).
- *ARM® CoreSight™ SoC Technical Reference Manual* (ARM DDI 0480).
- *ARM® CoreSight™ v2.0 Architecture Specification* (ARM IHI 0029).
- *ARM® Procedure Call Standard for the ARM 64-bit Architecture* (ARM IHI 0055).

## Other publications

The following publications are referred to in this manual, or provide more information:

- *Announcing the Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197, November 2001.
- IEEE Std 754-2008, *IEEE Standard for Floating-point Arithmetic*, August 2008.
- IEEE Std 754-1985, *IEEE Standard for Floating-point Arithmetic*, March 1985.
- *Secure Hash Standard (SHA)*, Federal Information Processing Standards Publication 180-2, August 2002.
- *The Galois/Counter Mode of Operation*, McGraw, D. and Viega, J., Submission to NIST Modes of Operation Process, January 2004.
- *Memory Consistency Models for Shared Memory-Multiprocessors*, Gharachorloo, Kourosh, 1995, Stanford University Technical Report CSL-TR-95-685.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Feedback

ARM welcomes feedback on its documentation.

## Feedback on this manual

If you have comments on the content of this manual, send e-mail to `errata@arm.com`. Give:

- The title.
- The number, ARM DDI 0487A.h.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Part A

## ARMv8 Architecture Introduction and Overview

# Chapter A1
# Introduction to the ARMv8 Architecture

This chapter introduces the ARM architecture and contains the following sections:


- *About the ARM architecture* on page A1-30.
- *Architecture profiles* on page A1-32.
- *ARMv8 architectural concepts* on page A1-33.
- *Supported data types* on page A1-36.
- *Floating-point and Advanced SIMD support* on page A1-46.
- *Cryptographic Extension* on page A1-52.
- *The ARM memory model* on page A1-53.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## A1.1 About the ARM architecture

The ARM architecture described in this Architecture Reference Manual defines the behavior of an abstract machine, referred to as a *processing element*, often abbreviated to *PE*. Implementations compliant with the ARM architecture must conform to the described behavior of the processing element. It is not intended to describe how to build an implementation of the PE, nor to limit the scope of such implementations beyond the defined behaviors.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with the ARM architecture must be the same as a simple sequential execution of the program on the processing element. This programmer-visible behavior does not include the execution time of the program.

The ARM Architecture Reference Manual also describes rules for software to use the processing element.

The ARM architecture includes definitions of:

- An associated debug architecture, see:
  - Chapter D2 *AArch64 Self-hosted Debug*.
  - Chapter G2 *AArch32 Self-hosted Debug*.
  - Part H of this manual, *External Debug* on page 4813.
- Associated trace architectures, that define trace macrocells that implementers can implement with the associated processor hardware. For more information see the *Embedded Trace Macrocell Architecture Specification.*

The ARM architecture is a *Reduced Instruction Set Computer* (RISC) architecture with the following RISC architecture features:

- A large uniform register file.
- A *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents.
- Simple addressing modes, with all load/store addresses determined from register contents and instruction fields only.

The architecture defines the interaction of the PE with memory, including caches, and includes a memory translation system. It also describes how multiple PEs interact with each other and with other observers in a system.

This document defines the ARMv8-A architecture *profile*. See *Architecture profiles* on page A1-32 for more information.

The ARM architecture supports implementations across a wide range of performance points. Implementation size, performance, and very low power consumption are key attributes of the ARM architecture.

An important feature of the ARMv8 architecture is backwards compatibility, combined with the freedom for optimal implementation in a wide range of standard and more specialized use cases. The ARMv8 architecture supports:

- A 64-bit Execution state, AArch64.
- A 32-bit Execution state, AArch32, that is compatible with previous versions of the ARM architecture.

**Note**

- The AArch32 Execution state is compatible with the ARMv7-A architecture profile, and enhances that profile to support some features included in the AArch64 Execution state.
- This document describes only the ARMv8-A architecture profile. For the behaviors required by the ARMv7-A and ARMv7-R architecture profiles, see the *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Both Execution states support SIMD and floating-point instructions:

- AArch32 state provides:
  - SIMD instructions in the base instruction sets, that operate on the 32-bit general-purpose registers.
  - Advanced SIMD instructions that operate on registers in the *SIMD and floating-point register* (SIMD&FP register) file.
  - Floating-point instructions that operate on registers in the SIMD&FP register file.
- AArch64 state provides:
  - Advanced SIMD instructions that operate on registers in the SIMD&FP register file.
  - Floating-point instructions that operate on registers in the SIMD&FP register file.

**Note**

See *Conventions* on page xxiii for information about conventions used in this manual, including the use of SMALL CAPITALS for particular terms that have ARM-specific meanings that are defined in the *Glossary*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## A1.2 Architecture profiles

The ARM architecture has evolved significantly since its introduction, and ARM continues to develop it. Eight major versions of the architecture have been defined to date, denoted by the version numbers 1 to 8. Of these, the first three versions are now obsolete.

The generic names AArch64 and AArch32 describe the 64-bit and 32-bit Execution states:

**AArch64** Is the 64-bit Execution state, meaning addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

**AArch32** Is the 32-bit Execution state, meaning addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

**Note**

The *Base instruction set* comprises the supported instructions other than the Advanced SIMD and floating-point instructions.

See sections *Execution state* on page A1-33 and *The ARM instruction sets* on page A1-34 for more information.

ARM defines three architecture profiles:

**A** Application profile, described in this manual:

- Supports a *Virtual Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU).

  **Note**

  An ARMv8-A implementation can be called an AArchv8-A implementation.

- Supports the A64, A32, and T32 instruction sets.

**R** Real-time profile:

- Supports a *Protected Memory System Architecture* (PMSA) based on a *Memory Protection Unit* (MPU).
- Supports the A32 and T32 instruction sets.

**M** Microcontroller profile:

- Implements a programmers' model designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.
- Implements a variant of the R-profile PMSA.
- Supports a variant of the T32 instruction set.

**Note**

This Architecture Reference Manual describes only the ARMv8-A profile.

For information about the R and M architecture profiles, and earlier ARM architecture versions see:

- The *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.
- The *ARM®v7-M Architecture Reference Manual*.
- The *ARM®v6-M Architecture Reference Manual*.

### A1.2.1 Debug architecture version

The ARM Debug architecture is fully integrated with the architecture, and does not have a separate version number.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## A1.3 ARMv8 architectural concepts

ARMv8 introduces major changes to the ARM architecture, while maintaining a high level of consistency with previous versions of the architecture. The ARMv8 Architecture Reference Manual includes significant changes in the terminology used to describe the architecture, and this section introduces both the ARMv8 architectural concepts and the associated terminology.

The following subsections describe key ARMv8 architectural concepts. Each section introduces the corresponding terms that are used to describe the architecture:

- *Execution state*.
- *The ARM instruction sets* on page A1-34.
- *System registers* on page A1-34.
- *ARMv8 Debug* on page A1-35.

### A1.3.1 Execution state

The Execution state defines the PE execution environment, including:

- The supported register widths.
- The supported instruction sets.
- Significant aspects of:
  - The exception model.
  - The *Virtual Memory System Architecture* (VMSA).
  - The programmers' model.

The Execution states are:

**AArch64** The 64-bit Execution state. This Execution state:

- Provides 31 64-bit general-purpose registers, of which X30 is used as the procedure link register.
- Provides a 64-bit *program counter* (PC), *stack pointer*s (SPs), and *exception link registers* (ELRs).
- Provides 32 128-bit registers for SIMD vector and scalar floating-point support.
- Provides a single instruction set, A64. For more information, see *The ARM instruction sets* on page A1-34.
- Defines the ARMv8 Exception model, with up to four Exception levels, EL0 - EL3, that provide an *execution privilege* hierarchy, see *Exception levels* on page D1-1514.
- Provides support for 64-bit *virtual addressing*. For more information, including the limits on address ranges, see Chapter D4 *The AArch64 Virtual Memory System Architecture*.
- Defines a number of *Process state* (PSTATE) elements that hold PE state. The A64 instruction set includes instructions that operate directly on various PSTATE elements.
- Names each System register using a suffix that indicates the lowest Exception level at which the register can be accessed.

**AArch32** The 32-bit Execution state. This Execution state:

- Provides 13 32-bit general-purpose registers, and a 32-bit PC, SP, and *link register* (LR). The LR is used as both an ELR and a procedure link register.
  Some of these registers have multiple *banked* instances for use in different PE *modes*.
- Provides a single ELR, for exception returns from Hyp mode.
- Provides 32 64-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides two instruction sets, A32 and T32. For more information, see *The ARM instruction sets* on page A1-34.
- Supports the ARMv7-A exception model, based on *PE modes*, and maps this onto the ARMv8 Exception model, that is based on the Exception levels.
- Provides support for 32-bit virtual addressing.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Defines a number of *Process state* (PSTATE) elements that hold PE state. The A32 and T32 instruction sets include instructions that operate directly on various PSTATE elements, and instructions that access PSTATE by using the *Application Program Status Register* (APSR) or the *Current Program Status Register* (CPSR).

Later subsections give more information about the different properties of the Execution states.

Transitioning between the AArch64 and AArch32 Execution states is known as *interprocessing*. The PE can move between Execution states only on a change of Exception level, and subject to the rules given in *Interprocessing* on page D1-1619. This means different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.

### A1.3.2 The ARM instruction sets

In ARMv8 the possible instruction sets depend on the Execution state:

**AArch64** AArch64 state supports only a single instruction set, called A64. This is a fixed-length instruction set that uses 32-bit instruction encodings.

For information on the A64 instruction set, see Chapter C3 *A64 Instruction Set Overview*.

**AArch32** AArch32 state supports the following instruction sets:

**A32** This is a fixed-length instruction set that uses 32-bit instruction encodings.

**T32** This is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings.

In previous documentation, these instruction sets were called the ARM and Thumb instruction sets. ARMv8 extends each of these instruction sets. In AArch32 state, the Instruction set state determines the instruction set that the PE executes.

For information on the A32 and T32 instruction sets, see Chapter F1 *The AArch32 Instruction Sets Overview*.

The ARMv8 instruction sets support SIMD and scalar floating-point instructions. See *Floating-point and Advanced SIMD support* on page A1-46.

### A1.3.3 System registers

System registers provide control and status information of architected features.

The System registers use a standard naming format: <register_name>.<bit_field_name> to identify specific registers as well as control and status bits within a register.

Bits can also be described by their numerical position in the form <register_name>[x:y] or the generic form bits[x:y].

In addition, in AArch64 state, most register names include the lowest Exception level that can access the register as a suffix to the register name:

- <register_name>_ELx, where x is 0, 1, 2, or 3.

For information about Exception levels, see *Exception levels* on page D1-1514.

The System registers comprise:

- The following registers that are described in this manual:
  - General system control registers.
  - Debug registers.
  - Generic Timer registers.
  - Optionally, Performance Monitor registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Optionally, one or more of the following groups of registers that are defined in other ARM architecture specifications:
  - Trace System registers, as defined in the *Embedded Trace Macrocell Architecture Specification, ETMv4*.
  - *Generic Interrupt Controller* (GIC) System registers, see *The ARM Generic Interrupt Controller System registers*.

For information about the AArch64 System registers, see Chapter D7 *AArch64 System Register Descriptions*.

For information about the AArch32 System registers, see Chapter G6 *AArch32 System Register Descriptions*.

### The ARM Generic Interrupt Controller System registers

From version 3 of the ARM Generic Interrupt Controller architecture, GICv3, the GIC architecture specification defines a System register interface to some of its functionality. The System register summaries in this manual include these registers, see:

- *About the GIC System registers* on page C5-287, for more information about the AArch64 GIC System registers.
- *Generic Interrupt Controller System registers, functional groups* on page G4-4175, for more information about the AArch32 GIC System registers.

These sections give only short overviews of the GIC System registers. For more information, including descriptions of the registers, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).

**Note**

The programmers' model for earlier versions of the GIC architecture is wholly memory-mapped.

## A1.3.4 ARMv8 Debug

ARMv8 supports the following:

**Self-hosted debug**

In this model, the PE generates *debug exceptions*. Debug exceptions are part of the ARMv8 Exception model.

**External debug**

In this model, *debug events* cause the PE to enter *Debug state*. In Debug state the PE is controlled by an external debugger.

All ARMv8 implementations support both models. The model chosen by a particular user depends on the debug requirements during different stages of the design and development life cycle of the product. For example, external debug might be used during debugging of the hardware implementation and OS bring-up, and self-hosted debug might be used during application development.

For more information about self-hosted debug:

- In AArch64 state, see Chapter D2 *AArch64 Self-hosted Debug*.
- In AArch32 state, see Chapter G2 *AArch32 Self-hosted Debug*.

For more information about external debug, see *Part H External Debug*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## A1.4 Supported data types

The ARMv8 architecture supports the following integer data types:

**Byte** 8 bits.
**Halfword** 16 bits.
**Word** 32 bits.
**Doubleword** 64 bits.
**Quadword** 128 bits.

The architecture also supports the following floating-point data types:

- Half-precision, see *Half-precision floating-point formats* on page A1-40 for details.
- Single-precision, see *Single-precision floating-point format* on page A1-42 for details.
- Double-precision, see *Double-precision floating-point format* on page A1-43 for details.

It also supports:

- Fixed-point interpretation of words and doublewords. See *Fixed-point format* on page A1-44.
- Vectors, where a register holds multiple elements, each of the same data type. See *Vector formats* on page A1-37 for details.

The ARMv8 architecture provides two register files:

- A general-purpose register file.
- A SIMD&FP register file.

In each of these, the possible register widths depend on the Execution state.

In AArch64 state:

- A general-purpose register file contains 64-bit registers:
  - Many instructions can access these registers as 64-bit registers or as 32-bit registers, using only the bottom 32 bits.
- A SIMD&FP register file contains 128-bit registers:
  - The quadword integer data types only apply to the SIMD&FP register file.
  - The floating-point data types only apply to the SIMD&FP register file.
  - While the AArch64 vector registers support 128-bit vectors, the effective vector length can be 64-bits or 128-bits depending on the A64 instruction encoding used, see *Instruction Mnemonics* on page C1-121.

For more information on the register files in AArch64, see *Registers in AArch64 Execution state* on page B1-59.

In AArch32 state:

- A general-purpose register file contains 32-bit registers:
  - Two 32-bit registers can support a doubleword.
  - Vector formatting is supported, see Figure A1-4 on page A1-40.
- A SIMD&FP register file contains 64-bit registers:
  - AArch32 state does not support quadword integer or floating-point data types.

**Note**

Two consecutive 64-bit registers can be used as a 128-bit register.

For more information on the register files in AArch32, see *The general-purpose registers, and the PC, in AArch32 state* on page E1-2316.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### A1.4.1 Vector formats

In an implementation that includes the SIMD instructions that operate on the SIMD&FP register file, a register can hold one or more packed elements, all of the same size and type. The combination of a register and a data type describes a vector of elements. The vector is considered to be an array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the size of the register.

Vector indices are in the range 0 to (number of elements – 1). An index of 0 refers to the least significant end of the vector.

#### Vector formats in AArch64 state

In AArch64 state, the SIMD&FP registers can be referred to as V*n*, where *n* is a value from 0 to 31.

The SIMD&FP registers support three data formats for loads, stores and data processing operations:

- A single, scalar, element in the least significant bits of the register.
- A 64-bit vector of byte, halfword, or word elements.
- A 128-bit vector of byte, halfword, word or doubleword elements.

The element sizes are defined in Table A1-1 with the vector format described as:

- For a 128-bit vector: Vn{.2D, .4S, .8H, .16B}.
- For a 64-bit vector: Vn{.1D, .2S, .4H, .8B}.

**Table A1-1 SIMD elements**

| Mnemonic | Size |
|---|---|
| B | 8 bits |
| H | 16 bits |
| S | 32 bits |
| D | 64 bits |

Figure A1-1 on page A1-38 shows the SIMD vectors in AArch64 state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

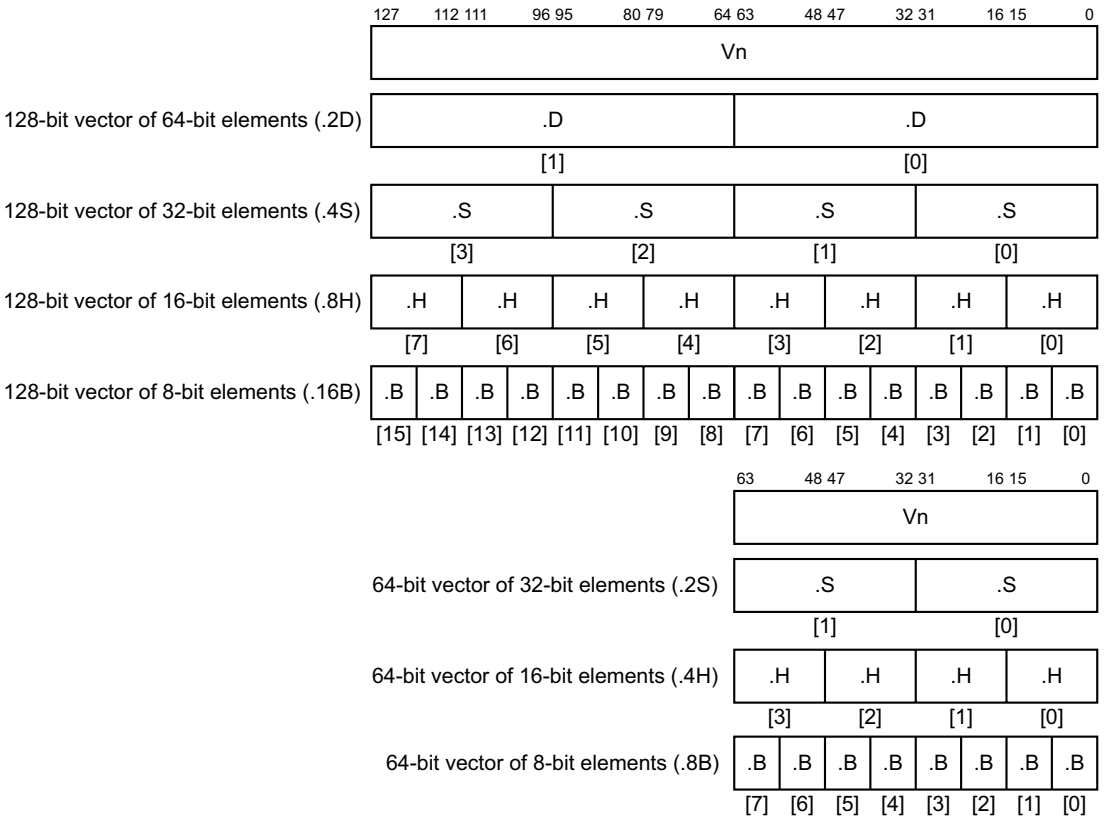



**Figure A1-1 SIMD vectors in AArch64 state**

### Vector formats in AArch32 state

Table A1-2 shows the available formats. Each instruction description specifies the data types that the instruction supports.

**Table A1-2 Advanced SIMD data types in AArch32**

| Data type specifier | Meaning |
|---|---|
| `.<size>` | Any element of <size> bits |
| `.F<size>` | Floating-point number of <size> bits |
| `.I<size>` | Signed or unsigned integer of <size> bits |
| `.P<size>` | Polynomial over {0, 1} of degree less than <size> |
| `.S<size>` | Signed integer of <size> bits |
| `.U<size>` | Unsigned integer of <size> bits |

*Polynomial arithmetic over {0, 1}* on page A1-45 describes the polynomial data type.

The `.F16` data type is the half-precision data type selected by the FPSCR.AHP bit.

The `.F32` data type is the ARM standard single-precision floating-point data type, see *Single-precision floating-point format* on page A1-42.

The instruction definitions use a data type specifier to define the data types appropriate to the operation. Figure A1-2 on page A1-39 shows the hierarchy of the Advanced SIMD data types.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| | | |
|---|---|---|
| .8 | .I8 | .S8 |
| | | .U8 |
| | .P8 | |
| | - | |
| .16 | .I16 | .S16 |
| | | .U16 |
| | .P16 † | |
| | .F16 | |
| .32 | .I32 | .S32 |
| | | .U32 |
| | - | |
| | .F32 | |
| .64 | .I64 | .S64 |
| | | .U64 |
| | .P64 ‡ | |
| | - | |

† Output format only. See VMULL instruction description.

‡ Available only if the Cyptographic Extension is implemented. See VMULL instruction description.

**Figure A1-2 Advanced SIMD data type hierarchy in AArch32**

For example, a multiply instruction must distinguish between integer and floating-point data types.

An integer multiply instruction that generates a double-width (long) result must specify the input data types as signed or unsigned. However, some integer multiply instructions use modulo arithmetic, and therefore do not have to distinguish between signed and unsigned inputs.

Figure A1-3 on page A1-40 shows the Advanced SIMD vectors in AArch32 state.

**Note**

In AArch32 state, a pair of even and following odd numbered doubleword registers can be concatenated and treated as a single quadword register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

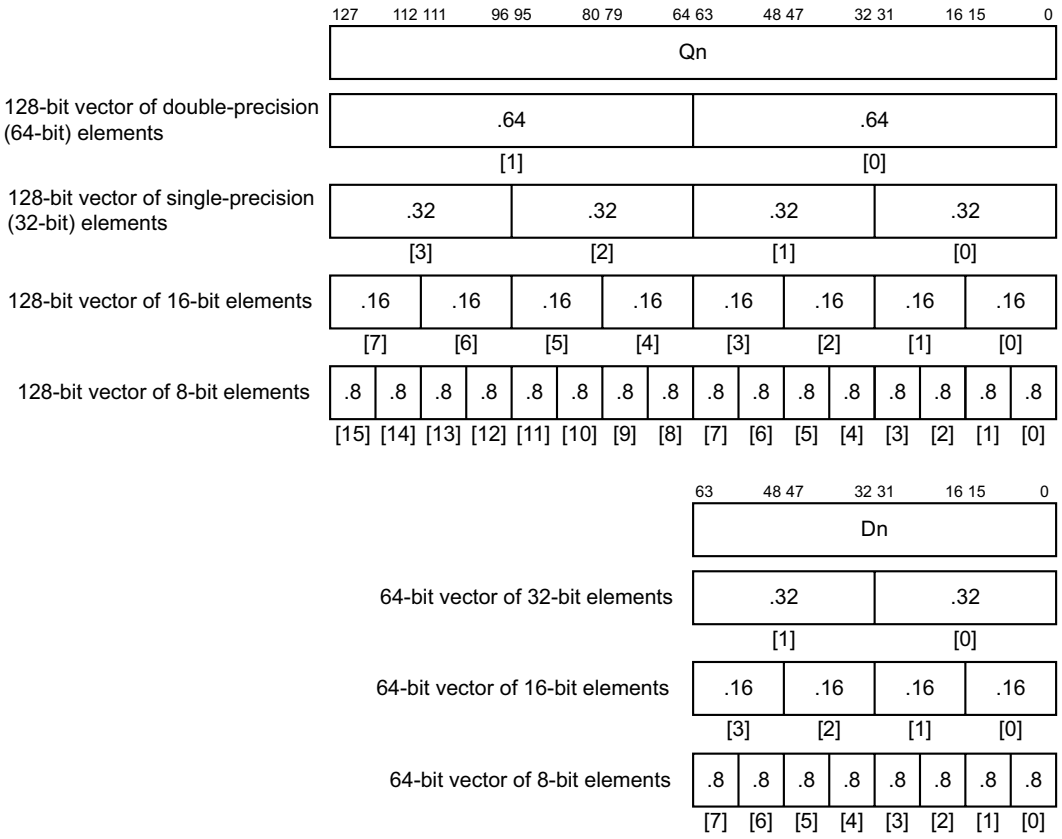



**Figure A1-3 Advanced SIMD vectors in AArch32**

The AArch32 general-purpose registers support vectors formats for use by the SIMD instructions in the Base instruction set. Figure A1-4 shows these formats, that means that a general-purpose register can be treated as either two halfwords or four bytes.

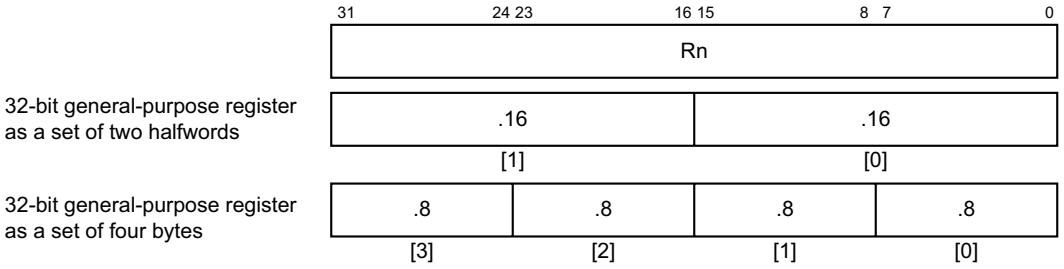



**Figure A1-4 Vector formatting in AArch32**

### A1.4.2 Half-precision floating-point formats

ARMv8 supports two half-precision floating-point formats:

- IEEE half-precision, as described in the IEEE 754-2008 standard.
- Alternative half-precision.

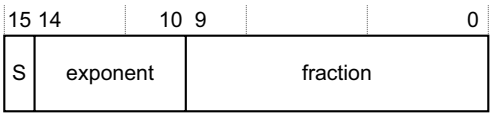
**Note**

Half-precision floating-point formats can only be converted to and from other floating-point formats. They cannot be used in any other data processing operations. This applies to both AArch32 state and AArch64 state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The description of IEEE half-precision includes ARM-specific details that are left open by the standard, and is only an introduction to the formats and to the values they can contain. For more information, especially on the handling of infinities, NaNs and signed zeros, see the IEEE 754 standard.

For both half-precision floating-point formats, the layout of the 16-bit format is the same. The format is:




The interpretation of the format depends on the value of the exponent field, bits[14:10] and on which half-precision format is being used.

**0 < exponent <** `0x1F`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(\text{exponent}-15)} \times (1.\text{fraction})$

The minimum positive normalized number is $2^{-14}$, or approximately $6.104 \times 10^{-5}$.

The maximum positive normalized number is $(2 - 2^{-10}) \times 2^{15}$, or 65504.

Larger normalized numbers can be expressed using the alternative format when the exponent == `0x1F`.

**exponent == 0**

The value is either a zero or a denormalized number, depending on the fraction bits:

**fraction == 0**

The value is a zero. There are two distinct zeros:

**+0** when S==0

**–0** when S==1.

**fraction != 0**

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-14} \times (0.\text{fraction})$

The minimum positive denormalized number is $2^{-24}$, or approximately $5.960 \times 10^{-8}$.

**exponent ==** `0x1F`

The value depends on which half-precision format is being used:

**IEEE half-precision**

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

**fraction == 0**

The value is an infinity. There are two distinct infinities:

**+infinity** When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

**-infinity** When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

**fraction != 0**

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[9]:

**bit[9] == 0** The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

**bit[9] == 1** The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Alternative half-precision**

The value is a normalized number and is equal to:

$-1^S \times 2^{16} \times (1.\text{fraction})$

The maximum positive normalized number is $(2-2^{-10}) \times 2^{16}$ or 131008.

## A1.4.3 Single-precision floating-point format

The single-precision floating-point format is as defined by the IEEE 754 standard.

This description includes ARM-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs and signed zeros, see the IEEE 754 standard.

A single-precision value is a 32-bit word with the format:




The interpretation of the format depends on the value of the exponent field, bits[30:23]:

**0 < exponent < `0xFF`**

The value is a *normalized number* and is equal to:

$(-1)^S \times 2^{(\text{exponent} - 127)} \times (1.\text{fraction})$

The minimum positive normalized number is $2^{-126}$, or approximately $1.175 \times 10^{-38}$.

The maximum positive normalized number is $(2 - 2^{-23}) \times 2^{127}$, or approximately $3.403 \times 10^{38}$.

**exponent == 0**

The value is either a zero or a *denormalized number*, depending on the fraction bits:

**fraction == 0**

The value is a zero. There are two distinct zeros:

**+0** When S==0.

**–0** When S==1.

These usually behave identically. In particular, the result is *equal* if +0 and –0 are compared as floating-point numbers. However, they yield different results in some circumstances. For example, the sign of the infinity produced as the result of dividing by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words.

**fraction != 0**

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-126} \times (0.\text{fraction})$

The minimum positive denormalized number is $2^{-149}$, or approximately $1.401 \times 10^{-45}$.

Denormalized numbers are always flushed to zero in AArch32 Advanced SIMD processing. They are optionally flushed to zero in floating-point processing and AArch64 SIMD. For details see *Flush-to-zero* on page A1-49.

**exponent == `0xFF`**

The value is either an *infinity* or a *Not a Number* (NaN), depending on the fraction bits:

**fraction == 0**

The value is an infinity. There are two distinct infinities:

**+infinity** When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

**-infinity** When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**fraction != 0**

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[22]:

**bit[22] == 0**

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

**bit[22] == 1**

The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

For details of the *default NaN* see *NaN handling and the Default NaN* on page A1-50.

**Note**

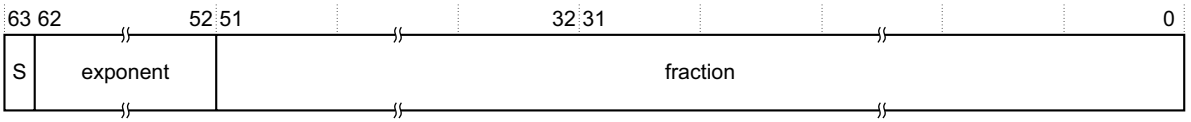
NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

### A1.4.4 Double-precision floating-point format

The double-precision floating-point format is as defined by the IEEE 754 standard. Double-precision floating-point is supported by both floating-point and SIMD instructions in AArch64 state, and only by floating-point instructions in AArch32 state.

This description includes implementation-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs and signed zeros, see the IEEE 754 standard.

A double-precision value is a 64-bit doubleword, with the format:




Double-precision values represent numbers, infinities and NaNs in a similar way to single-precision values, with the interpretation of the format depending on the value of the exponent:

**0 < exponent <** `0x7FF`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-1023)} \times (1.fraction)$

The minimum positive normalized number is $2^{-1022}$, or approximately $2.225 \times 10^{-308}$.

The maximum positive normalized number is $(2 - 2^{-52}) \times 2^{1023}$, or approximately $1.798 \times 10^{308}$.

**exponent == 0**

The value is either a zero or a denormalized number, depending on the fraction bits:

**fraction == 0**

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

**+0** when S==0

**–0** when S==1.

**fraction != 0**

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-1022} \times (0.fraction)$

The minimum positive denormalized number is $2^{-1074}$, or approximately $4.941 \times 10^{-324}$.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Optionally, denormalized numbers are flushed to zero in floating-point calculations. For details see *Flush-to-zero* on page A1-49.

**exponent ==** `0x7FF`

The value is either an infinity or a NaN, depending on the fraction bits:

**fraction == 0**

The value is an infinity. As for single-precision, there are two infinities:

**+infinity** When S==0.

**-infinity** When S==1.

**fraction != 0**

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[51] of the doubleword:

**bit[51] == 0**

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

**bit[51] == 1**

The NaN is a quiet NaN. The sign bit and the remaining fraction bits can take any value.

For details of the *default NaN* see *NaN handling and the Default NaN* on page A1-50.

**Note**

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

### A1.4.5 Fixed-point format

Fixed-point formats are used only for conversions between floating-point and fixed-point values. They apply to general-purpose registers.

Fixed-point values can be signed or unsigned, and can be 16-bit or 32-bit. Conversion instructions take an argument that specifies the number of fraction bits in the fixed-point number. That is, it specifies the position of the binary point.

### A1.4.6 Conversion between floating-point and fixed-point values

ARMv8 supports the conversion of a scalar floating-point to or from a signed or unsigned fixed-point value in a general-purpose register.

The instruction argument #fbits indicates that the general-purpose register holds a fixed-point number with fbits bits after the binary point, where fbits is in the range 1 to 64 for a 64-bit general-purpose register, or 1 to 32 for a 32-bit general-purpose register.

More specifically:

- For a 64-bit register $X_d$:
  - The integer part is $X_d[63:\#fbits]$.
  - The fractional part is $X_d[(\#fbits-1):0]$.
- For a 32-bit register $W_d$ or $R_d$:
  - The integer part is $W_d[31:\#fbits]$ or $R_d[31:\#fbits]$.
  - The fractional part is $W_d[(\#fbits-1):0]$ or $R_d[(\#fbits-1):0]$.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These instructions might generate the following exceptions:

**Invalid Operation** When the floating-point input is NaN or Infinity or when a numerical value cannot be represented within the destination register.

**Inexact** When the numeric result differs from the input.

**Input Denormal** When flush-to-zero mode is enabled and the denormal input is replaced by a zero.

**Note**

An out of range fixed-point result is saturated to the destination size.

### A1.4.7 Polynomial arithmetic over {0, 1}

Some SIMD instructions that operate on SIMD&FP registers can operate on polynomials over {0, 1}, see *Supported data types* on page A1-36. The polynomial data type represents a polynomial in x of the form $b_{n-1}x^{n-1} + \ldots + b_1x + b_0$ where $b_k$ is bit[k] of the value.

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic:

- $0 + 0 = 1 + 1 = 0$
- $0 + 1 = 1 + 0 = 1$
- $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$
- $1 \times 1 = 1$.

That is:

- Adding two polynomials over {0, 1} is the same as a bitwise exclusive OR.
- Multiplying two polynomials over {0, 1} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

A64, A32 and T32 provide instructions for performing polynomial multiplication of 8-bit values.

- For AArch32, see *VMUL (integer and polynomial)* on page F7-3502 and *VMULL (integer and polynomial)* on page F7-3508.
- For AArch64, see *PMUL* on page C7-1177 and *PMULL, PMULL2* on page C7-1179.

The Cryptographic Extension adds the ability to perform long polynomial multiplies of 64-bit values. See *PMULL, PMULL2* on page C7-1179.

#### Pseudocode description of polynomial multiplication

In pseudocode, polynomial addition is described by the EOR operation on bitstrings.

Polynomial multiplication is described by the `PolynomialMult()` function defined in Chapter J1 *ARMv8 Pseudocode*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## A1.5 Floating-point and Advanced SIMD support

**Note**

In AArch32 state, the SIMD instructions that operate on SIMD&FP registers are always described as the Advanced SIMD instructions, to distinguish them from the SIMD instructions in the base instruction sets, that operate on the 32-bit general-purpose registers. The A64 instruction set does not provide any SIMD instructions that operate on the general-purpose registers, and therefore some AArch64 state descriptions use SIMD as a synonym for Advanced SIMD. Unless the context clearly indicates otherwise, this section describes the support for SIMD instructions that operate on SIMD&FP registers.

ARMv8 can support the following levels of support for floating-point and Advanced SIMD instructions:

- Full floating-point and SIMD support without exception trapping.
- Full floating-point and SIMD support with exception trapping.
- No floating-point or SIMD support. This option is licensed only for implementations targeting specialised markets.

  **Note**

  All systems that support standard operating systems with rich application environments provide hardware support for floating-point and Advanced SIMD. It is a requirement of the ARM Procedure Call Standard for AArch64, see *Procedure Call Standard for the ARM 64-bit Architecture*.

ARMv8 supports single-precision (32-bit) and double-precision (64-bit) floating-point data types and arithmetic as defined by the IEEE 754 floating-point standard. It also supports the half-precision (16-bit) floating-point data type for data storage only, by supporting conversions between single-precision and half-precision data types and double-precision and half-precision data types.

The SIMD instructions provide packed *Single Instruction Multiple Data (SIMD)* and single-element scalar operations, and support:

- Single-precision and double-precision arithmetic in AArch64 state.
- Single-precision arithmetic only in AArch32 state.

Floating-point support in AArch64 state SIMD is IEEE 754-2008 compliant with:

- Configurable rounding modes.
- Configurable Default NaN behavior.
- Configurable Flush-to-zero behavior.

Floating-point computation using AArch32 Advanced SIMD instructions remains unchanged from ARMv7. A32 and T32 Advanced SIMD floating-point always uses ARM standard floating-point arithmetic and performs IEEE 754 floating-point arithmetic with the following restrictions:

- Denormalized numbers are flushed to zero, see *Flush-to-zero* on page A1-49.
- Only default NaNs are supported, see *NaN handling and the Default NaN* on page A1-50.
- The Round to Nearest rounding mode is used.
- Untrapped floating-point exception handling is used for all floating-point exceptions.

ARMv8 introduces new instructions for AArch32 state:

- Floating-point selection, see *VSELEQ, VSELGE, VSELGT, VSELVS* on page F7-3661.
- Floating-point maximum and minimum numbers, see *VMAXNM* on page F7-3440 and *VMINNM* on page F7-3447.
- Floating-point integer conversions with directed rounding modes, see:
  - *VCVTA (Advanced SIMD)* on page F7-3343 and *VCVTA (floating-point)* on page F7-3345.
  - *VCVTM (Advanced SIMD)* on page F7-3350 and *VCVTM (floating-point)* on page F7-3352.
  - *VCVTN (Advanced SIMD)* on page F7-3354 and *VCVTN (floating-point)* on page F7-3356.
  - *VCVTP (Advanced SIMD)* on page F7-3358 and *VCVTP (floating-point)* on page F7-3360.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Floating-point round to integral floating-point, see:
  - *VRINTA (Advanced SIMD)* on page F7-3617 and *VRINTA (floating-point)* on page F7-3619.
  - *VRINTM (Advanced SIMD)* on page F7-3621 and *VRINTM (floating-point)* on page F7-3623.
  - *VRINTN (Advanced SIMD)* on page F7-3625 and *VRINTN (floating-point)* on page F7-3627.
  - *VRINTP (Advanced SIMD)* on page F7-3629 and *VRINTP (floating-point)* on page F7-3631.
  - *VRINTR* on page F7-3633.
  - *VRINTX (Advanced SIMD)* on page F7-3635 and *VRINTX (floating-point)* on page F7-3637.
  - *VRINTZ (Advanced SIMD)* on page F7-3639 and *VRINTZ (floating-point)* on page F7-3641.
- Floating-point conversions between half-precision and double-precision, see *VCVTB* on page F7-3347 and *VCVTT* on page F7-3365.

If floating-point exception trapping is supported, floating-point exceptions, such as overflow or division by zero, can be handled without trapping. This applies to both floating-point and SIMD operations. When handled in this way, a floating-point exception causes a cumulative status register bit to be set to 1 and a default result to be produced by the operation. For more information about floating-point exceptions, see *Floating-point exception traps* on page D1-1568.

In AArch64 state, the following registers control floating-point operation and return floating-point status information:

- The Floating-Point Control Register, FPCR, controls:
  - The half-precision format where applicable, FPCR.AHP bit.
  - Default NaN behavior, FPCR.DN bit.
  - Flush to zero behavior, FPCR.FZ bit.
  - Rounding mode support, FPCR.Rmode field.
  - Len and Stride fields associated with AArch32 execution, and only supported for a context save and restore in AArch64. These fields are obsolete in ARMv8 and can be implemented as RAZ/WI. If they are implemented as RW and are programmed to a nonzero value, they make some AArch32 floating-point instructions UNDEFINED.
  - Floating-point exception trap controls, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits, see *Floating-point exception traps* on page D1-1568. In an implementation that does not support trapping of floating-point exceptions these bits are RES0.
- The Floating-Point Status Register, FPSR, provides:
  - Cumulative floating-point exceptions flags, FPSR.{IDC, IXC, UFC, OFC, DZC, IOC and QC}.
  - TheAArch32 floating-point comparison flags {N,Z,C,V}. These bits are RES0 if AArch32 floating-point is not implemented.

    **Note**

    In AArch64, the process state flags, PSTATE.{N,Z,C,V} are used for all data processing compares and any associated conditional execution.

AArch32 state provides a single Floating-Point Status and Control Register, FPSCR, combining the FPCR and FPSR fields.

For system level information about the SIMD and floating-point support, see *Advanced SIMD and floating-point support* on page G1-3841.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### A1.5.1 Instruction support

The floating-point and SIMD support includes the following types of instructions:

- Load and store for single elements and vectors of multiple elements.

  **Note**

  Single elements are also referred to as scalar elements.

- Data processing on single and multiple elements for both integer and floating-point data types.
- Floating-point conversion:
  - Half-precision, single-precision, and double-precision conversions.
  - Single-precision, double-precision, and fixed point integer conversions.
  - Single-precision, double-precision, and integer conversions.
- Floating-point rounding.

For more information on the floating-point and SIMD instructions in AArch64 state, see Chapter C3 *A64 Instruction Set Overview*.

For more information on the floating-point and Advanced SIMD instructions in AArch32 state, see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*.

### A1.5.2 Floating-point standards, and terminology

The ARM includes support for all the required features of ANSI/IEEE Std 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic*, referred to as IEEE 754-2008. However, some terms in this manual are based on the 1985 version of this standard, referred to as IEEE 754-1985:

- ARM floating-point terminology generally uses the IEEE 754-1985 terms. This section summarizes how IEEE 754-2008 changes these terms.
- References to IEEE 754 that do not include the issue year apply to either issue of the standard.

Table A1-3 shows how the terminology in this manual differs from that used in IEEE 754-2008.

**Table A1-3 Floating-point terminology**

| This manual | IEEE 754-2008 |
|---|---|
| Normalized [a] | Normal |
| Denormal, or denormalized | Subnormal |
| Round towards Minus Infinity (RM) | roundTowardsNegative |
| Round towards Plus Infinity (RP) | roundTowardsPositive |
| Round towards Zero (RZ) | roundTowardZero |
| Round to Nearest (RN) | roundTiesToEven |
| Round to Nearest with Ties to Away | roundTiesToAway |
| Rounding mode | Rounding-direction attribute |

a. *Normalized number* is used in preference to *normal number*, because of the other specific uses of *normal* in this manual.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### A1.5.3 ARM standard floating-point input and output values

ARMv8 provides full IEEE 754 floating-point arithmetic support. In AArch32, floating-point operations performed using Advanced SIMD instructions are limited to *ARM standard floating-point operation*, regardless of the selected rounding mode in the FPSCR. Unlike AArch32, AArch64 SIMD floating point arithmetic is performed using the rounding mode selected by the FPCR.

ARM standard floating-point arithmetic supports the following input formats defined by the IEEE 754 floating-point standard:

- Zeros.
- Normalized numbers.
- Denormalized numbers are flushed to 0 before floating-point operations, see *Flush-to-zero*.
- NaNs.
- Infinities.

ARM standard floating-point arithmetic supports the Round to Nearest (roundTiesToEven) rounding mode defined by the IEEE 754 standard.

ARM standard floating-point arithmetic supports the following output result formats defined by the IEEE 754 standard:

- Zeros.
- Normalized numbers.
- Results that are less than the minimum normalized number are flushed to zero, see *Flush-to-zero*.
- NaNs produced in floating-point operations are always the default NaN, see *NaN handling and the Default NaN* on page A1-50.
- Infinities.

### A1.5.4 Flush-to-zero

The performance of floating-point processing can be reduced when doing calculations involving denormalized numbers and Underflow exceptions. In many algorithms, this performance can be recovered, without significantly affecting the accuracy of the final result, by replacing the denormalized operands and intermediate results with zeros. To permit this optimization, ARM floating-point implementations have a processing mode called *Flush-to-zero* mode. AArch32 Advanced SIMD floating-point instructions always use Flush-to-zero mode.

Behavior in Flush-to-zero mode differs from standard IEEE 754 arithmetic in the following ways:

- All inputs to floating-point operations that are double-precision denormalized numbers or single-precision denormalized numbers are treated as though they were zero. This causes an Input Denormal exception, but does not cause an Inexact exception. The Input Denormal exception occurs only in Flush-to-zero mode.

  In AArch32, the FPSCR contains a cumulative exception bit FPSCR.IDC and optional trap enable bit FPSCR.IDE corresponding to Input Denormal exception.

  In AArch64 the FPSR contains a cumulative exception bit FPSR.IDC and optional trap enable bit FPCR.IDE corresponding to the Input Denormal exception.

  The occurrence of all exceptions except Input Denormal is determined using the input values after flush-to-zero processing has occurred.

- The result of a floating-point operation is flushed to zero if the result of the operation before rounding satisfies the condition:

  $0 < \texttt{Abs(result)} <$ MinNorm, where:

  — MinNorm is $2^{-126}$ for single-precision

  — MinNorm is $2^{-1022}$ for double-precision.

  This causes the FPSR.UFC bit to be set to 1, and prevents any Inexact exception from occurring for the operation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Underflow exceptions occur only when a result is flushed to zero.

In all implementations Underflow exceptions that occur in Flush-to-zero mode are always treated as untrapped, even when the Underflow trap enable bit, FPCR.UFE, is set to 1.

- An Inexact exception does not occur if the result is flushed to zero, even though the final result of zero is not equivalent to the value that would be produced if the operation were performed with unbounded precision and exponent range.

When an input or a result is flushed to zero the value of the sign bit of the zero is preserved. That is, the sign bit of the zero matches the sign bit of the input or result that is being flushed to zero.

Flush-to-zero mode has no effect on half-precision numbers that are inputs to floating-point operations, or results from floating-point operations.

**Note**

Flush-to-zero mode is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Flush-to-zero mode must be used with care. Although it can improve performance on some algorithms, there are significant limitations on its use. These are application dependent:

- On many algorithms, it has no noticeable effect, because the algorithm does not normally use denormalized numbers.
- On other algorithms, it can cause exceptions to occur or seriously reduce the accuracy of the results of the algorithm.

### A1.5.5 NaN handling and the Default NaN

The IEEE 754 standard specifies that:

- An operation that produces an Invalid Operation floating-point exception generates a quiet NaN as its result if that exception is untrapped.
- An operation involving a quiet NaN operand, but not a signaling NaN operand, returns an input NaN as its result.

The floating-point processing behavior when Default NaN mode is disabled adheres to this, with the following additions:

- If an untrapped Invalid Operation floating-point exception is produced, the quiet NaN result is derived from:
  - The first signaling NaN operand, if the exception was produced because at least one of the operands is a signaling NaN.
  - Otherwise, the default NaN.
- If an untrapped Invalid Operation floating-point exception is not produced, but at least one of the operands is a quiet NaN, the result is derived from the first quiet NaN operand.

Depending on the operation, the exact value of a derived quiet NaN result may differ in both sign and number of fraction bits from its source. For a quiet NaN result derived from signaling NaN operand, the most-significant fraction bit is set to 1.

**Note**

- In these descriptions, *first operand* relates to the left-to-right ordering of the arguments to the pseudocode function that describes the operation.
- The IEEE 754 standard specifies that the sign bit of a NaN has no significance.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The floating-point and SIMD processing behavior when Default NaN mode is enabled is that the Default NaN is the result of all floating-point operations that either:

- Generate untrapped Invalid Operation floating-point exceptions.
- Have one or more quiet NaN inputs, but no signaling NaN inputs.

Table A1-4 shows the format of the default NaN for ARM floating-point operations.

Default NaN mode is selected for the floating-point processing by setting the FPCR.DN bit to 1.

Other aspects of the functionality of the Invalid Operation exception are not affected by Default NaN mode. These are that:

- If untrapped, it causes the FPSR.IOC bit be set to 1.
- If trapped, it causes a user trap handler to be invoked.

**Table A1-4 Default NaN encoding**

| | Half-precision, IEEE Format | Single-precision | Double-precision |
|---|---|---|---|
| Sign bit | 0 | 0 | 0 |
| Exponent | `0x1F` | `0xFF` | `0x7FF` |
| Fraction | Bit[9] == 1, bits[8:0] == 0 | bit[22] == 1, bits[21:0] == 0 | bit[51] == 1, bits[50:0] == 0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## A1.6 Cryptographic Extension

The presence of this Extension in an implementation is subject to export license controls. The Cryptographic Extension is an extension of the SIMD support and operates on the vector register file. It provides instructions for the acceleration of encryption and decryption to support the following:

- AES
- SHA1
- SHA2-256

The Cryptographic Extension also provides multiply instructions that operate on long polynomials, see *PMULL, PMULL2* on page C7-1179.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## A1.7 The ARM memory model

The ARM memory model supports:

- Generating an exception on an unaligned memory access.
- Restricting access by applications to specified areas of memory.
- Translating virtual addresses provided by executing instructions into physical addresses.
- Altering the interpretation of multi-byte data between big-endian and little-endian.
- Controlling the order of accesses to memory.
- Controlling caches and address translation structures.
- Synchronizing access to shared memory by multiple PEs.

Virtual address (VA) support depends on the Execution state, as follows:

**AArch64 state**

Supports 64-bit virtual addressing, with the Translation Control Register determining the supported VA range. Execution at EL1 and EL0 supports two independent VA ranges, each with its own translation controls.

**AArch32 state**

Supports 32-bit virtual addressing, with the Translation Control Register determining the supported VA range. For execution at EL1 and EL0, system software can split the VA range into two subranges, each with its own translation controls.

The supported physical address space is IMPLEMENTATION DEFINED, and can be discovered by system software.

Regardless of the Execution state, the *Virtual Memory System Architecture* (VMSA) can translate VAs to blocks or pages of memory anywhere within the supported physical address space.

For more information, see:

**For execution in AArch64 state**

- Chapter B2 *The AArch64 Application Level Memory Model*.
- Chapter D3 *The AArch64 System Level Memory Model*.
- Chapter D4 *The AArch64 Virtual Memory System Architecture*.

**For execution in AArch32 state**

- Chapter E2 *The AArch32 Application Level Memory Model*.
- Chapter G3 *The AArch32 System Level Memory Model*.
- Chapter G4 *The AArch32 Virtual Memory System Architecture*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Part B

## The AArch64 Application Level Architecture

# Chapter B1
# The AArch64 Application Level Programmers' Model

This chapter gives an application level view of the ARM programmers' model. It contains the following sections:


- *About the Application level programmers' model* on page B1-58.
- *Registers in AArch64 Execution state* on page B1-59.
- *Software control features and EL0* on page B1-64.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## B1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of the system information is needed to put the Application level programmers' model into context.

Depending on the implementation choices, the architecture supports multiple levels of execution privilege, indicated by different *Exception levels* that number upwards from EL0 to EL3. EL0 corresponds to the lowest privilege level and is often described as unprivileged. The Application level programmers' model is the programmers' model for software executing at EL0. For more information see *Exception levels* on page D1-1514.

System software determines the Exception level, and therefore the level of privilege, at which software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged at EL0. This:

- Permits the operating system to allocate system resources to an application in a unique or shared manner.
- Provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some system level understanding is necessary, and where relevant it gives a reference to the system level description.

Execution at any Exception level above EL0 is often referred to as privileged execution.

For more information on the system level view of the architecture refer to Chapter D1 *The AArch64 System Level Programmers' Model*.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## B1.2 Registers in AArch64 Execution state

This section describes the registers and process state visible at EL0 when executing in the AArch64 state. It includes the following:

- *Registers in AArch64 state*
- *Process state, PSTATE* on page B1-61
- *System registers* on page B1-62

### B1.2.1 Registers in AArch64 state

In the AArch64 application level view, an ARM processing element has:

**R0-R30** 31 general-purpose registers, R0 to R30. Each register can be accessed as:

- A 64-bit general-purpose register named X0 to X30.
- A 32-bit general-purpose register named W0 to W30.

See the register name mapping in Figure B1-1.

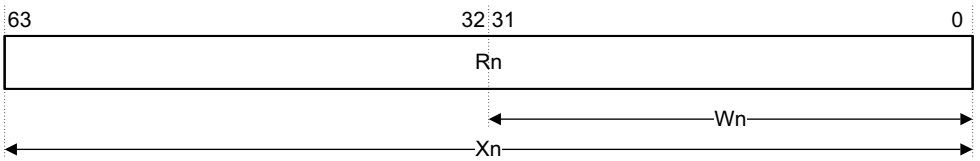



**Figure B1-1 General-purpose register naming**

The X30 general-purpose register is used as the procedure call link register.

**Note**

In instruction encodings, the value `0b11111` (31) is used to indicate the ZR (zero register). This indicates that the argument takes the value zero, but does not indicate that the ZR is implemented as a physical register.

**SP** A 64-bit dedicated Stack Pointer register. The least significant 32 bits of the stack-pointer can be accessed via the register name WSP.

The use of SP as an operand in an instruction, indicates the use of the current stack pointer.

**Note**

Stack pointer alignment to a 16-byte boundary is configurable at EL1. For more information see the *Procedure Call Standard for the ARM 64-bit Architecture*.

**PC** A 64-bit Program Counter holding the address of the current instruction.

Software cannot write directly to the PC. It can only be updated on a branch, exception entry or exception return.

**Note**

Attempting to execute an A64 instruction that is not word-aligned generates an Alignment fault, see *PC alignment checking* on page D1-1531.

**V0-V31** 32 SIMD&FP registers, V0 to V31. Each register can be accessed as:

- A 128-bit register named Q0 to Q31.
- A 64-bit register named D0 to D31.
- A 32-bit register named S0 to S31.
- A 16-bit register named H0 to H31.
- An 8-bit register named B0 to B31.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A 128-bit vector of elements.
- A 64-bit vector of elements.

Where the number of bits described by a register name does not occupy an entire SIMD&FP register, it refers to the least significant bits. See Figure B1-2.

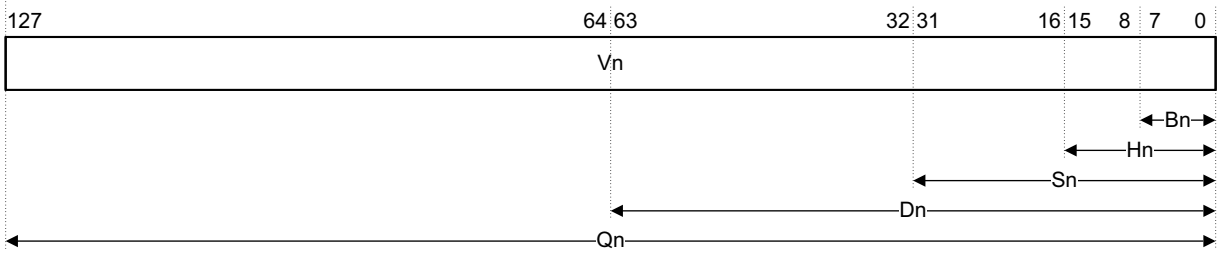



**Figure B1-2 SIMD and floating-point register naming**

For more information about data types and vector formats, see *Supported data types* on page A1-36.

**FPCR, FPSR** Two SIMD and floating-point control and status registers, FPCR and FPSR.

See *Registers for instruction processing and exception handling* on page D1-1523 for more information on the registers.

## Pseudocode description of registers in AArch64 state

In the pseudocode functions that access registers:

- The assignment form is used for register writes.
- The non-assignment for register reads.

The uses of the `X[]` function are:

- Reading or writing X0-X30, using n to index the required register.
- Reading the zero register ZR, accessed as `X[31]`.

**Note**

The pseudocode use of X[31] to represent the zero register does not indicate that hardware must implement this register.

The AArch64 `SP[]` function is used to read or write the current SP.

The AArch64 `PC[]` function is used to read the PC.

The AArch64 `V[]` function is used to read or write the Advanced SIMD and floating-point registers V0-V31, using a parameter `n` to index the required register.

The AArch64 `Vpart[]` function is used to read or write a part of one of V0-V31, using a parameter `n` to index the required register, and a parameter `part` to indicate the required part of the register, see the function description for more information.

The `SP[]`, `PC[]`, `V[]`, and `Vpart[]` functions are defined in Chapter J1 *ARMv8 Pseudocode*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### B1.2.2 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

The following PSTATE information is accessible at EL0:

**The condition flags**

Flag-setting instructions set these. They are:

**N** Negative condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:
- 1 if the result is negative.
- 0 if the result is positive or zero.

**Z** Zero condition flag. Set to:
- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

**C** Carry condition flag. Set to:
- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

**V** Overflow condition flag. Set to:
- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

Conditional instructions test the N, Z, C and V condition flags, combining them with the condition code for the instruction to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see *Condition flags and related instructions* on page C6-430.

**The exception masking bits**

**D** Debug exception mask bit. When EL0 is enabled to modify the mask bits, this bit is visible and can be modified. However, this bit is architecturally ignored at EL0.

**A** SError interrupt mask bit.

**I** IRQ interrupt mask bit.

**F** FIQ interrupt mask bit.

For each bit, the values are:

**0** Exception not masked.

**1** Exception masked.

Access at EL0 using AArch64 state depends on SCTLR_EL1.UMA. See *Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks* on page D1-1579.

See *Process state, PSTATE* on page D1-1529 for the system level view of PSTATE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing PSTATE fields at EL0

At EL0 using AArch64 state, PSTATE fields can be accessed using Special-purpose registers that can be directly read using the MRS instruction and directly written using the MSR (register) instructions. Table B1-1 shows the Special-purpose registers that access the PSTATE fields that hold AArch64 state when the PE is at EL0 using AArch64. All other PSTATE fields do not have direct read and write access at EL0.

**Table B1-1 Accessing PSTATE fields at EL0 using MRS and MSR (register)**

| Special-purpose register | PSTATE fields |
|---|---|
| NZCV | N, Z, C, V |
| DAIF | D, A, I, F |

Software can also use the MSR (immediate) instruction to directly write to PSTATE.{D, A, I, F}. Table B1-2 shows the MSR (immediate) operands that can directly write to PSTATE.{D, A, I, F} when the PE is at EL0 using AArch64 state.

**Table B1-2 Accessing PSTATE.{D, A, I, F} at EL0 using MSR (immediate)**

| Operand | PSTATE fields | Notes |
|---|---|---|
| DAIFSet | D, A, I, F | Directly sets any of the PSTATE.{D,A, I, F} bits to 1 |
| DAIFClr | D, A, I, F | Directly clears any of the PSTATE.{D, A, I, F} bits to 0 |

However, access to the PSTATE.{D, A, I, F} fields at EL0 using AArch64 state depends on SCTLR_EL1.UMA. *Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks* on page D1-1579.

Writes to the PSTATE fields have side-effects on various aspects of the PE operation. All of these side-effects, are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

## B1.2.3 System registers

System registers provide support for execution control, status and general system configuration. The majority of the System registers are not accessible at EL0.

However, some System registers can be configured to allow access from software executing at EL0. Any access from EL0 to a System register with the access right disabled causes the instruction to behave as an UNDEFINED instruction. The registers that can be accessed from EL0 are:

**Cache ID registers** The CTR_EL0 and DCZID_EL0 registers provide implementation parameters for EL0 cache management support.

**Debug registers** A debug communications channel is supported by the MDCCSR_EL0, DBGDTR_EL0, DBGDTRRX_EL0 and DBGDTRTX_EL0 registers.

**Performance Monitors registers**

See *Performance Monitors support* on page B1-63.

**Thread ID registers** The TPIDR_EL0 and TPIDRRO_EL0 registers are two thread ID registers with different access rights.

**Timer registers** In ARMv8 the following operations are performed:

- Read access to the system counter clock frequency using CNTFRQ_EL0.
- Physical and virtual timer count registers, CNTPCT_EL0 and CNTVCT_EL0.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

- Physical up-count comparison, down-count value and timer control registers, CNTP_CVAL_EL0, CNTP_TVAL_EL0, and CNTP_CTL_EL0.
- Virtual up-count comparison, down-count value and timer control registers, CNTV_CVAL_EL0, CNTV_TVAL_EL0, and CNTV_CTL_EL0.

### Performance Monitors support

The ARMv8 architecture defines optional Performance Monitors.

The basic form of the Performance Monitors is:

- A 64-bit cycle counter.
- Up to a maximum of 32 IMPLEMENTATION DEFINED event counters, where the number is identified by the PMCR_EL0.N field.
- System register access to the cycle counter and event registers, and related controls for:
  - Enabling and resetting counters.
  - Flagging overflows.
  - Generating interrupts on overflow.

  Software can enable the cycle counter independently of the event counters.

Software executing at EL1 or a higher Exception level, for example an operating system, can enable access to the counters from EL0. This allows an application to monitor its own performance with fine grain control without requiring operating system support. For example, an application might implement per-function performance monitoring.

For details on the features, configuration and control of the Performance Monitors, see Chapter D5 *The Performance Monitors Extension*.

#### *EL0 access to Performance Monitors*

To allow application code to make use of the Performance Monitors, software executing at a higher Exception level must set the following bits in the PMUSERENR_EL0 System register:

**EN** When set to 1, access to all Performance Monitors registers is allowed at EL0, except for writes to PMUSERENR_EL0, and reads/writes of PMINTENSET_EL1 and PMINTENCLR_EL1.

**ER** When set to 1, read access to event counters is allowed at EL0. This includes read/write access to PMSELR_EL0, so that the event counter to read through PMXEVCNTR_EL0 can be set.

**CR** When set to 1, read access to PMCCNTR_EL0 is allowed at EL0.

**SW** When set to 1, write access to PMSWINC_EL0 is allowed at EL0.

**Note**

Register PMUSERENR_EL0 is always read-only at EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## B1.3 Software control features and EL0

The following sections describe the EL0 view of the ARMv8 software control features:

- *Exception handling*
- *Wait for Interrupt and Wait for Event*
- *The YIELD instruction*
- *Application level cache management*
- *Debug events* on page B1-65

### B1.3.1 Exception handling

In the ARM architecture, an *exception* causes a change of program flow. Execution of an exception handler starts, at an Exception level higher than EL0, from a defined vector that relates to the exception taken.

Exceptions include:

- Interrupts.
- Memory system aborts.
- Undefined instructions.
- System calls.
- Secure monitor or Hypervisor traps.

Most details of exception handling are not visible to application level software, and are described in Chapter D1 *The AArch64 System Level Programmers' Model*.

The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a system call to an operating system.

### B1.3.2 Wait for Interrupt and Wait for Event

Issuing a `WFI` instruction indicates that no further execution is required until a WFI wake-up event occurs, see *Wait For Interrupt* on page D1-1614. This permits entry to a low-power state.

Issuing a `WFE` instruction indicates that no further execution is required until a WFE wake-up event occurs, see *Wait for Event mechanism and Send event* on page D1-1611. This permits entry to a low-power state.

### B1.3.3 The YIELD instruction

The `YIELD` instruction provides a hint that the task performed by a thread is of low importance so that it could yield, see *YIELD* on page C6-833. This mechanism can be used to improve overall performance in a *Symmetric Multi-Threading* (SMT) or *Symmetric Multi-Processing* (SMP) system.

Examples of when the `YIELD` instruction might be used include a thread that is sitting in a spin-lock, or where the arbitration priority of the snoop bit in an SMP system is modified. The `YIELD` instruction permits binary compatibility between SMT and SMP systems.

The `YIELD` instruction is a NOP (No Operation) hint instruction.

The `YIELD` instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use for future migration to a multiprocessor or multithreading system. Operating systems can use `YIELD` in places where a yield hint is wanted, knowing that it will be treated as a NOP if there is no implementation benefit.

### B1.3.4 Application level cache management

A small number of cache management instructions can be enabled at EL0 from higher levels of privilege using the SCTLR_EL1 System register. Any access from EL0 to an operation with the access right disabled causes the instruction to behave as an UNDEFINED instruction.

About the available operations, see *Application level cache instructions* on page B2-72.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### B1.3.5 Debug events

The debug logic is responsible for generating debug events. Most aspects of debug events are not visible to application level software, and are described in Chapter H1 *Introduction to External Debug*. Aspects that are visible to application level software include:

- The `BKPT` instruction, which causes a BKPT instruction debug event to occur.
- The `DBG` instruction, which provides a hint to the debug system.
- The `HLT` instruction, which causes entry to Debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter B2
# The AArch64 Application Level Memory Model

This chapter gives an application level view of the memory model. It contains the following sections:


- *Address space* on page B2-68.
- *Memory type overview* on page B2-69.
- *Caches and memory hierarchy* on page B2-70.
- *Alignment support* on page B2-75.
- *Endian support* on page B2-77.
- *Atomicity in the ARM architecture* on page B2-80.
- *Memory ordering* on page B2-83.
- *Memory types and attributes* on page B2-93.
- *Mismatched memory attributes* on page B2-104.
- *Synchronization and semaphores* on page B2-107.


**Note**

In this chapter, System register names usually link to the description of the register in Chapter D7 *AArch64 System Register Descriptions*, for example SCTLR_EL1.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## B2.1 Address space

Address calculations are performed using 64-bit registers. However, supervisory software can configure the top eight address bits for use as a tag, as described in *Address tagging in AArch64 state* on page D4-1742. If this is done, address bits[63:56]:

- Are not considered when determining whether the address is valid.
- Are never propagated to the program counter.

Supervisory software determines the valid address range. Attempting to access an address that is not valid generates an MMU fault.

*Simple sequential execution* of instructions might overflow the valid address range. For more information see *Instruction address space overflow* on page D3-1705.

Memory accesses use the `Mem[]` function. This function makes an access of the required type. If supervisory software configures the top eight address bits for use as a tag, the top eight address bits are ignored.

The `AccType{}` enumeration defines the different access types.

**Note**

- Chapter D3 *The AArch64 System Level Memory Model* and Chapter D4 *The AArch64 Virtual Memory System Architecture* include descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also include pseudocode descriptions of these operations.
- For information on the pseudocode that relates to memory accesses, see *Basic memory access* on page D3-1731, *Unaligned memory access* on page D3-1732, and *Aligned memory access* on page D3-1731.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## B2.2 Memory type overview

ARMv8 provides the following mutually-exclusive memory types:

**Normal** This is generally used for bulk memory operations, both read/write and read-only operations.

**Device** The ARM architecture forbids speculative reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive locations.

Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.

Device memory has additional attributes that have the following effects:

- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See *Gathering* on page B2-100.
- They preserve the access order and synchronization requirements, both for accesses to a single peripheral and where there is a synchronization requirement on the observability of one or more memory write and read accesses. See *Reordering* on page B2-101
- They indicate whether a write can be acknowledged other than at the end point. See *Early Write Acknowledgement* on page B2-102.

For more information on Normal memory and Device memory, see *Memory types and attributes* on page B2-93.

**Note**

Earlier versions of the ARM architecture defined a single Device memory type and a Strongly-ordered memory type. A *Note* in *Device memory* on page B2-98 describes how these memory types map onto the ARMv8 memory types.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## B2.3 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. ARMv8 defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- *Introduction to caches*.
- *Memory hierarchy*.
- *Application level cache instructions* on page B2-72
- *Implication of caches for the application programmer* on page B2-72.
- *Preloading caches* on page B2-74.

### B2.3.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a *tag*.
- The associated data.

Caches increase the average speed of a memory access. Caching takes account of two principles of locality:

**Spatial locality**

An access to one location is likely to be followed by accesses to adjacent locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

**Temporal locality**

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a *cache line*. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a *cache hit*, and other accesses are called *cache misses*.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. ARMv8 permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

### B2.3.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, an ARMv8 memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure B2-1 on page B2-71 shows an example of such a system in an ARMv8-A system that supports virtual addressing.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

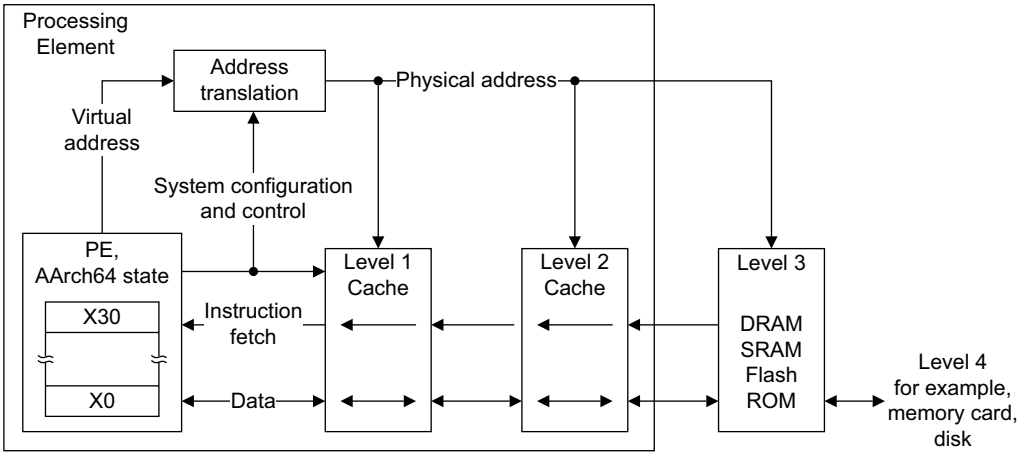



**Figure B2-1 Multiple levels of cache in a memory hierarchy**

**Note**

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the processing element, as shown in Figure B2-1.

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches that are located at the levels closest to the main memory. Memory coherency for cache topologies can be defined by two conceptual points:

**Point of Unification (PoU)**

The point at which the instruction cache, data cache, and translation table walks of a particular PE are guaranteed to see the same copy of a memory location. In many cases, the point of unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged. The point of unification might coincide with the point of coherency.

**Point of Coherency (PoC)**

The point at which all agents that can access memory are guaranteed to see the same copy of a memory location. In many cases this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherency between memory system agents.

**Note**

The presence of system caches can affect the definition of point of coherency as described in *System level caches* on page D3-1726.

See also *Overview of the cache maintenance instructions* on page D3-1712.

## The cacheability and shareability memory attributes

Cacheability and shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

**Cacheability** This attribute defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer cacheability locations.

**Shareability** This attribute defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer shareability domains.

For more information about cacheability and shareability see *Memory types and attributes* on page B2-93.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### B2.3.3 Application level cache instructions

In the ARM architecture, the application level is defined as *Exception level 0* (EL0). The architecture defines a set of cache maintenance instructions that software can use to manage cache coherency. Software executing at a higher Exception level can enable EL0 access to the following:

- The data cache maintenance instructions, `DC CVAU`, `DC CVAC`, and `DC CIVAC`. See *Data cache maintenance instructions (DC*)* on page D3-1717.
- The instruction cache maintenance instruction, `IC IVAU`. See *Instruction cache maintenance instructions (IC*)* on page D3-1717.
- The cache type register. See CTR_EL0.
- The data cache zero instruction, `DC ZVA`. See *Data cache zero instruction* on page D3-1724.

These instructions are UNDEFINED from EL0 unless software executing at a higher Exception level has enabled them. See *Cache maintenance instructions* on page D3-1716.

For all of these instructions, if the addresses do not have read access permission at EL0, executing these instructions at EL0 generates a Permission fault.

For more information about the system controls, see *Cache support* on page D3-1707.

### B2.3.4 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

For example:

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.
- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

#### Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
  - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  - Not enabling caches in the system.
- By using cache maintenance instructions to manage the coherency issues in software. See *Application level cache instructions*.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see *Non-shareable Normal memory* on page B2-95 and *Shareable, Inner Shareable, and Outer Shareable Normal memory* on page B2-94.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The performance of these hardware coherency mechanisms is highly implementation-specific. In some implementations the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

**Note**

Not all these mechanisms are directly available to software operating at EL0 and might involve interaction with software operating at a higher Exception level.

### Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last *Context synchronization operation* on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being refetched from memory. In the absence of an `ISB`, there is no limit on the number of times such an instruction might be executed without being refetched from memory.

The ARM architecture does not require the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory.

If software requires coherency between instruction execution and memory, it must manage this coherency using the `ISB` and `DSB` memory barriers and cache maintenance instructions. The following code sequence can be used to allow a PE to execute code that the same PE has written.

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
        STR Wt, [Xn]
        DC CVAU, Xn         ; Clean data cache by VA to point of unification (PoU)
        DSB ISH             ; Ensure visibility of the data cleaned from cache
        IC IVAU, Xn         ; Invalidate instruction cache by VA to PoU
        DSB ISH             ; Ensure completion of the invalidations
        ISB                 ; Synchronize the fetched instruction stream
```

**Note**

- For Non-cacheable or Write-Through accesses, the clean data cache by VA instruction is not required. However, the invalidate instruction cache instruction is required because the ARMv8-A AArch64 architecture allows Non-cacheable accesses to be held in an instruction cache. See *Non-cacheable accesses and instruction caches* on page D3-1712.
- This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The ARMv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See *Concurrent modification and execution of instructions* on page B2-82.
- The system software controls whether these cache maintenance instructions are available to the application level by setting SCTLR_EL1.UCI.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

If this sequence is not executed between writing data to a location and executing the instruction at that location, the lack of coherency between instruction caches and memory means that the instructions that are executed might be the old instruction or the updated instruction, and which is used can arbitrarily vary during execution. It must not be assumed by software, before the synchronization sequence is executed, that once the updated instruction has been seen, the old instruction will not be seen again.

### B2.3.5 Preloading caches

The ARM architecture provides memory system hints `PRFM`, `LDNP`, and `STNP` that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring the data or instruction locations into caches.

The Preload instructions are hints, and so implementations can treat them as `NOP`s without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous external abort, which is taken using an SError interrupt exception. For more information, see *Exception from a Data abort* on page D1-1549.

`PrefetchHint{}` defines the prefetch hint types.

The `Hint_Prefetch()` function signals to the memory system that memory accesses of the type `hint` to or from the specified `address` are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as preloading the specified address into one or more caches as indicated by the innermost cache level `target` and non-temporal hint `stream`.

For more information on `PRFM` and Load/Store instructions that provide hints to the memory system, see *Prefetch memory* on page C3-151 and *Load/Store SIMD and Floating-point Non-temporal pair* on page C3-149.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## B2.4 Alignment support

This section describes alignment support. It contains the following subsections:

- *Instruction alignment*.
- *Alignment of data accesses*.
- *Unaligned data access restrictions* on page B2-76.

### B2.4.1 Instruction alignment

A64 instructions must be word-aligned.

Attempting to fetch an instruction from a misaligned location results in a Misaligned PC fault. See *PC alignment checking* on page D1-1531.

### B2.4.2 Alignment of data accesses

An unaligned access to any type of Device memory causes an Alignment fault.

The alignment requirements for accesses to Normal memory are as follows:

- For all instructions that load or store a single or multiple registers, other than Load-Exclusive/Store-Exclusive and Load-Acquire/Store-Release, if the address that is accessed is not aligned to the size of the data element being accessed, then one of the following occurs:
  - An Alignment fault is generated.
  - An unaligned access is performed.

  When the value of SCTLR_ELx.A at the current Exception level is 1, alignment checking is enabled, and unaligned accesses generate Alignment faults.

  **Note**

  - The SCTLR_EL1.A bit applies to software running at EL0 and at EL1, although it can only be accessed from EL1 and higher.
  - Alignment checks are based on the size of the accessed elements, not the overall access size. This affects SIMD element and structure loads and stores, and also Load/Store pair instructions.
  - These alignment checking rules mean the ARMv8 architecture introduces requirements for 64-bit and 128-bit alignment checking.

- For all Load-Exclusive/Store-Exclusive and Load-Acquire/Store-Release memory accesses that access a single element or a pair of elements, an Alignment fault is generated if the address being accessed is not aligned to the size of the data structure being accessed.

A failed alignment check results in an Alignment fault, which is taken as a Data Abort exception, that is taken as follows:

- For an access from Non-secure EL0 or EL1, if the Alignment fault is generated only because the translation tables identify the address being accessed as Device memory then:
  - If the first stage of address translation marks the address as Device memory then the exception is taken to EL1.
  - If only the second stage of address translation marks the address as Device memory then the exception is taken to EL2.
- Otherwise, the exception is taken to the lowest Exception level that can handle the exception, consistent with the basic requirement that the Exception level never decreases on taking an exception. Therefore:
  - Alignment faults taken from EL0 or EL1 are taken to EL1 unless redirected by HCR_EL2.TGE
  - Alignment faults taken from EL2 are taken to EL2.
  - Alignment faults taken from EL3 are taken to EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### B2.4.3 Unaligned data access restrictions

The following points apply to unaligned data accesses in ARMv8:

- Accesses are not guaranteed to be single-copy atomic except at the byte access level, see *Atomicity in the ARM architecture* on page B2-80.
- Unaligned accesses typically take a number of additional cycles to complete compared to a naturally-aligned access.
- An operation that performs an unaligned access can abort on any memory access that it makes, and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the boundary.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## B2.5 Endian support

*General description of endianness in the ARM architecture* describes the relationship between endianness and memory addressing in the ARM architecture.

The following subsections then describe the endianness schemes supported by the architecture:

- *Instruction endianness* on page B2-78.
- *Data endianness* on page B2-78.

### B2.5.1 General description of endianness in the ARM architecture

This section only describes memory addressing and the effects of endianness for data elements up to quadwords of 128 bits. However, this description can be extended to apply to larger data elements.

For an address A, Figure B2-2 shows, for big-endian and little-endian memory systems, the relationship between:

- The quadword at address A.
- The doubleword at address A and A+8.
- The words at addresses A, A+4, A+8, and A+12.
- The halfwords at addresses A, A+2, A+4, A+6, A+8, A+10, A+12, and A+14.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, A+7, A+8, A+9, A+10, A+11, A+12, A+13, A+14, and A+15.

The terms in Figure B2-2 have the following definitions:

**B_A** Byte at address A.
**HW_A** Halfword at address A.
**MSByte** Most-significant byte.
**LSByte** Least-significant byte.

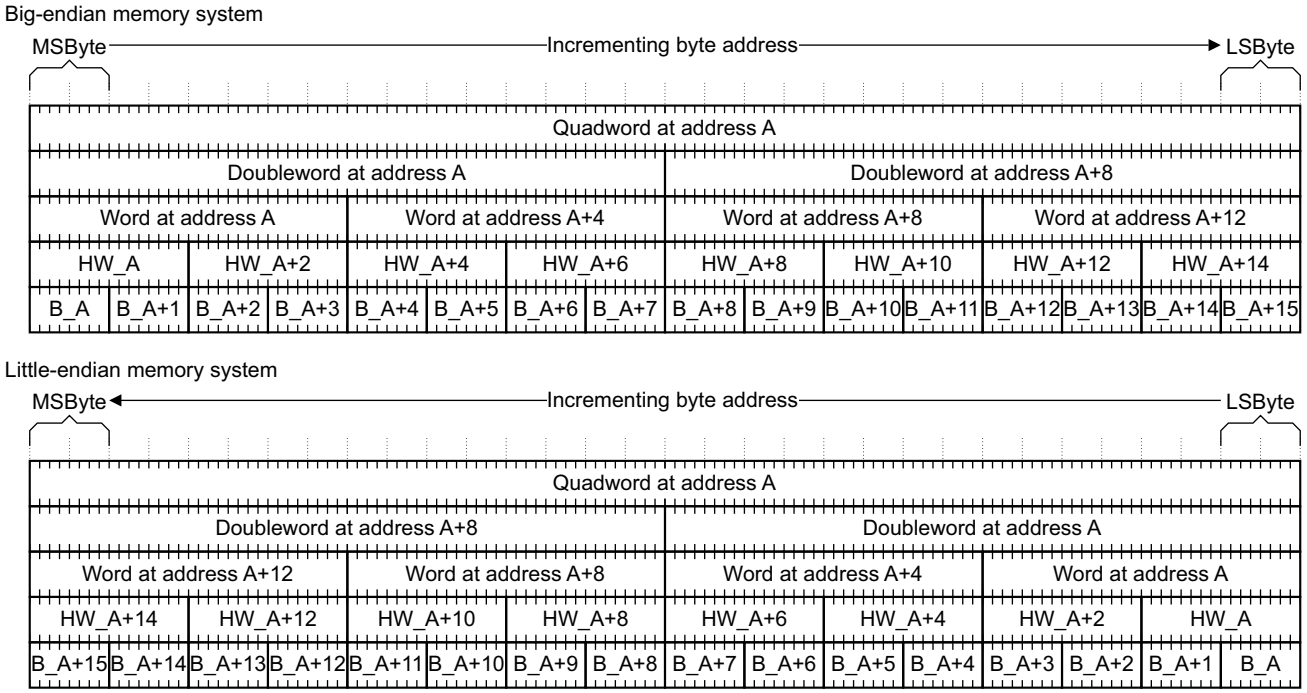



**Figure B2-2 Endianness relationships**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The big-endian and little-endian mapping schemes determine the order in which the bytes of a quadword, doubleword, word or halfword are interpreted. For example, a load of a word from address `0x1000` always results in an access to the bytes at memory locations `0x1000`, `0x1001`, `0x1002`, and `0x1003`. The endianness mapping scheme determines the significance of these four bytes.

### B2.5.2 Instruction endianness

In ARMv8-A, A64 instructions have a fixed length of 32 bits and are always little-endian.

### B2.5.3 Data endianness

SCTLR_EL1.E0E, configurable at EL1 or higher, determines the data endianness for execution at EL0.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for SIMD element and data structure loads and stores. For more information see *Endianness in SIMD operations* on page B2-79.

**Note**

This means the ARMv8 architecture introduces a requirement for 128-bit endian conversions.

#### Instructions to reverse bytes in a general-purpose register or a SIMD and floating-point register

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Table B2-1 shows the instructions that provide this functionality:

**Table B2-1 Byte reversal instructions**

| Function | Instructions | Notes |
|---|---|---|
| Reverse bytes in 32-bit word or words[a] | REV32 | For use with general-purpose registers |
| Reverse bytes in whole register | REV | For use with general-purpose registers |
| Reverse bytes in 16-bit halfwords | REV16 | For use with general-purpose registers |
| Reverse elements in doublewords, vector | REV64 | For use with SIMD and floating-point registers |
| Reverse elements in words, vector | REV32 | For use with SIMD and floating-point registers |
| Reverse elements in halfwords, vector | REV16 | For use with SIMD and floating-point registers |

a. Can operate on multiple words.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Endianness in SIMD operations

SIMD element Load/Store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used to load and store data correctly in both big-endian and little-endian systems.

For example:

```
LD1 {V0.4H}, [X1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure B2-3. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

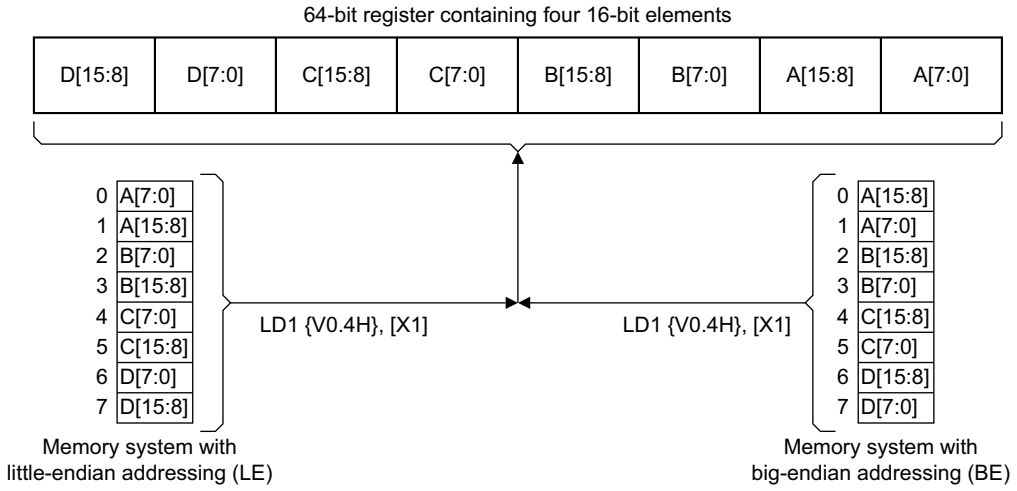



**Figure B2-3 SIMD byte order example**

The `BigEndian()` pseudocode function determines the current endianness of the data.

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in Chapter J1 *ARMv8 Pseudocode*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## B2.6 Atomicity in the ARM architecture

*Atomicity* is a feature of memory accesses, described as *atomic* accesses. The ARM architecture description refers to two types of atomicity, *single-copy atomicity* and *multicopy atomicity*. In the ARMv8 architecture, the atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information see:

- *Requirements for single-copy atomicity*.
- *Properties of single-copy atomic accesses* on page B2-81.
- *Multi-copy atomicity* on page B2-81.
- *Requirements for multi-copy atomicity* on page B2-81.

For more information about the memory types, see *Memory type overview* on page B2-69.

### B2.6.1 Requirements for single-copy atomicity

For explicit memory accesses generated from an Exception level the following rules apply:

- All reads generated by load instructions that load a single general-purpose register and that are aligned to the size of the read in that instruction are single-copy atomic.
- All writes generated by store instructions that store a single general-purpose register and that are aligned to the size of the write in that instruction are single-copy atomic.
- Reads of general-purpose registers generated by Load Pair instructions that are aligned to the size of the load to each register are treated as two single-copy atomic reads, one for each register being loaded.
- Writes of general-purpose registers generated by Store pair instructions that are aligned to the size of the store of each register are treated as two single-copy atomic writes, one for each register being stored.
- Load-Exclusive Pair instructions of two 32-bit quantities and Store-Exclusive Pair instructions of 32-bit quantities are single-copy atomic.
- When the Store-Exclusive of a Load-Exclusive/Store-Exclusive pair instruction using two 64-bit quantities succeeds, it causes a single-copy atomic update of the entire memory location being updated.

  **Note**

  To atomically load two 64-bit quantities, perform a Load-Exclusive pair/Store-Exclusive pair sequence of reading and writing the same value for which the Store-Exclusive pair succeeds, and use the read values from the Load-Exclusive pair.

- Where translation table walks generate a read of a translation table entry, this read is single-copy atomic.
- For the atomicity of instruction fetches, see *Concurrent modification and execution of instructions* on page B2-82.
- Reads to floating-point and SIMD registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being loaded are treated as single-copy atomic reads.
- Writes from floating-point and SIMD registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being stored are treated as single-copy atomic writes.
- Element or Structure Reads to floating-point and SIMD registers of 64-bit or smaller elements, where each element is aligned to the size of the element being loaded, have each element treated as a single-copy atomic read.
- Element or Structure Writes from floating-point and SIMD registers of 64-bit or smaller elements, where each element is aligned to the size of the element being stored, have each element treated as a single-copy atomic store.
- Reads to floating-point and SIMD registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit reads.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Writes from floating-point and SIMD registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit writes.

All other memory accesses are regarded as streams of accesses to bytes, and no atomicity between accesses to different bytes is ensured by the architecture.

All accesses to any byte are single-copy atomic.

**Note**

In AArch64 state, no memory accesses from a DC ZVA have single-copy atomicity of any quantity greater than individual bytes.

If, according to these rules, an instruction is executed as a sequence of accesses, exceptions, including interrupts, can be taken during that sequence, regardless of the memory type being accessed. If any of these exceptions are returned from using their preferred return address, the instruction that generated the sequence of accesses is re-executed, and so any access performed before the exception was taken is repeated. See also *Taking an interrupt or other exception during a multiple-register load or store* on page D1-1574.

**Note**

The exception behavior for these multiple access instructions means they are not suitable for use for writes to memory for the purpose of software synchronization.

### B2.6.2 Properties of single-copy atomic accesses

A read or write operation that is *single-copy atomic* has the following properties:

1. For a single-copy atomic store, if the store overlaps another single-copy atomic store, then all of the writes from one of the stores are inserted into the Coherence order of each overlapping byte before any of the writes of the other store are inserted into the Coherence orders of the overlapping bytes.
2. If a single-copy atomic load overlaps a single-copy atomic store and for any of the overlapping bytes the load returns the data written by the write inserted into the Coherence order of that byte by the single-copy atomic store then the load must return data from a point in the Coherence order no earlier than the writes inserted into the Coherence order by the single-copy atomic store of all of the overlapping bytes.

### B2.6.3 Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

**Note**

Writes that are not coherent are not multi-copy atomic.

### B2.6.4 Requirements for multi-copy atomicity

In a multiprocessing system, coherent writes to a memory location are multi-copy atomic if the read of a location returns the value of a write only when all observers have observed that write.

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory with the non-Gathering attribute, writes that are single-copy atomic are also multi-copy atomic.

For Device memory with the Gathering attribute, writes are not required to be multi-copy atomic.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### B2.6.5 Concurrent modification and execution of instructions

The ARMv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where the instruction before modification and the instruction after modification is a `B`, `BL`, `NOP`, `BRK`, `SVC`, `HVC`, or `SMC` instruction.

For the `B`, `BL`, `NOP`, `BRK`, `SVC`, `HVC`, and `SMC` instructions the architecture guarantees that, after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

If one thread of execution changes a conditional branch instruction, such as `B` or `BL`, to another conditional instruction and the change affects both the condition field and the branch target, execution of the changed instruction by another thread of execution before the change is synchronized can lead to either:

- The old condition being associated with the new target address.
- The new condition being associated with the old target address.

These possibilities apply regardless of whether the condition, either before or after the change to the branch instruction, is the *always* condition.

For all other instructions, to avoid UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. No PE must be executing an instruction when another PE is modifying that instruction.

2. To ensure that the modified instructions are observable, the PE that modified the instructions must issue the following sequence of instructions and operations:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
        STR Wt, [Xn]
        DC CVAU, Xn         ; Clean data cache by VA to point of unification (PoU)
        DSB ISH             ; Ensure visibility of the data cleaned from cache
        IC IVAU, Xn         ; Invalidate instruction cache by VA to PoU
        DSB ISH             ; Ensure completion of the invalidations
```

**Note**

The `DC CVAU` operation is not required if the area of memory is either Non-cacheable or Write-through Cacheable.

3. In a multiprocessor system, the `IC IVAU` is broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. However, when the modified instructions are observable, each PE that is executing the modified instructions must issue the following instruction to ensure execution of the modified instructions:

```
ISB                             ; Synchronize fetched instruction stream
```

For more information about the required synchronization operation, see *Synchronization and coherency issues between data and instruction accesses* on page B2-73.

**Note**

For information about memory accesses caused by instruction fetches, see *Ordering requirements* on page B2-84.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## B2.7 Memory ordering

This section describes observation ordering. It contains the following subsections:

- *Observability and completion*.
- *Ordering requirements* on page B2-84.
- *Memory barriers* on page B2-86.
- *Summary of the memory ordering rules* on page B2-90.

For information on endpoint ordering of memory accesses, see *Reordering* on page B2-101.

In the ARMv8 memory model, the shareability memory attribute indicates whether hardware must ensure memory coherency.

The ARMv8 memory system architecture defines additional attributes and associated behaviors, defined in the system level section of this manual. See:

- Chapter D3 *The AArch64 System Level Memory Model*.
- Chapter D4 *The AArch64 Virtual Memory System Architecture*.

See also *Mismatched memory attributes* on page B2-104.

### B2.7.1 Observability and completion

An *observer* is a master in the system that is capable of observing memory accesses. For a PE, the following mechanisms must be treated as independent observers:

- The mechanism that performs reads or writes to memory.
- A mechanism that causes an instruction cache to be filled from memory or that fetches instructions to be executed directly from memory. These are treated as reads.
- A mechanism that performs translation table walks. These are treated as reads.

The set of observers that can observe a memory access is defined by the system.

In the definitions in this subsection, *subsequent* means whichever of the following is appropriate to the context:

- After the point in time where the location is observed by that observer.
- After the point in time where the location is globally observed.

For all memory:

- A write to a location in memory is said to be *observed* by an observer when:
  - A subsequent read of the location by the same observer returns the value written by the observed write, or written by a write to that location by any observer that is sequenced in the *Coherence order* of the location after the observed write.
  - A subsequent write of the location by the same observer is sequenced in the *Coherence order* of the location after the observed write.
- A write to a location in memory is said to be *globally observed* for a shareability domain or set of observers when:
  - A subsequent read of the location by any observer in that shareability domain returns the value written by the globally observed write, or written by a write to that location by any observer that is sequenced in the *Coherence order* of the location after the globally observed write.
  - A subsequent write of the location by any observer in that shareability domain is sequenced in the *Coherence order* of the location after the globally observed write.
- A read of a location in memory is said to be observed by an observer when a subsequent write to the location by the same observer has no effect on the value returned by the read.
- A read of a location in memory is said to be globally observed for a shareability domain when a subsequent write to the location by any observer in that shareability domain has no effect on the value returned by the read.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Additionally, for Device-nGnRnE memory:

- A read or write of a memory-mapped location in a peripheral that exhibits side-effects is said to be observed, and globally observed, only when the read or write:
  - Meets the general conditions listed.
  - Can begin to affect the state of the memory-mapped peripheral.
  - Can trigger all associated side-effects, whether they affect other peripheral devices, PEs, or memory.

  **Note**

  This definition is consistent with the memory access having reached the peripheral.

For all memory, the completion rules are defined as:

- A read or write is complete for a shareability domain when all of the following are true:
  - The read or write is globally observed for that shareability domain.
  - Any translation table walks associated with the read or write are complete for that shareability domain.
- A translation table walk is complete for a shareability domain when the memory accesses associated with the translation table walk are globally observed for that shareability domain, and the TLB is updated.
- A cache or TLB maintenance instruction is complete for a shareability domain when the effects of the instruction are globally observed for that shareability domain, and any translation table walks that arise from the instruction are complete for that shareability domain.

  The completion of any cache or TLB maintenance instruction includes its completion on all PEs that are affected by both the instruction and the `DSB` operation that is required to guarantee visibility of the maintenance instruction.

#### Completion of side-effects of accesses to Device memory

The completion of a memory access to Device memory other than Device-nGnRnE is not guaranteed to be sufficient to determine that the side-effects of the memory access are visible to all observers. The mechanism that ensures the visibility of side-effects of a memory access is IMPLEMENTATION DEFINED.

### B2.7.2 Ordering requirements

ARMv8 defines restrictions for the permitted ordering of memory accesses. These restrictions depend on the memory type of the addresses that are accessed, see *Memory types and attributes* on page B2-93.

**Note**

See *Summary of the memory ordering rules* on page B2-90 for the definition of *address dependency*.

The only stores by an observer that can be observed by another observer are those stores that have been *Architecturally executed*. Speculative writes by an observer cannot be observed by another observer. For the purposes of this requirement, speculative writes are all of:

- Writes generated by store instructions that appear in the *Execution stream* after a branch that is not architecturally resolved.
- Writes generated by store instructions that appear in the *Execution stream* after an instruction where a synchronous exception condition has not been architecturally resolved.
- Writes generated by conditional store instructions for which the conditions for the instruction have not been architecturally resolved.
- Writes generated by store instructions for which the data being written comes from a register that has not been architecturally committed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following additional restrictions apply to the order in which accesses to memory are observed:

- Reads and writes to different addresses can be observed in any order provided the following constraints are met:
  - If an address dependency exists between two reads or between a read and a write, then those memory accesses are observed in program order by all observers within the common shareability domain of the memory addresses being accessed.

    The ARMv8 architecture relaxes this rule for execution where the second read is generated by a Load Non-Temporal Pair instruction. See *Load/Store Non-temporal Pair* on page C3-145 and *Load/Store SIMD and Floating-point Non-temporal pair* on page C3-149.
  - Ordering can be achieved by using a `DMB` or `DSB` barrier. For more information on `DMB` and `DSB` instructions, see *Memory barriers* on page B2-86.
- Reads and writes to the same address are coherent within the shareability domain of the memory address being accessed.
- Two reads to the same address by the same observer are observed in program order by all observers within the shareability domain of the memory address being accessed.
- Writes are not required to be multi-copy atomic. This means that in the absence of barriers, the observation of a store by one observer does not imply the observation of the store by another observer.
- Instructions that access multiple elements have no defined ordering requirements for the memory accesses relative to each other.

For Device memory with the non-Reordering attribute, the order of memory accesses arriving at a *single peripheral* is the same as occurs in a *Simple sequential execution* on page Glossary-5775 of the program. This means the accesses arrive in program order. This ordering applies for all accesses using any of the memory types with the non-Reordering attribute, which means Device-nGnRE accesses are ordered with respect to Device-nGnRnE accesses to the same peripheral. If the memory accesses are not to a peripheral then there are no ordering restrictions from the non-Reordering attribute. For the purposes of this definition, a *single peripheral* is a region of memory of an IMPLEMENTATION DEFINED size that is defined by the peripheral.

Memory accesses caused by instruction fetches are not required to be observed in program order, unless they are separated by an `ISB` or other context synchronization event.

### Address dependencies and order

In the ARMv8 architecture, a register data dependency between the value returned by a load instruction and the address used by a subsequent memory transaction creates order between that load instruction and the subsequent memory transaction.

A register data dependency exists between a first data value and a second data value when either:

- The register, excluding the zero register (XZR or WZR), used to hold the first data value is used in the calculation of the second data value, and the calculation between the first data value and the second data value does not consist of either:
  - A conditional branch whose condition is determined by the first data value.
  - A conditional selection, move, or computation whose condition is determined by the first data value, where the input data values for the selection, move, or computation do not have a data dependency on the first data value.
- There is a register data dependency between the first data value and a third data value, and between the third data value and the second data value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

A register data dependency can exist even if the value of the first data value is discarded as part of the calculation, as might be the case if it is ANDed with `0x0` or if arithmetic using the first data value cancels out its contribution.

For example, each of the following code sequences creates order between the memory transactions:

**Sequence 1**
```
LDR X1, [X2]
AND X1, X1, XZR
LDR X4, [X3, X1]
```

**Sequence 2**
```
LDR X1, [X2]
ADD X3, X3, X1
SUB X3, X3, X1
STR X4, [X3]
```

#### *Address dependencies of Load Non-temporal Pair instructions*

Where an address dependency exists between two reads, and the second read was generated by a Load Non-temporal Pair instruction, then in the absence of any other barrier mechanism to achieve order, those memory accesses can be observed in any order by other observers within the shareability domain of the memory addresses being accessed.

This affects the following instruction:

- *LDNP* on page C6-552.

### B2.7.3 Memory barriers

The ARM architecture is a weakly ordered memory architecture that supports out of order completion. *Memory barrier* is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring Load/Store instructions. The memory barriers defined by the ARMv8 architecture provide a range of functionality, including:

- Ordering of Load/Store instructions.
- Completion of Load/Store instructions.
- Context synchronization.

The following subsections describe the ARMv8 memory barrier instructions:

- *Instruction Synchronization Barrier (ISB)* on page B2-87
- *Data Memory Barrier (DMB)* on page B2-87.
- *Data Synchronization Barrier (DSB)* on page B2-88.
- *Shareability and access limitations on the data barrier operations* on page B2-89.
- *Load-Acquire, Store-Release* on page B2-89.

**Note**

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are only available when software execution is at EL1 or higher.

The `DMB` and `DSB` memory barriers affect reads and writes to the memory system generated by Load/Store instructions and data or unified cache maintenance instructions being executed by the PE. Instruction fetches or accesses caused by a hardware translation table access are not explicit accesses.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Instruction Synchronization Barrier (ISB)

An ISB instruction flushes the pipeline in the PE, so that all instructions that come after the ISB instruction in program order are fetched from the cache or memory only after the ISB instruction has completed. Using an ISB ensures that the effects of context-changing operations executed before the ISB are visible to the instructions fetched after the ISB instruction. Examples of context-changing operations that require the insertion of an ISB instruction to ensure the effects of the operation are visible to instructions fetched after the ISB instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the ISB instruction only take effect after the ISB has been executed.

The pseudocode function for the operation of an ISB is `InstructionSynchronizationBarrier()`.

See also *Memory barriers* on page D3-1738.

### Data Memory Barrier (DMB)

The DMB instruction is a data memory barrier. The PE that executes the DMB instruction is referred to as the executing PE, PEe. The DMB instruction takes an `<option>` argument that specifies the shareability domains and access types to which the instruction applies, see *Shareability and access limitations on the data barrier operations* on page B2-89.

If the required shareability is *Full system* then the operation applies to all observers within the system.

A DMB creates two groups of memory accesses, Group A and Group B:

**Group A** Contains:

- All explicit memory accesses of the required access types from observers in the same required shareability domain as PEe that are observed by PEe before the DMB instruction. These accesses include any accesses of the required access types performed by PEe.
- All loads of required access types from an observer PEx in the same required shareability domain as PEe that have been observed by any given different observer, PEy, in the same required shareability domain as PEe before PEy has performed a memory access that is a member of Group A.

**Group B** Contains:

- All explicit memory accesses of the required access types by PEe that occur in program order after the DMB instruction.
- All explicit memory accesses of the required access types by any given observer PEx in the same required shareability domain as PEe that can only occur after a load by PEx has returned the result of a store that is a member of Group B.

Any observer with the same required shareability domain as PEe observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined by the shareability and cacheability of the memory addresses accessed by the group members.

If members of Group A and members of Group B access the same memory-mapped peripheral of arbitrary system-defined size, then members of Group A that are accessing Device or Normal Non-cacheable memory arrive at that peripheral before members of Group B that are accessing Device or Normal Non-cacheable memory. Where the members of Group A and Group B that must be ordered are from the same PE, a `DMB NSH` is sufficient for this guarantee.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- A memory access might be in neither Group A nor Group B. The DMB does not affect the order of observation of such a memory access.
- The second part of the definition of Group A is recursive. Ultimately, membership of Group A derives from the observation by PEy of a load before PEy performs an access that is a member of Group A as a result of the first part of the definition of Group A.
- The second part of the definition of Group B is recursive. Ultimately, membership of Group B derives from the observation by any observer of an access by PEe that is a member of Group B as a result of the first part of the definition of Group B.

DMB only affects memory accesses and the operation of data cache and unified cache maintenance instructions, see *Cache maintenance instructions* on page D3-1716. It has no effect on the ordering of any other instructions executing on the PE. A DMB instruction intended to ensure the completion of cache maintenance instructions must have an access type of both loads and stores.

The pseudocode function for the operation of a DMB is `DataMemoryBarrier()`.

See also *Memory barriers* on page D3-1738.

### Data Synchronization Barrier (DSB)

The DSB instruction is a memory barrier, that synchronizes the execution stream with memory accesses.

The DSB instruction takes an <option> argument that specifies the shareability domains and access types to which the instruction applies, see *Shareability and access limitations on the data barrier operations* on page B2-89.

If the required shareability is *Full system* then the operation applies to all observers within the system.

A DSB behaves as a DMB with the same arguments, and also has the additional properties defined in this section. The PE that executes the DSB instruction is referred to as the executing PE, PEe

Execution of a DSB:

- At EL2 ensures that any memory accesses caused by speculative translation table walks from the Non-secure EL1&0 translation regime have been observed.
- At EL3 ensures that any memory accesses caused by speculative translation table walks from any of the following translation regimes have been observed:
  - The EL2 translation regime.
  - The Secure EL1&0 translation regime.
  - The Non-secure EL1&0 translation regime.

For more information, see *Use of out-of-context translation regimes* on page D4-1754.

A DSB completes when all of the following apply:

- All explicit memory accesses that are observed by PEe before the DSB is executed and are of the required access types, and are from observers in the same required shareability domain as PEe, are complete for the set of observers in the required shareability domain.
- If the required access types of the DSB is reads and writes, then all cache maintenance instructions and all TLB maintenance instructions issued by PEe before the DSB are complete for the required shareability domain.

In addition, no instruction that appears in program order after the DSB instruction can execute until the DSB completes.

The pseudocode function for the operation of a DSB is `DataSynchronizationBarrier()`.

See also *Memory barriers* on page D3-1738.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Shareability and access limitations on the data barrier operations

The DMB and DSB instructions take an argument that specifies:

- The shareability domain over which the instruction must operate. This is one of:
  - Full system.
  - Outer Shareable.
  - Inner Shareable.
  - Non-shareable.
- The accesses for which the instruction operates. This is one of:
  - Read and write accesses in Group A and Group B.
  - Write accesses only in Group A and Group B.
  - Read access only in Group A and read and write accesses in Group B.

**Note**

This form of a DMB or DSB instruction can be described as a Load-Load/Store barrier.

Table B2-2 shows how these options are encoded in the <option> field of the instruction:

**Table B2-2 Encoding of the DMB and DSB <option> parameter**

| Accesses | | Shareability domain | | | |
|---|---|---|---|---|---|
| **Group A** | **Group B** | **Full system** | **Outer Shareable** | **Inner Shareable** | **Non-shareable** |
| Reads and writes | Reads and writes | SY | OSH | ISH | NSH |
| Writes | Writes | ST | OSHST | ISHST | NSHST |
| Reads | Reads and writes | LD | OSHLD | ISHLD | NSHLD |

See the instruction descriptions for more information:

- *DMB* on page C6-515.
- *DSB* on page C6-518.

**Note**

ISB also supports an optional limitation argument that can only contain one value that corresponds to full system operation, see *ISB* on page C6-534.

### Load-Acquire, Store-Release

ARMv8 provides a set of instructions with Acquire semantics for loads, and Release semantics for stores. See *Load-Acquire/Store-Release* on page C3-147.

For all memory types, these instructions have the following ordering requirements:

- A Store-Release followed by a Load-Acquire is observed in program order by any observers that are in both:
  - The shareability domain of the address accessed by the Store-Release.
  - The shareability domain of the address accessed by the Load-Acquire.
- For a Load-Acquire, observers in the shareability domain of the address accessed by the Load-Acquire observe accesses in the following order:
  1. The read caused by the Load-Acquire.
  2. Reads and writes caused by loads and stores that appear in program order after the Load-Acquire for which the shareability of the address accessed by the load or store requires that the observer observes the access.

  There are no additional ordering requirements on loads or stores that appear before the Load-Acquire.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For a Store-Release, observers in the shareability domain of the address accessed by the Store-Release observe accesses in the following order:
  1. All of the following for which the shareability of the address accessed requires that the observer observes the access:
     - Reads and writes caused by loads and stores that appear in program order before the Store-Release.
     - Writes that were observed by the PE executing the Store-Release before it executed the Store-Release.
  2. The write caused by the Store-Release.

  There are no other ordering requirements on loads or stores that appear in program order after the Store-Release.
- A Store-Release instruction is multi-copy atomic when observed with a Load-Acquire instruction.

In addition, for accesses to a memory-mapped peripheral of an arbitrary system-defined size that are defined as any type of Device memory accesses, these instructions have the following requirements:

- A Load-Acquire to an address in the memory-mapped peripheral will ensure that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed after the Load-Acquire will arrive at the memory-mapped peripheral after the memory access of the Load-Acquire.
- A Store-Release to an address in the memory-mapped peripheral will ensure that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed before the Store-Release will arrive at the memory-mapped peripheral before the memory access of the Store-Release.
- If a Load-Acquire to a memory address in the memory-mapped peripheral has observed the value stored to that address by a Store-Release, then any memory access to the memory-mapped peripheral that is architecturally required to be ordered before the memory access of the Store-Release will arrive at the memory-mapped peripheral before any memory access to the same peripheral that is architecturally required to be ordered after the memory access of the Load-Acquire.

Load-Acquire and Store-Release, other than Load-Acquire Exclusive Pair and Store-Release-Exclusive Pair, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.

Load-Acquire Exclusive Pair and Store-Release Exclusive Pair access two data elements. The address supplied to the instructions must be aligned to twice the size of the element being loaded, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction only has the release semantics if the store is successful.

**Note**

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
  - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.
- The Load-Acquire/Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction.

### B2.7.4 Summary of the memory ordering rules

The following is a concise list of the situations that are required, by the ARM architecture specification, to cause externally-visible order of memory. This ordering means that if memory transaction A has externally visible order ahead of memory transaction B, then all observers within the shareability domains of A and B will observe A before B. See *Terms used in the summary of the memory ordering rules* for definitions of the terms used.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

This list applies to both AArch32 state and AArch64 state, and is consistent with the requirements of ARMv7.

1. `DMB` and `DSB` barrier instructions, and load acquire/store release instructions, create externally-visible order as defined by those instructions.
2. A True or False Address dependency from a Load to a Load or from a Load to a Store creates externally-visible order.
3. A True Control dependency from a Load to an `ISB` instruction creates externally-visible order between the load and any memory accesses after the `ISB` instruction.
4. A True Register data dependency from a Load to a Store creates externally-visible order.
5. A True Control dependency from a Load to a Store creates externally-visible order.
6. Memory is coherent within the shareability domain of a memory address, which means there is a total order of all writes to that address that all observers within that shareability domain will agree on.

   **Note**

   A consequence of this is that reads to the same address by the same PE are observed in order.

7. A Dependency from a Store to a Load through memory between different PEs creates externally-visible order but stores are not multi-copy atomic except where explicitly defined to be by the definition of the store.

   **Note**

   A consequence of the lack of multi-copy atomicity is that a Store to Load dependency through memory on the same PE does not create externally-visible order.

No other effects are required to create externally visible order in the ARM architecture.

### Terms used in the summary of the memory ordering rules

The summary uses the following terms:

**Register data dependency**

This is defined in *Address dependencies and order* on page B2-85.

**False Register data dependency**

A False Register data dependency is a Register data dependency where no register in the system holds a variable for which a change of the first data value causes a change of the second data value.

**True Register data dependency**

A True Register data dependency is a Register data dependency that is not a false Register data dependency.

**True Address dependency**

A True Address dependency between a load and a subsequent memory transaction exists where there is a True Register data dependency between the data value returned from the load and the address used by the subsequent memory transaction.

**False Address dependency**

A False Address dependency between a load and a subsequent memory transaction exists where there is a False Register data dependency between the data value returned from the load and the address used by the subsequent memory transaction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**True Control dependency**

A True Control dependency between a load and a subsequent instruction exists:

- Where there is a True Register data dependency between the data value returned from the load and data value used in the evaluation of a conditional branch and the subsequent instruction is only executed as a result of one of the possible outcomes of that conditional branch.
- Where there is a True Register data dependency between the data value returned from the load and the data value used in the evaluation of a subsequent instruction that is a conditional selection, move, or computation for which both:
  - The condition is determined by the returned data value.
  - No input data value for the selection, move, or computation has a register data dependency on the returned data value.

**Dependency from a Store to a Load through memory**

A Dependency from a Store to a Load through memory exists where the Store and Load are to the same physical address, and value returned by the Load is the value that was written by the Store, and could not be the value that was previously held at that memory address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## B2.8 Memory types and attributes

In ARMv8 the ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- *Normal memory*.
- *Device memory* on page B2-98.
- *Memory access restrictions* on page B2-103.

### B2.8.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform speculative data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- A write to a memory location with the Normal attribute completes in finite time. This means that it is globally observed for the shareability domain of the memory location in finite time. For a Non-cacheable location, the location is observed by all observers in finite time.
- A completed write to a memory location with the Normal attribute is globally observed for the shareability domain of the memory location in finite time without the need for explicit cache maintenance instructions or barriers. For a Non-cacheable location, the completed write is globally observed for all observers in finite time without the need for explicit cache maintenance instructions or barriers.
- Writes to a memory location with the Normal memory attribute that are Non-cacheable must reach the endpoint for that location in the memory system in finite time.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register Load/Store instructions. See *Multi-register loads and stores that access Normal memory* on page B2-98.

**Note**

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side-effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.
- An instruction that generates a sequence of accesses as described in *Atomicity in the ARM architecture* on page B2-80 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

The following sections describe the other attributes for Normal memory:

- *Shareable Normal memory* on page B2-94.
- *Non-shareable Normal memory* on page B2-95.
- *Write-Through Cacheable, Write-Back Cacheable and Non-cacheable Normal memory* on page B2-95.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See also:

- *Multi-register loads and stores that access Normal memory* on page B2-98.
- *Atomicity in the ARM architecture* on page B2-80.
- *Memory barriers* on page B2-86. For accesses to Normal memory, a `DMB` instruction is required to ensure the required ordering.
- *Concurrent modification and execution of instructions* on page B2-82.

## Shareable Normal memory

A Normal memory location has a Shareability attribute that is one of:

- Inner Shareable, meaning it applies across the Inner Shareable shareability domain.
- Outer Shareable, meaning it applies across both the Inner Shareable and the Outer Shareable shareability domains.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, that hardware must enforce. They do not affect the coherency requirements of instruction fetches, see *Synchronization and coherency issues between data and instruction accesses* on page B2-73.

**Note**

- System designers can use the shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different processing elements.
- This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

### *Shareable, Inner Shareable, and Outer Shareable Normal memory*

The ARM architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is only a member of a single Inner Shareability domain.
- Each observer is only a member of a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.
- The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.

The details of the use of the shareability attributes are system-specific. Example B2-1 shows how they might be used.

**Example B2-1 Use of shareability attributes**

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable shareability domain.

For shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

### Non-shareable Normal memory

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

### Write-Through Cacheable, Write-Back Cacheable and Non-cacheable Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Also, for cacheable Normal memory regions:

- A region might be assigned a cache allocation hint.
- It is IMPLEMENTATION DEFINED whether the Write-Through Cacheable and Write-Back Cacheable attributes can have an additional attribute of Transient or Non-transient, see *Transient cacheability attribute* on page B2-97.

A memory location can be marked as having different cacheability attributes, for example when using aliases in a virtual to physical address mapping:

- If the attributes differ only in the cache allocation hint this does not affect the behavior of accesses to that location.
- For other cases see *Mismatched memory attributes* on page B2-104.

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

- A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.
- For accesses to Normal memory that is Non-cacheable, a `DMB` instruction ensures that all members of Group A reach a single peripheral or block of memory, of IMPLEMENTATION DEFINED size, before any member of Group B, where:
  - The definition of the operation of a `DMB` instruction defines Group A and Group B, see *Data Memory Barrier (DMB)* on page B2-87.
  - The IMPLEMENTATION DEFINED size of the single peripheral or block of memory is defined by the peripheral or block of memory.

  This applies for all types of `DMB` instruction.

**Note**

Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the ARM architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.
- No cache that is controlled by the Inner cacheability attributes can lie outside a cache that is controlled by the Outer cacheability attributes.
- An implementation might not have any outer cache.

Example B2-2 on page B2-97, Example B2-3 on page B2-97, and Example B2-4 on page B2-97 describe the possible ways of implementing a system with three levels of cache, *level 1* (L1) to *level 3* (L3).


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- L1 cache is the level closest to the PE, see *Memory hierarchy* on page B2-70.
- When managing coherency, system designs must consider both the inner and outer cacheability attributes, as well as the shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

**Example B2-2 Implementation with two inner and one outer cache levels**

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 and L2 cache.
- The Outer cacheability attribute applied to L3 cache.

**Example B2-3 Implementation with three inner and no outer cache levels**

Implement the three levels of cache in the system, L1 to L3, with the Inner cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer cacheability attribute.

**Example B2-4 Implementation with one inner and two outer cache levels**

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 cache.
- The Outer cacheability attribute applied to L2 and L3 cache.

### *Transient cacheability attribute*

In ARMv8, it is IMPLEMENTATION DEFINED whether a Transient attribute is supported for cacheable Normal memory regions. If an implementation supports this attribute, the set of possible cacheability attributes for a Normal memory region becomes:

- Write-Through Cacheable, Non-transient.
- Write-Back Cacheable, Non-transient.
- Write-Through Cacheable, Transient.
- Write-Back Cacheable, Transient.
- Non-cacheable.

The cacheability attribute can be defined independently for the inner and outer levels of caching.

The transient attribute indicates that the benefit of caching is for a relatively short period. Therefore it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.

**Note**

The architecture does not specify what is meant by a *relatively short period*.

The description of the AArch64 MAIR_EL1, MAIR_EL2, and MAIR_EL3 registers, and the AArch32 MAIR0, MAIR1, HMAIR0, and HMAIR1 registers, includes the assignment of the Transient attribute in an implementation that supports this option.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more SIMD and floating-point register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

## B2.8.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for ARMv8 Device memory are:

**Gathering** Identified as G or nG, see *Gathering* on page B2-100.

**Reordering** Identified as R or nR, see *Reordering* on page B2-101.

**Early Write Acknowledgement hint**

Identified as E or nE, see *Early Write Acknowledgement* on page B2-102.

The ARMv8 Device memory types are:

**Device-nGnRnE** Device non-Gathering, non-Reordering, No Early write acknowledgement.

Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.

**Device-nGnRE** Device non-Gathering, non-Reordering, Early Write Acknowledgement.

Equivalent to the Device memory type in earlier versions of the architecture.

**Device-nGRE** Device non-Gathering, Reordering, Early Write Acknowledgement.

ARMv8 adds this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.

**Device-GRE** Device Gathering, Reordering, Early Write Acknowledgement.

ARMv8 adds this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that speculative accesses to Device-GRE memory is forbidden.

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.

**Note**

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
- The architecture does not require an implementation to distinguish between each of these memory types and ARM recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.
- Earlier versions of the ARM architecture defined the following memory types:
  - Strongly-ordered memory. This is the equivalent of the Device-nGnRnE memory type.
  - Device memory. This is the equivalent of the Device-nGnRE memory type.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.

  The following exceptions to this apply:

  — Reads generated by the SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  — For Device memory with the Gathering attribute, reads generated by the `LDNP` instructions are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a 128-byte window, aligned to 128-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  — Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction. See *Properties of single-copy atomic accesses* on page B2-81.

  **Note**

  — An instruction that generates a sequence of accesses as described in *Atomicity in the ARM architecture* on page B2-80 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program only defines a single access. For this reason, ARM strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.

  — Write speculation that is visible to other observers is prohibited for all memory types.

- A write to a memory location with any Device memory attribute completes in finite time. This means that it is globally observed for all observers in the system in finite time.

- If a location with any Device memory attribute changes without an explicit write by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.

- A completed write to a memory location with any Device memory attribute is globally observed for all observers in finite time without the need for explicit maintenance.

- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.

- A memory location with any Device memory attribute cannot be allocated into a cache.

- Writes to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Typically, the endpoint is a peripheral or some physical memory.

- For accesses to any Device memory type, a `DMB` instruction ensures that all members of Group A reach a single peripheral or block of memory, of IMPLEMENTATION DEFINED size, before any member of Group B, where:

  — The definition of the operation of a `DMB` instruction defines Group A and Group B, see *Data Memory Barrier (DMB)* on page B2-87.

  — The IMPLEMENTATION DEFINED size of the single peripheral or block of memory is defined by the peripheral or block of memory.

  This applies for all types of `DMB` instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- All accesses to memory with any Device memory attribute must be aligned. Any unaligned access generates an Alignment fault at the first stage of translation that defined the location as being Device.

  **Note**

  In the Non-secure EL1&0 translation regime in systems where HCR_EL2.TGE==1 and HCR_EL2.DC==0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes ESR_EL2.ISS[24] to be 0.

- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as Execute-never for all Exception levels.

  **Note**

  This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as Execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as Execute-never for all Exception levels is a programming error.

See also *Memory access restrictions* on page B2-103.

The memory types for Translation table walks cannot be defined as any Device memory type within the TCR_ELx. For the Non-secure EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR_EL2.PTW bit is 1, a stage 2 permission fault is generated if a first stage translation table walk is made to any Device memory type.

**Note**

In general, making a translation table walk to any Device memory type is the result of a programming error.

For instruction fetches, if branches cause the program counter to point to an area of memory with the Device attribute which is not marked as Execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

### Gathering

In the Device memory attribute:

**G** Indicates that the location has the Gathering attribute.

**nG** Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

**Note**

This also applies to writebacks from the cache, whether caused by a *Natural eviction* or as a result of a cache maintenance instruction.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All accesses occur at their programmed size, except that there is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register Load/Store instructions. See *Multi-register loads and stores that access Device memory* on page B2-103.

Gathering between memory accesses separated by a memory barrier that affects those memory accesses is not permitted. This applies if one memory access is in Group A and one memory access is in Group B. That is, gathering is not permitted between a memory access in Group A and a memory access in Group B if the two accesses are separated by a barrier that affects at least one of the accesses.

Gathering between two memory accesses generated by a Load-Acquire/Store-Release is not permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

**Note**

- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  - The accesses are not separated by a `DMB` or `DSB` barrier that affects both of the accesses, for example if one access is in Group A and the other is in Group B.
  - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
  - The accesses are not generated by a Store-Release instruction.
- The ARM architecture only defines programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the Non-gathering attribute.

An implementation is not permitted to perform an access with the Non-gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

### Reordering

In the Device memory attribute:

**R** Indicates that the location has the Reordering attribute. Accesses to the location can be reordered within the same rules that apply to accesses to Normal Non-cacheable memory. All memory types with the Reordering attribute have the same ordering rules as accesses to Normal Non-cacheable memory, see *Memory ordering* on page B2-83.

**nR** Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

**Note**

Some interconnect fabrics, such as PCIe, perform very limited re-ordering, which is not important for the software usage. It is outside the scope of the ARM architecture to prohibit the use of a Non-reordering memory type with these interconnects.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the DMB instruction.
- The ARM architecture only defines programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.

An implementation:

- Is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.
- Is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses with the non-Reordering attribute and accesses with the Reordering attribute.
- Accesses with the non-Reordering attribute and accesses to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

The non-Reordering attribute has no effect on the ordering of cache maintenance instructions, even if the memory location specified in the instruction has the non-Reordering attribute.

### Early Write Acknowledgement

In the Device memory attribute:

**E** Indicates that the location has the Early Write Acknowledgement attribute.

**nE** Indicates that the location has the No Early Write Acknowledgement attribute.

Early Write Acknowledgement is a hint to the platform memory system. Assigning the No Early Write Acknowledgement attribute to a Device memory location recommends that only the endpoint of the write access returns a write acknowledgement of the access, and that no earlier point in the memory system returns a write acknowledge. This means that a DSB barrier, executed by the PE that performed the write to the No Early Write Acknowledgement location, completes only after the write has reached its endpoint in the memory system. Typically, this endpoint is a peripheral or physical memory.

When the Early Write Acknowledgement attribute is assigned to a Device memory location, there is no such recommendation for the handling of accesses to that location.

**Note**

- The Early Write Acknowledgement hint has no effect on the ordering rules. The purpose of signaling no Early Write Acknowledgement is to signal to the interconnect that the peripheral requires the ability to signal the acknowledgement. The No Write Acknowledgement signal also provides an additional semantic that can be interpreted by the driver that is accessing the peripheral.
- This attribute is treated as a hint, as the exact nature of the interconnects accessed by a PE is outside the scope of the ARM architecture definition, and not all interconnects provide a mechanism to ensure that a write has reached the physical endpoint of the memory system.
- ARM recommends that writes with the No Early Write Acknowledgement hint are used for PCIe configuration writes. However, the mechanisms by which PCIe configuration writes are identified are IMPLEMENTATION DEFINED.
- ARM strongly recommends that the Early Write Acknowledgement hint is not ignored by a PE, but is made available for use by the system.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Because the No Early Write Acknowledgement attribute is a hint:

- An implementation is permitted to perform an access with the Early Write Acknowledgement attribute in a manner consistent with the requirements specified by the No Early Write Acknowledgement attribute.
- An implementation is permitted to perform an access with the No Early Write Acknowledgement attribute in a manner consistent with the relaxations allowed by the Early Write Acknowledgement attribute.

### Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture. This applies even to accesses to any type of Device memory.

For all instructions that load or store one or more floating-point and SIMD register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions, even for access to any type of Device memory.

## B2.8.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For accesses to any two bytes, *p* and *q*, that are generated by the same instruction:
  - The bytes *p* and *q* must have the same memory type and shareability attributes, otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an `LD1`, `ST1`, or an unaligned load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.
  - Except for possible differences in the cache allocation hints, ARM deprecates having different cacheability attributes for bytes *p* and *q*.

  For the permitted CONSTRAINED UNPREDICTABLE behavior, see *Crossing a page boundary with different memory types or Shareability attributes* on page K1-5504.
- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross an address boundary that corresponds to the smallest implemented translation granule then behavior is CONSTRAINED UNPREDICTABLE, and *Crossing a peripheral boundary with a Device access* on page K1-5505 describes the permitted behaviors. For this reason, it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

  **Note**

  — The boundary referred to is between two Device memory regions that are both of the size of the smallest implemented translation granule and aligned to the size of the smallest implemented translation granule.

  — This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

  — ARM expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## B2.9 Mismatched memory attributes

Memory attributes are controlled by privileged software. For more information, see Chapter D4 *The AArch64 Virtual Memory System Architecture*.

Physical memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all of the following attributes of that location:

- Memory type, Device or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.

Collectively these are referred to as memory attributes.

**Note**

The terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory location is accessed with mismatched attributes the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory location might be lost. This means:
  - A read of the memory location by one agent might not return the value most recently written to that memory location by the same agent.
  - Multiple writes to the memory location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory location do not use consistent memory attributes, the exclusive monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of speculative read accesses.
- Prohibition on Gathering.
- Prohibition on Re-ordering.

For the following situations, when a physical memory location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. If the only memory type mismatch associated with a memory location across all users of the memory location is between different types of Device memory, then all accesses might take the properties of the weakest Device memory type.
2. Any agent that reads that memory location using the same common definition of the shareability and cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all of the following conditions are met:
   - All aliases to the memory location with write permission both use a common definition of the shareability and cacheability attributes for the memory location, and either:
     - Have the inner cacheability attribute the same as the outer cacheability attribute.
     - In the Non-secure EL1&0 translation regime, have HCR_EL2.MIOCNCE set to 0.
   - All aliases to a memory location use a definition of the shareability attributes that encompasses all the agents with permission to access the location.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

3. The possible software-visible effects caused by mismatched attributes for a memory location are defined more precisely if all of the mismatched attributes define the memory location as one of:
   - Any Device memory type.
   - Normal Inner Non-cacheable, Outer Non-cacheable memory.

   In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:
   - Possible loss of properties derived from the memory type when multiple agents attempt to access the memory location.
   - Possible reordering of memory transactions to the same memory location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting `DMB` barrier instructions between accesses to the same memory location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory location all assign the same shareability attribute to the location, any loss of uniprocessor semantics, ordering, or coherency within a shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable locations between agents in different shareability domains. This means:
   - Before writing to a location not using the Write-Back attribute, software must invalidate, or clean, a location from the caches if any agent might have written to the location with the Write-Back attribute. This avoids the possibility of overwriting the location with stale data.
   - After writing to a location with the Write-Back attribute, software must clean the location from the caches, to make the write visible to external memory.
   - Before reading the location with a cacheable attribute, software must invalidate the location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
   - Executing a `DMB` barrier instruction, with scope that applies to the common shareability of the accesses, between any accesses to the same memory location that use different attributes.

   In all cases:
   - Location refers to any byte within the current coherency granule.
   - A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
   - In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see *Ordering and completion of data and instruction cache instructions* on page D3-1722.

   **Note**

   With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

2. If the mismatched attributes for a location mean that multiple cacheable accesses to the location might be made with different shareability attributes, then uniprocessor semantics, ordering, and coherency are guaranteed only if:
   - Each PE that accesses the location with a cacheable attribute performs a clean and invalidate of the location before and after accessing that location.
   - A `DMB` barrier with scope that covers the full shareability of the accesses is placed between any accesses to the same memory location that use different attributes.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a location, and the accesses from the different agents have different memory attributes associated with the location, the exclusive monitor state becomes UNKNOWN.

ARM strongly recommends that software does not use mismatched attributes for aliases of the same location. An implementation might not optimize the performance of a system that uses mismatched aliases.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## B2.10 Synchronization and semaphores

ARMv8 provides non-blocking synchronization of shared memory, using *synchronization primitives*. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal memory and to any type of Device memory.

**Note**

Use of the ARMv8 synchronization primitives scales for multiprocessing system designs.

Table B2-3 shows the synchronization primitives and the associated `CLREX` instruction.

**Table B2-3 Synchronization primitives and associated instruction**

| Function | A64 instruction |
|---|---|
| Load-Exclusive | |
| Byte | `LDXRB`, `LDAXRB` |
| Halfword | `LDXRH`, `LDAXRH` |
| Register[a] | `LDXR`, `LDAXR` |
| Pair[a] | `LDXP`, `LDAXP` |
| Store-Exclusive | |
| Byte | `STXRB`, `STLXRB` |
| Halfword | `STXRH`, `STLXRH` |
| Register[a] | `STXR`, `STLXR` |
| Pair[a] | `STXP`, `STLXP` |
| Clear-Exclusive | `CLREX` |

a. The instruction operates on a doubleword if accessing an X register, or on a word if accessing a W register.

The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address *x* is:

- The Load-Exclusive instruction reads a value from memory address *x*.
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address *x* only if no other observer, process, or thread has performed a more recent store to address *x*. The Store-Exclusive instruction returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see *Marking and the size of the marked memory block* on page B2-113. A Store-Exclusive instruction to any address in the marked block clears the marking.

**Note**

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- *Exclusive access instructions and Non-shareable memory locations* on page B2-108.
- *Exclusive access instructions and Shareable memory locations* on page B2-110.
- *Marking and the size of the marked memory block* on page B2-113.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Context switch support* on page B2-113.
- *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page B2-114.
- *Use of WFE and SEV instructions by spin-locks* on page B2-116.

### B2.10.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the shareability attribute is Non-shareable, the exclusive access instructions rely on a *local monitor* that marks any address from which the PE executes a Load-Exclusive instruction. Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

**If the local monitor is in the Exclusive Access state**

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - If the store took place the status value is 0.
  - Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

**If the local monitor is in the Open Access state**

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

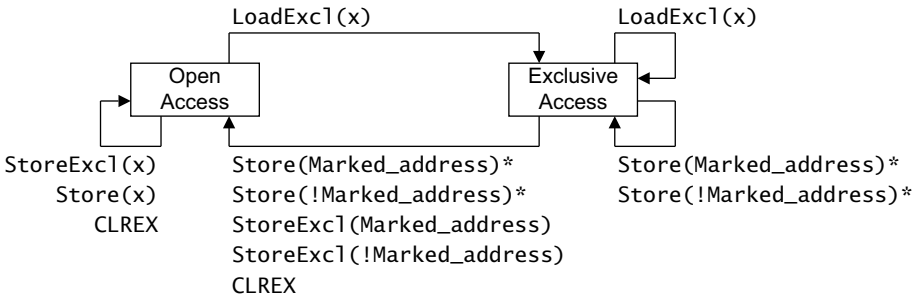
When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a physical address that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a physical address that is marked as Exclusive Access by its local monitor it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked physical address causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the physical address to be marked.

Figure B2-4 on page B2-109 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

**Figure B2-4 Local monitor state machine diagram**

For more information about marking see *Marking and the size of the marked memory block* on page B2-113.

**Note**

For the local monitor state machine, as shown in Figure B2-4:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any physical address, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

## Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure B2-4.

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure B2-4.
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure B2-4 must not indefinitely delay forward progress of execution.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### B2.10.2 Exclusive access instructions and Shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.

For shareable memory locations, exclusive access instructions rely on:

- A *local monitor* for each PE in the system, that marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in *Exclusive access instructions and Non-shareable memory locations* on page B2-108, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:
  - Updating memory.
  - Returning a status value of 0.

  The local monitor can ignore accesses from other PEs in the system.
- A *global monitor* that marks a physical address as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:
  - Can hold at least one marked block.
  - Maintains a state machine for each marked block it can hold.

  **Note**

  For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is UNPREDICTABLE, see *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page B2-114.

**Note**

The global monitor can either reside within the PE, or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

**Note**

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all ARM PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport's Bakery algorithm to achieve mutual exclusion.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global exclusive monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.

The set of memory types that support atomic instructions must include all of the memory types for which a global monitor is implemented.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive or a Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an external abort.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Fault Status code of ESR_ELx.DFSC = `110101`.

  If the IMPLEMENTATION DEFINED MMU fault is generated for the Non-secure EL1&0 translation regime then:

  — If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.

  — Otherwise, the fault is a second stage fault and the exception is taken to EL2.

  The priority of this fault is IMPLEMENTATION DEFINED.
- The instruction is treated as a `NOP`.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN. In this case, if the store exclusive instruction is a store exclusive pair of 64-bit quantities, then the two quantities being stored might not be stored atomically.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by ARM PEs is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

### Operation of the global monitor

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the physical address of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other physical address that has been marked by the requesting PE.

**Note**

The global monitor only supports a single outstanding exclusive access to shareable memory per PE.

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the physical address accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:

  — A status value of 0 is returned to a register to acknowledge the successful store.

  — The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— If the address accessed is marked for exclusive access in the global monitor state machine for any other PE then that state machine transitions to Open Access state.

- If no address is marked as exclusive access for the requesting PE, the store does not succeed:
  — A status value of 1 is returned to a register to indicate that the store failed.
  — The global monitor is not affected and remains in Open Access state for the requesting PE.
- If a different physical address is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  — If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.
  — If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means it responds to:

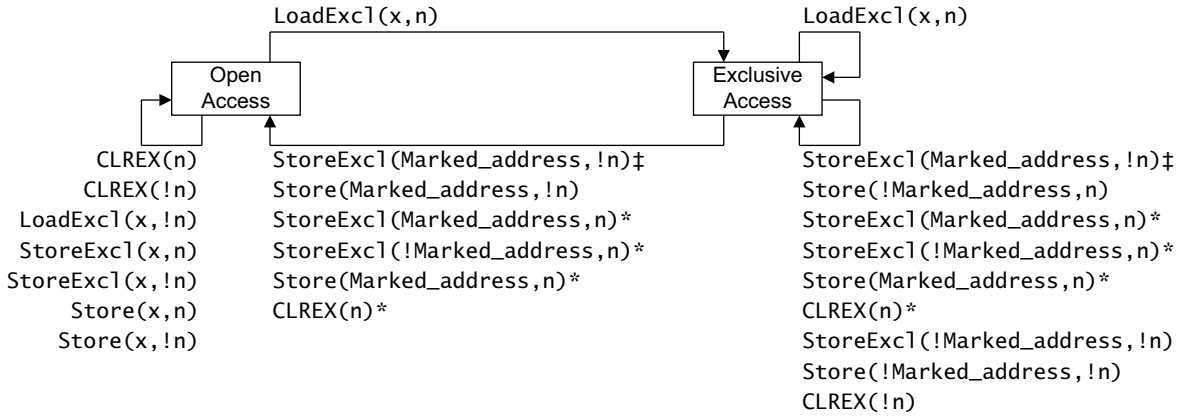
- Accesses generated by PE(n).
- Accesses generated by the other observers in the shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

### *Clear global monitor event*

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see *Mechanisms for entering a low-power state* on page D1-1611.

Figure B2-5 shows the state machine for PE(n) in a global monitor.




‡StoreExcl(Marked_address,!n) clears the monitor only if the StoreExcl updates memory
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any other store instruction.

Any LoadExcl operation updates the marked address to the most significant bits of the address x used for the operation.

**Figure B2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system**

For more information about marking see *Marking and the size of the marked memory block* on page B2-113.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

For the global monitor state machine, as shown in Figure B2-5 on page B2-112:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.
- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure B2-5 on page B2-112 only shows how the operations by (!n) cause state transitions of the state machine for PE(n).
- A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
- When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a `CLREX` instruction causes the global monitor to transition from Exclusive Access to Open Access state.
- It is IMPLEMENTATION DEFINED:
  - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

### B2.10.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size $2^a$ bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the *Exclusives reservation granule*. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4 - 512 words.

**Note**

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where $a$ is 4.
- 512 words in an implementation where $a$ is 11.

For example, in an implementation where $a$ is 4, a successful `LDXRB` of address `0x341B4` defines a marked block using bits[47:4] of the address. This means that the four words of memory from `0x341B0` to `0x341BF` are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR_EL0. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

### B2.10.4 Context switch support

An exception return clears the local monitor. As a result, performing a `CLREX` instruction as part of a context switch is not required in most situations.

**Note**

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### B2.10.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a `LDXP`/`STXP` pair or a `LDXR`/`STXR` pair. To support different implementations of these functions, software must follow the notes and restrictions given here.

The following notes describe the use of a `LoadExcl`/`StoreExcl` pair, to indicate the use of any of the Load-Exclusive/Store-Exclusive pairs shown in Table B2-3 on page B2-107:

- The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target virtual address of a `StoreExcl` is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution, behavior can be UNPREDICTABLE. As a result, a `LoadExcl`/`StoreExcl` pair can only be relied upon to eventually succeed if the `LoadExcl` and the `StoreExcl` are executed with the same virtual address.

- If two `StoreExcl` instructions are executed without an intervening `LoadExcl` instruction the second `StoreExcl` instruction returns a status value of 1. This means that:
  — ARM recommends that, in a given thread of execution, every `StoreExcl` instruction has a preceding `LoadExcl` instruction associated with it.

  It is not necessary for every `LoadExcl` instruction to have a subsequent `StoreExcl` instruction.

- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a Store-Exclusive instruction is the same as the transaction size of the preceding Load-Exclusive instruction executed in that thread. If the transaction size of a Store-Exclusive instruction is different from the preceding Load-Exclusive instruction in the same thread of execution, behavior can be UNPREDICTABLE. As a result, software can rely on an `LoadExcl`/`StoreExcl` pair to eventually succeed only if they have the same size.

- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the Store-Exclusive instruction accesses the same number of registers as the preceding Load-Exclusive instruction executed in that thread. If the Store-Exclusive instruction accesses a different number of registers than the preceding Load-Exclusive instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. As a result, software can rely on an `LoadExcl`/`StoreExcl` pair to eventually succeed only if they access the same number of registers. For more information see *UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers* on page B2-115.

- An implementation might clear an exclusive monitor between the `LoadExcl` instruction and the `StoreExcl`, instruction without any application-related cause. For example, this might happen because of cache evictions. Software must, in any single thread of execution, avoid having any explicit memory accesses, System register updates, or cache maintenance instructions between the `LoadExcl` instruction and the associated `StoreExcl` instruction.

- Implementations can benefit from keeping the `LoadExcl` and `StoreExcl` operations close together in a single thread of execution. This minimizes the likelihood of the exclusive monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, ARM strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.

- The architecture sets an upper limit of 2048 bytes on the exclusive reservation granule that can be marked as exclusive. For performance reasons, ARM recommends that objects that are accessed by exclusive accesses are separated by the size of the exclusive reservations granule. This is a performance guideline rather than a functional requirement.

- After taking a Data Abort exception, the state of the exclusive monitors is UNKNOWN.

- For the memory location accessed by a `LoadExcl`/`StoreExcl` pair, if the memory attributes for the `LoadExcl` instruction differ from the memory attributes for the `StoreExcl` instruction, behavior is UNPREDICTABLE.

  This can occur either:

  — Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— As a result of using different virtual addresses, with different attributes, that point to the same physical address. This case is covered by another bullet point in this list.

- The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global exclusive monitor that is in the Exclusive Access state is UNPREDICTABLE. The instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same shareability domain as the PE executing the cache maintenance instruction, as determined by the shareability domain of the address being maintained.

  **Note**

  ARM strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in the Non-secure state cannot cause a denial of service on a PE in the Secure state.

- If the mapping of the virtual to physical address is changed between the `LDREX` instruction and the `STREX` instruction, and the change is performed using a break-before-make sequence as described in *Using break-before-make when updating translation table entries* on page D4-1838, if the `STREX` is performed after another write to the same physical address as the `STREX`, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the `STREX` will not pass its monitor check.

  **Note**

  ARM expects that, in many implementations, either:

  — The TLB invalidation will clear either the local or global monitor.

  — The physical address will be checked between the `LDREX` and `STREX`.

**Note**

In the event of repeatedly-contending Load-Exclusive/Store-Exclusive instruction sequences from multiple PEs, an implementation must ensure that forward progress is made by at least one PE.

### UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers

As stated in this section, an implementation can require that the instructions of a Load-Exclusive/Store-Exclusive pair access the same number of registers. In such an implementation, this means behavior is CONSTRAINED UNPREDICTABLE if, in a single thread of execution, either:

- An `LDXP` instruction of two 32-bit quantities is followed by an `STXR` instruction of one 64-bit quantity at the same address.
- An `LDXR` instruction of one 64-bit quantity is followed by an `STXP` instruction of two 32-bit quantities at the same address.

In these cases, the CONSTRAINED UNPREDICTABLE behavior must be one of:

- The `STXP` or `STXR` instruction generates an external Data Abort.
- The `STXP` or `STXR` instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Fault Status code of ESR_ELx.DFSC = `0b110101`.
- The `STXP` or `STXR` instruction always fails, returning a status of 1.
- The `STXP` or `STXR` instruction always passes, returning a status of 0.
- This `STXP` or `STXR` instruction has the same pass or fail behavior that it would have had if the instruction had used the same size and number of registers as the preceding `LDXR` or `LDXP` instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### B2.10.6 Use of WFE and SEV instructions by spin-locks

ARMv8 provides Wait For Event, Send Event, and Send Event Local instructions, `WFE`, `SEV`, and `SEVL`, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in *Wait for Event mechanism and Send event* on page D1-1611. However, in ARMv8, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

**Note**

This is equivalent to issuing an `SEV` instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an `SEV` instruction after clearing a spinlock.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Part C
## The AArch64 Instruction Set

# Chapter C1
# The A64 Instruction Set

This chapter describes the A64 instruction set. It contains the following sections:


- *Introduction* on page C1-120.
- *Structure of the A64 assembler language* on page C1-121.
- *Address generation* on page C1-126.
- *Instruction aliases* on page C1-129.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## C1.1 Introduction

The instruction set supported in the AArch64 Execution state is known as A64.

All A64 instructions have a width of 32 bits. The A64 encoding structure breaks down into the following functional groups:

- A miscellaneous group of branch instructions, exception generating instructions, and system instructions.
- Data processing instructions associated with general-purpose registers. These instructions are supported by two *functional groups*, depending on whether the operands:
  - Are all held in registers.
  - Include an operand with a constant immediate value.
- Load and store instructions associated with the general-purpose register file and the SIMD and floating-point register file.
- SIMD and scalar floating-point data processing instructions that operate on the SIMD and floating-point registers.

The encoding hierarchy within a functional group breaks down as follows:

- A functional group consists of a set of related instruction classes. *A64 instruction index by encoding* on page C4-186 provides an overview of the instruction encodings in the form of a list of instruction classes within their functional groups.
- An instruction class consists of a set of related instruction forms. Instruction forms are documented in one of two alphabetic lists:
  - The load, store, and data processing instructions associated with the general-purpose registers, together with those in the other instruction classes. See Chapter C6 *A64 Base Instruction Descriptions*.
  - The load, store, and data processing instructions associated with the SIMD and floating-point support. See Chapter C7 *A64 Advanced SIMD and Floating-point Instruction Descriptions*.
- An instruction form might support a single instruction syntax. Where an instruction supports more than one syntax, each syntax is an *instruction variant*. Instruction variants can occur because of differences in:
  - The size or format of the operands.
  - The register file used for the operands.
  - The addressing mode used for load/load/store memory operands.

  Instruction variants might also arise as the result of other factors.

  Instruction variants are described in the instruction description for the individual instructions.

A64 instructions have a regular bit encoding structure:

- 5-bit register operand fields at fixed positions within the instruction. For general-purpose register operands, the values 0-30 select one of 31 registers. The value 31 is used as a special case that can:
  - Indicate use of the current stack pointer, when identifying a load/store base register or in a limited set of data processing instructions. See *The stack pointer registers* on page D1-1523.
  - Indicate the value zero when used as a source register operand.
  - Indicate discarding the result when used as a destination register operand.

  For SIMD and floating-point register access, the value used selects one of 32 registers.
- Immediate bits that provide constant data processing values or address offsets are placed in contiguous bitfields. Some computed values in instruction variants use one or more immediate bitfields together with the secondary encoding bitfields.

All encodings that are not fully defined are described as unallocated. An attempt to execute an unallocated instruction results in an Undefined Instruction exception, unless otherwise defined in the Exception model.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C1.2 Structure of the A64 assembler language

The letter *W* denotes a general-purpose register holding a 32-bit word, and *X* denotes a general-purpose register holding a 64-bit doubleword.

An A64 assembler recognizes both upper-case and lower-case variants of the instruction mnemonics and register names, but not mixed case variants. An A64 disassembler can output either upper-case or lower-case mnemonics and register names. Program and data labels are case-sensitive.

The A64 assembly language does not require the # character to introduce constant immediate operands, but an assembler must allow immediate values introduced with or without the # character. ARM recommends that an A64 disassembler outputs a # before an immediate operand.

In Example C1-1 on page C1-122 the sequence // is used as a comment leader and A64 assemblers are encouraged to accept this syntax.

### C1.2.1 Common syntax terms

The following syntax terms are used frequently throughout the A64 instruction set description.

`UPPER` Text in upper-case letters is fixed. Text in lower-case letters is variable. This means that register name `Xn` indicates that the `X` is required, followed by a variable register number, for example `X29`.

`< >` Any text enclosed by angle braces, < >, is a value that the user supplies. Subsequent text might supply additional information.

`{ }` Any item enclosed by curly brackets, { }, is optional. A description of the item and how its presence or absence affects the instruction is normally supplied by subsequent text. In some cases curly braces are actual symbols in the syntax, for example when they surround a register list. These cases are called out in the surrounding text.

`[ ]` Any items enclosed by square brackets, [ ], constitute a list of alternative characters. A single one of the characters can be used in that position and the subsequent text describes the meaning of the alternatives. In some case the square brackets are part of the syntax itself, such as addressing modes or vector elements. These cases are called out in the surrounding text.

`a|b` Alternative words are separated by a vertical bar, |, and can be surrounded by parentheses to delimit them. For example, `U(ADD|SUB)W` represents `UADDW` or `USUBW`.

`±` This indicates an optional + or - sign. If neither is used then + is assumed.

`uimm`*n* An *n*-bit unsigned, positive, immediate value.

`simm`*n* An *n*-bit two's complement, signed immediate value, where *n* includes the sign bit.

`SP` See *Register names* on page C1-122.

`Wn` See *Register names* on page C1-122.

`WSP` See *Register names* on page C1-122.

`WZR` See *Register names* on page C1-122.

`Xn` See *Register names* on page C1-122.

`XZR` See *Register names* on page C1-122

### C1.2.2 Instruction Mnemonics

The A64 assembly language overloads instruction mnemonics and distinguishes between the different forms of an instruction based on the operand types. For example, the following `ADD` instructions all have different opcodes. However, the programmer must only remember one mnemonic, as the assembler automatically chooses the correct opcode based on the operands. The disassembler follows the same procedure in reverse.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Example C1-1 ADD instructions with different opcodes**

```
ADD W0, W1, W2              // add 32-bit register
ADD X0, X1, X2              // add 64-bit register
ADD X0, X1, W2, SXTW        // add 64-bit extended register
ADD X0, X1, #42             // add 64-bit immediate
```

### C1.2.3 Condition Code

The A64 ISA has some instructions that set condition flags or test condition codes or both. For information about instructions that set the condition flags or use the condition mnemonics, see *Condition flags and related instructions* on page C6-430.

Table C1-1 shows the available condition codes.

**Table C1-1 Condition codes**

| cond | Mnemonic | Meaning (integer) | Meaning (floating-point)[a] | Condition flags |
|---|---|---|---|---|
| 0000 | EQ | Equal | Equal | Z == 1 |
| 0001 | NE | Not equal | Not equal or unordered | Z == 0 |
| 0010 | CS or HS | Carry set | Greater than, equal, or unordered | C == 1 |
| 0011 | CC or LO | Carry clear | Less than | C == 0 |
| 0100 | MI | Minus, negative | Less than | N == 1 |
| 0101 | PL | Plus, positive or zero | Greater than, equal, or unordered | N == 0 |
| 0110 | VS | Overflow | Unordered | V == 1 |
| 0111 | VC | No overflow | Ordered | V == 0 |
| 1000 | HI | Unsigned higher | Greater than, or unordered | C ==1 && Z == 0 |
| 1001 | LS | Unsigned lower or same | Less than or equal | !(C ==1 && Z ==0) |
| 1010 | GE | Signed greater than or equal | Greater than or equal | N == V |
| 1011 | LT | Signed less than | Less than, or unordered | N! = V |
| 1100 | GT | Signed greater than | Greater than | Z == 0 && N == V |
| 1101 | LE | Signed less than or equal | Less than, equal, or unordered | !(Z == 0 && N == V) |
| 1110 | AL | Always | Always | Any |
| 1111 | NV[b] | Always | Always | Any |

a. Unordered means at least one NaN operand.

b. The condition code NV exists only to provide a valid disassembly of the 0b1111 encoding, otherwise its behavior is identical to AL.

### C1.2.4 Register names

This section describes the AArch64 registers. It contains the following subsections:

- *General-purpose register file and the stack pointer* on page C1-123.
- *SIMD and floating-point register file* on page C1-123.
- *SIMD and floating-point scalar register names* on page C1-124.
- *SIMD vector register names* on page C1-124.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *SIMD vector element names* on page C1-124.

### General-purpose register file and the stack pointer

The 31 general-purpose registers in the general-purpose register file are named R0-R30 and encoded in the instruction register fields with values 0-30. A general-purpose register field that encodes the value 31 represents either the current stack pointer or the zero register, depending on the instruction and the operand position.

When the registers are used in a specific instruction variant, they must be qualified to indicate the operand data size, 32 bits or 64 bits, and the data size of the instruction.

When the data size is 32 bits, the lower 32 bits of the register are used and the upper 32 bits are ignored on a read and cleared to zero on a write.

Table C1-2 shows the qualified names for registers, where *n* is a register number 0-30.

**Table C1-2 General-purpose register names**

| Name | Size | Encoding | Description |
|---|---|---|---|
| Wn | 32 bits | 0-30 | General-purpose register 0-30 |
| Xn | 64 bits | 0-30 | General-purpose register 0-30 |
| WZR | 32 bits | 31 | Zero register |
| XZR | 64 bits | 31 | Zero register |
| WSP | 32 bits | 31 | Current stack pointer |
| SP | 64 bits | 31 | Current stack pointer |

The following list provides further details relating to Table C1-2.

- The names Xn and Wn both refer to the same general-purpose register, Rn.
- There is no register named W31 or X31.
- The name SP represents the stack pointer for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as a read or write of the current stack pointer. When instructions do not interpret this operand encoding as the stack pointer, use of the name SP is an error.
- The name WSP represents the current stack pointer in a 32-bit context.
- The name XZR represents the zero register for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as returning zero when read or discarding the result when written. When instructions do not interpret this operand encoding as the zero register, use of the name XZR is an error.
- The name WZR represents the zero register in a 32-bit context.
- The architecture does not define a specific name for general-purpose register R30 to reflect its role as the link register on procedure calls. However, an A64 assembler must always use W30 and X30 for this purpose, and additional software names might be defined as part of the Procedure Call Standard, see *Procedure Call Standard for the ARM 64-bit Architecture*.

### SIMD and floating-point register file

The 32 registers in the SIMD and floating-point register file, V0-V31, hold floating-point operands for the scalar floating-point instructions, and both scalar and vector operands for the SIMD instructions. When they are used in a specific instruction form, the names must be further qualified to indicate the data shape, that is the data element size and the number of elements or lanes within the register. A similar requirement is placed on the general-purpose registers. See *General-purpose register file and the stack pointer*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The data type is described by the instruction mnemonics that operate on the data. The data type is not described by the register name. The data type is the interpretation of bits within each register or vector element, whether these are integers, floating-point values, polynomials or cryptographic hashes.

### SIMD and floating-point scalar register names

SIMD and floating-point instructions that operate on scalar data only access the lower bits of a SIMD and floating-point register. The unused high bits are ignored on a read and cleared to 0 on a write.

Table C1-3 shows the qualified names for accessing scalar SIMD and floating-point registers. The letter *n* denotes a register number between 0 and 31.

**Table C1-3 SIMD and floating-point scalar register names**

| Size | Name |
|---|---|
| 8 bits | `Bn` |
| 16 bits | `Hn` |
| 32 bits | `Sn` |
| 64 bits | `Dn` |
| 128 bits | `Qn` |

### SIMD vector register names

If a register holds multiple data elements on which arithmetic is performed in a parallel, SIMD, manner, then a qualifier describes the vector shape. The vector shape is the element size and the number of elements or lanes. If the element size in bits multiplied by the number of lanes does not equal 128, then the upper 64 bits of the register are ignored on a read and cleared to zero on a write.

Table C1-4 shows the SIMD vector register names. The letter *n* denotes a register number between 0 and 31.

**Table C1-4 SIMD vector register names**

| Shape | Name |
|---|---|
| 8 bits × 8 lanes | `Vn.8B` |
| 8 bits × 16 lanes | `Vn.16B` |
| 16 bits × 4 lanes | `Vn.4H` |
| 16 bits × 8 lanes | `Vn.8H` |
| 32 bits × 2 lanes | `Vn.2S` |
| 32 bits × 4 lanes | `Vn.4S` |
| 64 bits × 1 lane | `Vn.1D` |
| 64 bits × 2 lanes | `Vn.2D` |

### SIMD vector element names

Appending a constant, zero-based element index to the register name inside square brackets indicates that a single element from a SIMD and floating-point register is used as a scalar operand. The number of lanes is not represented, as it is not encoded in the instruction and can only be inferred from the index value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C1-5 shows the vector register names and the element index. The letter *i* denotes the element index.

**Table C1-5 Vector register names with element index**

| Size | Name |
|---|---|
| 8 bits | `Vn.B[i]` |
| 16 bits | `Vn.H[i]` |
| 32 bits | `Vn.S[i]` |
| 64 bits | `Vn.D[i]` |

An assembler must accept a fully qualified SIMD register name, if the number of lanes is greater than the index value. See *SIMD vector register names* on page C1-124. For example, an assembler must accept all of the following forms as the name for the 32-bit element in bits [63:32] of the SIMD and floating-point register V9:

```
V9.S[1]     //standard disassembly
V9.2S[1]    //optional number of lanes
V9.4S[1]    //optional number of lanes
```

**Note**

The SIMD and floating-point register element name `Vn.S[0]` is not equivalent to the scalar SIMD and floating-point register name `Sn`. Although they represent the same bits in the register, they select different instruction encoding forms, either the vector element or the scalar form.

### *SIMD vector register list*

Where an instruction operates on multiple SIMD and floating-point registers, for example vector Load/Store structure and table lookup operations, the registers are specified as a list enclosed by curly braces. This list consists of either a sequence of registers separated by commas, or a register range separated by a hyphen. The registers must be numbered in increasing order, modulo 32, in increments of one. The hyphenated form is preferred for disassembly if there are more than two registers in the list and the register number are increasing. The following examples are equivalent representations of a set of four registers `V4` to `V7`, each holding four lanes of 32-bit elements:

```
{ V4.4S - V7.4S }              //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S } //alternative representation
```

### *SIMD vector element list*

Registers in a list can also have a vector element form. For example, the `LD4` instruction can load one element into each of four registers, and in this case the index is appended to the list as follows:

```
{ V4.S - V7.S }[3]                //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S }[3]  //alternative with optional number of lanes
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C1.3 Address generation

The A64 instruction set supports 64-bit addresses. The valid address range is determined by the following factors:

- The size of the implemented virtual address space.
- *Memory Management Unit* (MMU) configuration settings.

The top 8 bits of the 64-bit address can be used as a tag, see *Address tagging in AArch64 state* on page D4-1742. For more information on memory management and address translation, see Chapter D4 *The AArch64 Virtual Memory System Architecture*.

### C1.3.1 Register indexed addressing

The A64 instruction set allows a 64-bit index register to be added to the 64-bit base register, with optional scaling of the index by the access size. Additionally it allows for sign-extension or zero-extension of a 32-bit value within an index register, followed by optional scaling.

### C1.3.2 PC-relative addressing

The A64 instruction set has support for position-independent code and data addressing:

- PC-relative literal loads have an offset range of ± 1MB.
- Process state flag and compare based conditional branches have a range of ± 1MB. Test bit conditional branches have a restricted range of ± 32KB.
- Unconditional branches, including branch and link, have a range of ± 128MB.

PC-relative Load/Store operations, and address generation with a range of ± 4GB can be performed using two instructions.

### C1.3.3 Load/Store addressing modes

Load/Store addressing modes in the A64 instruction set require a 64-bit base address from a general-purpose register X0-X30 or the current stack pointer, `SP`, with an optional immediate or register offset. Table C1-6 shows the assembler syntax for the complete set of Load/Store addressing modes.

**Table C1-6 A64 Load/Store addressing modes**

| Addressing Mode | Offset | | |
|---|---|---|---|
| | Immediate | Register | Extended Register |
| Base register only (no offset) | `[base{, #0}]` | - | - |
| Base plus offset | `[base{, #imm}]` | `[base, Xm{, LSL #imm}]` | `[base, Wm, (S\|U)XTW {#imm}]` |
| Pre-indexed | `[base, #imm]!` | - | - |
| Post-indexed | `[base], #imm` | `[base], Xm`[a] | - |
| Literal (PC-relative) | `label` | - | - |

a. The post-indexed by register offset mode can be used with the SIMD Load/Store structure instructions described in *Load/Store Vector* on page C3-150. Otherwise the post-indexed by register offset mode is not available.

Some types of Load/Store instruction support only a subset of the Load/Store addressing modes listed in Table C1-6. Details of the supported modes are as follows:

- Base plus offset addressing means that the address is the value in the 64-bit base register plus an offset.
- Pre-indexed addressing means that the address is the sum of the value in the 64-bit base register and an offset, and the address is then written back to the base register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Post-indexed addressing means that the address is the value in the 64-bit base register, and the sum of the address and the offset is then written back to the base register.
- Literal addressing means that the address is the value of the 64-bit program counter for this instruction plus a 19-bit signed word offset. This means that it is a 4 byte aligned address within ±1MB of the address of this instruction with no offset. Literal addressing can only be used for loads of at least 32 bits and for prefetch instructions. The PC cannot be referenced using any other addressing modes. The syntax for labels is specific to individual toolchains.
- An immediate offset can be unsigned or signed, and scaled or unscaled, depending on the type of Load/Store instruction. When the immediate offset is scaled it is encoded as a multiple of the transfer size, although the assembly language always uses a byte offset, and the assembler or disassembler performs the necessary conversion. The usable byte offsets therefore depend on the type of Load/Store instruction and the transfer size.

  Table C1-7 shows the offset and the type of Load/Store instruction.

**Table C1-7 Immediate offsets and the type of Load/Store instruction**

| Offset bits | Sign | Scaling | Write-Back | Load/Store type |
|---|---|---|---|---|
| 0 | - | - | - | Exclusive/acquire/release |
| 7 | Signed | Scaled | Optional | Register pair |
| 9 | Signed | Unscaled | Optional | Single register |
| 12 | Unsigned | Scaled | No | Single register |

- A register offset means that the offset is the 64 bits from a general-purpose register, Xm, optionally scaled by the transfer size, in bytes, if `LSL #imm` is present and where `imm` must be equal to $\log_2$(transfer_size).
- An extended register offset means that offset is the bottom 32 bits from a general-purpose register Wm, sign-extended or zero-extended to 64 bits, and then scaled by the transfer size if so indicated by `#imm`, where imm must be equal to $\log_2$(transfer_size). An assembler must accept Wm or Xm as an extended register offset, but Wm is preferred for disassembly.
- Generating an address lower than the value in the base register requires a negative signed immediate offset or a register offset holding a negative value.
- When stack alignment checking is enabled by system software and the base register is the SP, the current stack pointer must be initially quadword aligned, that is aligned to 16 bytes. Misalignment generates a Stack Alignment fault. The offset does not have to be a multiple of 16 bytes unless the specific Load/Store instruction requires this. SP cannot be used as a register offset.

### Address calculation

General-purpose arithmetic instructions can calculate the result of most addressing modes and write the address to a general-purpose register or, in most cases, to the current stack pointer.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C1-8 shows the arithmetic instructions that can compute addressing modes.

**Table C1-8 Arithmetic instructions to compute addressing modes**

| Addressing Form | Offset | | |
|---|---|---|---|
| | Immediate | Register | Extended Register |
| Base register (no offset) | `MOV Xd\|SP, base` | - | - |
| Base plus offset | `ADD Xd\|SP, base, #imm`<br>or<br>`SUB Xd\|SP, base, #imm` | `ADD <Xd\|SP>, base, Xm{,LSL#imm}` | `ADD <Xd\|SP>, base, Wm,(S\|U)XT(W\|H\|B\|) {#imm}` |
| Pre-indexed | - | - | - |
| Post-indexed | - | - | - |
| Literal (PC-relative) | `ADR Xd, label` | - | - |

**Note**

- To calculate a base plus immediate offset the `ADD` instructions defined in *Arithmetic (immediate)* on page C3-153 accept an unsigned 12-bit immediate offset, with an optional left shift by 12. This means that a single `ADD` instruction cannot support the full range of byte offsets available to a single register Load/Store with a scaled 12-bit immediate offset. For example, a quadword `LDR` effectively has a 16-bit byte offset. To calculate an address with a byte offset that requires more than 12 bits it is necessary to use two `ADD` instructions. The following example shows this:

  ```
  ADD  Xd, base, #(imm & 0xFFF)
  ADD  Xd, Xd, #(imm>>12), LSL #12
  ```

- To calculate a base plus extended register offset, the `ADD` instructions defined in *Arithmetic (extended register)* on page C3-158 provide a superset of the addressing mode that also supports sign-extension or zero-extension of a byte or halfword value with any shift amount between 0 and 4, for example:

  ```
  ADD   Xd, base, Wm, SXTW #3    // Xd = base + (SignExtend(Wm) LSL 3)
  ADD   Xd, base, Wm, UXTH #4    // Xd = base + (ZeroExtend(Wm<15:0>) LSL 4)
  ```

- If the same extended register offset is used by more than one Load/Store instruction, then, depending on the implementation, it might be more efficient to calculate the extended and scaled intermediate result just once, and then re-use it as a simple register offset. The extend and scale calculation can be performed using the `SBFIZ` and `UBFIZ` bitfield instructions defined in *Bitfield move* on page C3-155, for example:

  ```
  SBFIZ Xd, Xm, #3, #32    //Xd = “Wm, SXTW #3”
  UBFIZ Xd, Xm, #4, #16    //Xd = “Wm, UXTH #4”
  ```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C1.4 Instruction aliases

Some instructions have an associated *architecture alias* that is used for disassembly of the encoding when the associated conditions are met. Architecture alias instructions are included in the alphabetic lists of instruction types and clearly presented as an alias form in descriptions for the individual instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter C2
# About the A64 Instruction Descriptions

This chapter describes the *instruction descriptions* contained in Chapter C6 *A64 Base Instruction Descriptions* and Chapter C7 *A64 Advanced SIMD and Floating-point Instruction Descriptions*.

It contains the following sections:


- *Understanding the A64 instruction descriptions* on page C2-132.
- *Conventions used in AArch64 instruction and System register descriptions* on page C2-135.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## C2.1 Understanding the A64 instruction descriptions

Each instruction description in Chapter C6 and Chapter C7 has the following content:

1. A title.
2. An introduction to the instruction.
3. The instruction encoding or encodings.
4. Any alias conditions.
5. A list of the assembler symbols for the instruction.
6. Pseudocode describing how the instruction operates.
7. Notes, if applicable.

The following sections describe each of these.

### C2.1.1 The title

The title of an instruction description includes the base mnemonic for the instruction.

If different forms of an instruction use the same base mnemonic, each form has its own description. In this case, the title is the mnemonic followed by a short description of the instruction form in parentheses. This is most often used when an operand is an immediate value in one instruction form, but is a register in another form.

For example, in Chapter C6 there are the following titles for different forms of the `ADD` instruction:

- *ADD (extended register)* on page C6-434.
- *ADD (immediate)* on page C6-437.
- *ADD (shifted register)* on page C6-439.

### C2.1.2 An introduction to the instruction

This briefly describes the function of the instruction. The introduction is not a complete description of the instruction, and it is not definitive. If there is any conflict between it and the more detailed information that follows it, the more detailed information takes priority.

### C2.1.3 The instruction encoding or encodings

This shows the instruction encoding diagram, or if the instruction has more than one encoding, shows all of the encoding diagrams. Each diagram has a subheading.

For example, for load and store instructions, the subheadings might be:

- Post-index.
- Pre-index.
- Unsigned offset.

Each diagram numbers the bits from 31 to 0. The diagram for an instruction at address *A* shows, from left to right, the bytes at addresses *A*+3, *A*+2, *A*+1, and *A*.

There might be variants of an encoding, if the *assembler syntax prototype* differs depending on the value in one or more of the encoding fields. In this case, each variant has a subheading that describes the variant and shows the distinguishing field value or values in parentheses. For example, in Chapter C6 there are the following subheadings for variants of the `ADC` instruction encoding:

- 32-bit variant (sf = 0).
- 64-bit variant (sf = 1).

The assembler syntax prototype for an encoding or variant of an encoding shows how to form a complete assembler source code instruction that assembles to the encoding. Unless otherwise stated, the prototype is also the preferred syntax for a disassembler to disassemble the encoding to. Disassemblers are permitted to omit optional symbols that represent the default value of a field or set of fields, to produce more readable disassembled code, provided that the output re-assembles to the same encoding.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Each encoding diagram, and its associated assembler syntax prototypes, is followed by encoding-specific pseudocode that translates the fields of that encoding into inputs for the encoding-independent pseudocode that describes the operation of the instruction. See *Pseudocode describing how the instruction operates* on page C2-134.

### C2.1.4 Any alias conditions, if applicable

This is an optional part of an instruction description. If included, it describes the set of conditions for which an alternative mnemonic and its associated assembler syntax prototypes are preferred for disassembly by a disassembler. It includes a link to the alias instruction description that defines the alternative syntax. The alias syntax and the original syntax can be used interchangeably in the assembler source code.

ARM recommends that if a disassembler outputs the alias syntax, it consistently outputs the alias syntax.

### C2.1.5 A list of the assembler symbols for the instruction

The *Assembler symbols* subsection of the instruction description contains a list of the symbols that the assembler syntax prototype or prototypes use, if any.

In assembler syntax prototypes, the following conventions are used:

`< >` Angle brackets. Any symbol enclosed by these is a name or a value that the user supplies. For each symbol, there is a description of what the symbol represents. The description usually also specifies which encoding field or fields encodes the symbol.

`{ }` Brace brackets. Any symbols enclosed by these are optional. For each optional symbol, there is a description of what the symbol represents and how its presence or absence is encoded.

In some assembler syntax prototypes, some brace brackets are mandatory, for example if they surround a register list. When the use of brace brackets is mandatory, they are separated from other syntax items by one or more spaces.

`#` This usually precedes a numeric constant. All uses of # are optional in A64 assembler source code. ARM recommends that disassemblers output the # where the assembler syntax prototype includes it.

`+/-` This indicates an optional + or - sign. If neither is coded, + is assumed.

Single spaces are used for clarity, to separate syntax items. Where a space is mandatory, the assembler syntax prototype shows two or more consecutive spaces.

Any characters not shown in this conventions list must be coded exactly as shown in the assembler syntax prototype. Apart from brace brackets, the characters shown are used as part of a meta-language to define the architectural assembler syntax for an instruction encoding or alias, but have no architecturally defined significance in the input to an assembler or in the output from a disassembler.

The following symbol conventions are used:

`<Xn>` The 64-bit name of a general-purpose register (X0-X30) or the zero register (XZR).

`<Wn>` The 32-bit name of a general-purpose register (W0-W30) or the zero register (WZR).

`<Xn|SP>` The 64-bit name of a general-purpose register (X0-X30) or the current stack pointer (SP).

`<Wn|WSP>` The 32-bit name of a general-purpose register (W0-W30) or the current stack pointer (WSP).

`<Bn>, <Hn>, <Sn>, <Dn>, <Qn>`

The 8, 16, 32, 64 or 128-bit name of a SIMD and floating-point register in a scalar context as described in section *Register names* on page C1-122.

`<Vn>` The name of a SIMD and floating-point register name in a vector context as described in *Register names* on page C1-122.

If the description of a symbol specifies that the symbol is a register, the description might also specify that the range of permitted registers is extended or restricted. It also specifies any differences from the default rules for such fields.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

*Register names* on page C1-122 provides the A64 register names.

### C2.1.6 Pseudocode describing how the instruction operates

The *Operation* subsection of the instruction description contains this pseudocode.

It is encoding-independent pseudocode that provides a precise description of what the instruction does.

**Note**

For a description of ARM pseudocode, see Appendix K11 *ARM Pseudocode Definition*. This appendix also describes the execution model for an instruction.

### C2.1.7 Notes, if applicable

If applicable, other notes about the instruction appear under additional subheadings.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C2.2 Conventions used in AArch64 instruction and System register descriptions

This section provides general information about the AArch64 instruction and System register descriptions.

The terms defined in *Fixed values in AArch64 instruction and System register descriptions* apply to the AArch64 descriptions throughout this manual.

### C2.2.1 Fixed values in AArch64 instruction and System register descriptions

This section summarizes the terms used to describe fixed values in AArch64 register and instruction descriptions. The Glossary gives full descriptions of these terms, and each entry in this section includes a link to the corresponding Glossary entry.

**Note**

In register descriptions, the meaning of some bits depends on the PE state. This affects the definitions of RES0 and RES1, as shown in the Glossary.

The following terms are used to describe bits or fields with fixed values:

**RAZ** Read-As-Zero. See *Read-As-Zero (RAZ)*.

In diagrams, a RAZ bit can be shown as 0.

**(0), RES0** Reserved, *Should-Be-Zero (SBZ)* or *RES0*.

In instruction encoding diagrams, and sometimes in other descriptions, `(0)` indicates an SBZ bit. If the bit is set to 1, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction executes as if the value of the bit was 0.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(0)(0)(0)` or as `(000)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES0. See the Glossary definition of RES0 for more information.

**Note**

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES0 fields,

The `(0)` and RES0 descriptions can be applied to bits or bitfields that are read-only, or are write-only. The Glossary definitions cover these cases.

**RAO** Read-As-One. See *Read-As-One (RAO)*.

In diagrams, a RAO bit can be shown as 1.

**(1), RES1** Reserved, *Should-Be-One (SBO)* or *RES1*.

In instruction encoding diagrams, and sometimes in other descriptions, `(1)` indicates a SBO bit. If the bit is set to 0, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction executes as if the value of the bit was 1.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(1)(1)(1)` or as `(111)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES1. See the Glossary definition of RES1 for more information.

**Note**

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES1 fields,


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The (1) and RES1 descriptions can be applied to bits or bitfields that are read-only, or are write-only. The Glossary definitions cover these cases.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter C3
# A64 Instruction Set Overview

This chapter provides an overview of the A64 instruction set. It contains the following sections:

- *Branches, Exception generating, and System instructions* on page C3-138.
- *Loads and stores* on page C3-142.
- *Data processing - immediate* on page C3-153.
- *Data processing - register* on page C3-158.
- *Data processing - SIMD and floating-point* on page C3-165.

For a structured breakdown of instruction groups by encoding, see Chapter C4 *A64 Instruction Set Encoding*.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## C3.1 Branches, Exception generating, and System instructions

This section describes the branch, exception generating, and system instructions. It contains the following subsections:

- *Conditional branch*.
- *Unconditional branch (immediate)*.
- *Unconditional branch (register)* on page C3-139.
- *Exception generation and return* on page C3-139.
- *System register instructions* on page C3-140.
- *System instructions* on page C3-140.
- *Hint instructions* on page C3-141.
- *Barriers and CLREX instructions* on page C3-141.

For information about the encoding structure of the instructions in this instruction group, see *Branches, exception generating and system instructions* on page C4-191.

**Note**

Software must:

- Use only `BLR` or `BL` to perform a nested subroutine call when that subroutine is expected to return to the immediately following instruction, that is, the instruction with the address of the `BLR` or `BL` instruction incremented by four.
- Use only `RET` to perform a subroutine return, when that subroutine is expected to have been entered by a `BL` or `BLR` instruction.
- Use only `B`, `BR`, or the instructions listed in Table C3-1 to perform a control transfer that is not a subroutine call or subroutine return described in this *Note*.

### C3.1.1 Conditional branch

Conditional branches change the flow of execution depending on the current state of the condition flags or the value in a general-purpose register. See Table C1-1 on page C1-122 for a list of the condition codes that can be used for `cond`.

Table C3-1 shows the Conditional branch instructions.

**Table C3-1 Conditional branch instructions**

| Mnemonic | Instruction | Branch offset range from the PC | See |
|---|---|---|---|
| `B.cond` | Branch conditionally | ±1MB | *B.cond* on page C6-462 |
| `CBNZ` | Compare and branch if nonzero | ±1MB | *CBNZ* on page C6-476 |
| `CBZ` | Compare and branch if zero | ±1MB | *CBZ* on page C6-477 |
| `TBNZ` | Test bit and branch if nonzero | ±32KB | *TBNZ* on page C6-810 |
| `TBZ` | Test bit and branch if zero | ±32KB | *TBZ* on page C6-811 |

### C3.1.2 Unconditional branch (immediate)

Unconditional branch (immediate) instructions change the flow of execution unconditionally by adding an immediate offset with a range of ±128MB to the value of the program counter that fetched the instruction. The `BL` instruction also writes the address of the sequentially following instruction to general-purpose register, X30.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-2 shows the Unconditional branch instructions with an immediate branch offset.

**Table C3-2 Unconditional branch instructions (immediate)**

| Mnemonic | Instruction | Immediate branch offset range from the PC | See |
|---|---|---|---|
| B | Branch unconditionally | ±128MB | *B* on page C6-463 |
| BL | Branch with link | ±128MB | *BL* on page C6-472 |

### C3.1.3 Unconditional branch (register)

Unconditional branch (register) instructions change the flow of execution unconditionally by setting the program counter to the value in a general-purpose register. The BLR instruction also writes the address of the sequentially following instruction to general-purpose register X30. The RET instruction behaves identically to BR, but provides an additional hint to the PE that this is a return from a subroutine. Table C3-3 shows Unconditional branch instructions that jump directly to an address held in a general-purpose register.

**Table C3-3 Unconditional branch instructions (register)**

| Mnemonic | Instruction | See |
|---|---|---|
| BLR | Branch with link to register | *BLR* on page C6-473 |
| BR | Branch to register | *BR* on page C6-474 |
| RET | Return from subroutine | *RET* on page C6-695 |

### C3.1.4 Exception generation and return

This section describes the following exceptions:

- *Exception generating*.
- *Exception return* on page C3-140.
- *Debug state* on page C3-140.

#### Exception generating

Table C3-4 shows the Exception generating instructions.

**Table C3-4 Exception generating instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| BRK | Software Breakpoint Instruction | *BRK* on page C6-475 |
| HLT | Halting Software Breakpoint Instruction | *HLT* on page C6-531 |
| HVC | Generate exception targeting Exception level 2 | *HVC* on page C6-532 |
| SMC | Generate exception targeting Exception level 3 | *SMC* on page C6-718 |
| SVC | Generate exception targeting Exception level 1 | *SVC* on page C6-803 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Exception return

Table C3-5 shows the Exception return instructions.

**Table C3-5 Exception return instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ERET | Exception return using current ELR and SPSR | *ERET* on page C6-526 |

### Debug state

Table C3-6 shows the Debug state instructions.

**Table C3-6 Debug state instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| DCPS1 | Debug switch to Exception level 1 | *DCPS1* on page C6-512 |
| DCPS2 | Debug switch to Exception level 2 | *DCPS2* on page C6-513 |
| DCPS3 | Debug switch to Exception level 3 | *DCPS3* on page C6-514 |
| DRPS | Debug restore PE state | *DRPS* on page C6-517 |

## C3.1.5 System register instructions

For detailed information about the System register instructions, see Chapter C5 *The A64 System Instruction Class*. Table C3-7 shows the System register instructions.

**Table C3-7 System register instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| MRS | Move System register to general-purpose register | *MRS* on page C6-660 |
| MSR | • Move general-purpose register to System register<br>• Move immediate to PE state field | • *MSR (register)* on page C6-663<br>• *MSR (immediate)* on page C6-661 |

## C3.1.6 System instructions

For detailed information about the System instructions, see Chapter C5 *The A64 System Instruction Class*.

Table C3-8 shows the System instructions.

**Table C3-8 System instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| SYS | System instruction | *SYS* on page C6-807 |
| SYSL | System instruction with result | *SYSL* on page C6-809 |
| IC | Instruction cache maintenance | *IC* on page C6-533 and Table C5-2 on page C5-272 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-8 System instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| DC | Data cache maintenance | *DC* on page C6-511 and Table C5-2 on page C5-272 |
| AT | Address translation | *AT* on page C6-461 and Table C5-3 on page C5-273 |
| TLBI | TLB Invalidate | *TLBI* on page C6-812 and Table C5-4 on page C5-274 |

### C3.1.7 Hint instructions

Table C3-9 shows the Hint instructions.

**Table C3-9 Hint instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| NOP | No operation | *NOP* on page C6-675 |
| YIELD | Yield hint | *YIELD* on page C6-833 |
| WFE | Wait for event | *WFE* on page C6-831 |
| WFI | Wait for interrupt | *WFI* on page C6-832 |
| SEV | Send event | *SEV* on page C6-714 |
| SEVL | Send event local | *SEVL* on page C6-715 |
| HINT | Unallocated hint | *HINT* on page C6-529 |

### C3.1.8 Barriers and CLREX instructions

Table C3-10 shows the barrier and CLREX instructions.

**Table C3-10 Barriers and CLREX instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| CLREX | Clear exclusive monitor | *CLREX* on page C6-484 |
| DSB | Data synchronization barrier | *DSB* on page C6-518 |
| DMB | Data memory barrier | *DMB* on page C6-515 |
| ISB | Instruction synchronization barrier | *ISB* on page C6-534 |

For more information about the barriers, see *Memory barriers* on page B2-86.

For information about the allocated values for the data barriers, see:

- *DMB* on page C6-515.
- *DSB* on page C6-518.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C3.2 Loads and stores

This section describes the Load/Store instructions. It contains the following subsections:

- *Load/Store register*.
- *Load/Store register (unscaled offset)* on page C3-143.
- *Load/Store Pair* on page C3-144.
- *Load/Store Non-temporal Pair* on page C3-145.
- *Load/Store Unprivileged* on page C3-145.
- *Load-Exclusive/Store-Exclusive* on page C3-146.
- *Load-Acquire/Store-Release* on page C3-147.
- *Load/Store scalar SIMD and floating-point* on page C3-147.
- *Load/Store Vector* on page C3-150.
- *Prefetch memory* on page C3-151.

Apart from Load-Exclusive, Store-Exclusive, Load-Acquire, and Store-Release, addresses can have any alignment unless strict alignment checking is enabled, that is if SCTLR_ELx.A == 1.

The additional control bits SCTLR_ELx.SA and SCTLR_EL1.SA0 control whether the stack pointer must be quadword aligned when used as a base register. See *Stack pointer alignment checking* on page D1-1531. Using a misaligned stack pointer generates a Stack Alignment exception.

For information about the encoding structure of the instructions in this instruction group, see *Loads and stores* on page C4-196.

**Note**

In some cases, Load/Store instructions can lead to CONSTRAINED UNPREDICTABLE behavior. See *Constraints on AArch64 state UNPREDICTABLE behaviors* on page K1-5501.

### C3.2.1 Load/Store register

The Load/Store register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset, optionally scaled.
- Base plus a 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

See also *Load/Store addressing modes* on page C1-126.

If a Load instruction specifies writeback and the register being loaded is also the base register, then one of the following behaviors occurs:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs the load using the specified addressing mode and the base register becomes UNKNOWN. In addition, if an exception occurs during the execution of such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store instruction performs a writeback and the register that is stored is also the base register, then one of the following behaviors occurs:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction performs the store to the designated register using the specified addressing mode, but the value stored is UNKNOWN.

Table C3-11 shows the Load/Store Register instructions.

**Table C3-11 Load/Store register instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDR | • Load register (register offset)<br>• Load register (immediate offset)<br>• Load register (PC-relative literal) | • *LDR (register)* on page C6-567<br>• *LDR (immediate)* on page C6-561<br>• *LDR (literal)* on page C6-565 |
| LDRB | • Load byte (register offset)<br>• Load byte (immediate offset) | • *LDRB (register)* on page C6-573<br>• *LDRB (immediate)* on page C6-570 |
| LDRSB | • Load signed byte (register offset)<br>• Load signed byte (immediate offset) | • *LDRSB (register)* on page C6-585<br>• *LDRSB (immediate)* on page C6-582 |
| LDRH | • Load halfword (register offset)<br>• Load halfword (immediate offset) | • *LDRH (register)* on page C6-579<br>• *LDRH (immediate)* on page C6-576 |
| LDRSH | • Load signed halfword (register offset)<br>• Load signed halfword (immediate offset) | • *LDRSH (register)* on page C6-591<br>• *LDRSH (immediate)* on page C6-588 |
| LDRSW | • Load signed word (register offset)<br>• Load signed word (immediate offset)<br>• Load signed word (PC-relative literal) | • *LDRSW (register)* on page C6-598<br>• *LDRSW (immediate)* on page C6-594<br>• *LDRSW (literal)* on page C6-597 |
| STR | • Store register (register offset)<br>• Store register (immediate offset) | • *STR (register)* on page C6-749<br>• *STR (immediate)* on page C6-746 |
| STRB | • Store byte (register offset)<br>• Store byte (immediate offset) | • *STRB (register)* on page C6-755<br>• *STRB (immediate)* on page C6-752 |
| STRH | • Store halfword (register offset)<br>• Store halfword (immediate offset) | • *STRH (register)* on page C6-761<br>• *STRH (immediate)* on page C6-758 |

### C3.2.2 Load/Store register (unscaled offset)

The Load/Store register instructions with an unscaled offset support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See *Load/Store addressing modes* on page C1-126.

The Load/Store register (unscaled offset) instructions are required to disambiguate this instruction class from the Load/Store register instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset, because that can represent some offset values in the same range.

The ambiguous immediate offsets are byte offsets that are both:

- In the range 0-255, inclusive.
- Naturally aligned to the access size.

Other byte offsets in the range -256 to 255 inclusive are unambiguous. An assembler program translating a Load/Store instruction, for example LDR, is required to encode an unambiguous offset using the unscaled 9-bit offset form, and to encode an ambiguous offset using the scaled 12-bit offset form. A programmer might force the generation of the unscaled 9-bit form by using one of the mnemonics in Table C3-12 on page C3-144. ARM recommends that a disassembler outputs all unscaled 9-bit offset forms using one of these mnemonics, but unambiguous offsets can be output using a Load/Store single register mnemonic, for example, LDR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-12 shows the Load/Store register instructions with an unscaled offset.

**Table C3-12 Load/Store register (unscaled offset) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDUR | Load register (unscaled offset) | *LDUR* on page C6-616 |
| LDURB | Load byte (unscaled offset) | *LDURB* on page C6-618 |
| LDURSB | Load signed byte (unscaled offset) | *LDURSB* on page C6-622 |
| LDURH | Load halfword (unscaled offset) | *LDURH* on page C6-620 |
| LDURSH | Load signed halfword (unscaled offset) | *LDURSH* on page C6-624 |
| LDURSW | Load signed word (unscaled offset) | *LDURSW* on page C6-626 |
| STUR | Store register (unscaled offset) | *STUR* on page C6-771 |
| STURB | Store byte (unscaled offset) | *STURB* on page C6-773 |
| STURH | Store halfword (unscaled offset) | *STURH* on page C6-775 |

### C3.2.3 Load/Store Pair

The Load/Store Pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also *Load/Store addressing modes* on page C1-126.

If a Load Pair instruction specifies the same register for the two registers that are being loaded, then one of the following behaviors occurs:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

If a Load Pair instruction specifies writeback and one of the registers being loaded is also the base register, then one of the following behaviors occurs:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the base register becomes UNKNOWN. In addition, if an exception occurs during the instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store Pair instruction performs a writeback and one of the registers being stored is also the base register, then one of the following behaviors occurs:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the stores of the registers indicated by the specified addressing mode, but the value stored for the base register is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-13 shows the Load/Store Pair instructions.

**Table C3-13 Load/Store Pair instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDP | Load Pair | *LDP* on page C6-554 |
| LDPSW | Load Pair signed words | *LDPSW* on page C6-558 |
| STP | Store Pair | *STP* on page C6-742 |

### C3.2.4 Load/Store Non-temporal Pair

The Load/Store Non-temporal Pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See *Load/Store addressing modes* on page C1-126.

The Load/Store Non-temporal Pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be preloaded and memory writes to be gathered to accelerate bulk memory transfers.

In addition there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a Load Non-temporal Pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a Load Non-Temporal Pair instruction specifies the same register for the two registers that are being loaded, then one of the following can occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Table C3-14 shows the Load/Store Non-temporal Pair instructions.

**Table C3-14 Load/Store Non-temporal Pair instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDNP | Load Non-temporal Pair | *LDNP* on page C6-552 |
| STNP | Store Non-temporal Pair | *STNP* on page C6-740 |

### C3.2.5 Load/Store Unprivileged

The Load/Store Unprivileged instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See *Load/Store addressing modes* on page C1-126.

The Load/Store Unprivileged instructions can be used when the PE is at EL1 to perform unprivileged memory accesses. If the PE is executing in any other Exception level, then the access permissions for that level apply.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-15 shows the Load/Store Unprivileged instructions.

**Table C3-15 Load-Store Unprivileged instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDTR | Load Unprivileged register | *LDTR* on page C6-601 |
| LDTRB | Load Unprivileged byte | *LDTRB* on page C6-604 |
| LDTRSB | Load Unprivileged signed byte | *LDTRSB* on page C6-608 |
| LDTRH | Load Unprivileged halfword | *LDTRH* on page C6-606 |
| LDTRSH | Load Unprivileged signed halfword | *LDTRSH* on page C6-611 |
| LDTRSW | Load Unprivileged signed word | *LDTRSW* on page C6-614 |
| STTR | Store Unprivileged register | *STTR* on page C6-764 |
| STTRB | Store Unprivileged byte | *STTRB* on page C6-767 |
| STTRH | Store Unprivileged halfword | *STTRH* on page C6-769 |

### C3.2.6 Load-Exclusive/Store-Exclusive

The Load-Exclusive/Store-Exclusive instructions support only one addressing mode:

- Base register with no offset.

See *Load/Store addressing modes* on page C1-126.

The Load-Exclusive instructions mark the physical address being accessed as an exclusive access. This exclusive access mark is checked by the Store-Exclusive instruction, permitting the construction of atomic read-modify-write operations on shared memory variables, semaphores, mutexes, and spinlocks. See *Synchronization and semaphores* on page B2-107.

The Load-Exclusive/Store-Exclusive instructions other than Load-Exclusive pair and Store-Exclusive pair require natural alignment, and an unaligned address generates an Alignment fault. Memory accesses generated by Load-Exclusive pair or Store-Exclusive pair instructions must be aligned to the size of the pair, otherwise the access generates an Alignment fault. When a Store-Exclusive pair succeeds, it causes a single-copy atomic update of the entire memory location.

Table C3-16 shows the Load-Exclusive/Store-Exclusive instructions.

**Table C3-16 Load-Exclusive/Store-Exclusive instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDXR | Load Exclusive register | *LDXR* on page C6-631 |
| LDXRB | Load Exclusive byte | *LDXRB* on page C6-634 |
| LDXRH | Load Exclusive halfword | *LDXRH* on page C6-637 |
| LDXP | Load Exclusive pair | *LDXP* on page C6-628 |
| STXR | Store Exclusive register | *STXR* on page C6-780 |
| STXRB | Store Exclusive byte | *STXRB* on page C6-783 |
| STXRH | Store Exclusive halfword | *STXRH* on page C6-786 |
| STXP | Store Exclusive pair | *STXP* on page C6-777 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C3.2.7 Load-Acquire/Store-Release

The Load-Acquire/Store-Release instructions support only one addressing mode:

- Base register with no offset.

See *Load/Store addressing modes* on page C1-126.

The Load-Acquire/Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction. For more information about the ordering of Load-Acquire/Store-Release, see *Load-Acquire, Store-Release* on page B2-89.

The Load-Acquire/Store-Release instructions other than Load-Acquire pair and Store-Release pair require natural alignment, and an unaligned address generates an Alignment fault. Memory accesses generated by Load-Acquire pair or Store-Release pair instructions must be aligned to the size of the pair, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction only has the Release semantics if the store is successful.

Table C3-17 shows the Non-exclusive Load-Acquire/Store-Release instructions.

**Table C3-17 Non-exclusive Load-Acquire and Store-Release instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDAR | Load-Acquire register | *LDAR* on page C6-536 |
| LDARB | Load-Acquire byte | *LDARB* on page C6-538 |
| LDARH | Load-Acquire halfword | *LDARH* on page C6-539 |
| STLR | Store-Release register | *STLR* on page C6-724 |
| STLRB | Store-Release byte | *STLRB* on page C6-726 |
| STLRH | Store-Release halfword | *STLRH* on page C6-727 |

Table C3-18 shows the Exclusive Load-Acquire/Store-Release instructions.

**Table C3-18 Exclusive Load-Acquire and Store-Release instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDAXR | Load-Acquire Exclusive register | *LDAXR* on page C6-543 |
| LDAXRB | Load-Acquire Exclusive byte | *LDAXRB* on page C6-546 |
| LDAXRH | Load-Acquire Exclusive halfword | *LDAXRH* on page C6-549 |
| LDAXP | Load-Acquire Exclusive pair | *LDAXP* on page C6-540 |
| STLXR | Store-Release Exclusive register | *STLXR* on page C6-731 |
| STLXRB | Store-Release Exclusive byte | *STLXRB* on page C6-734 |
| STLXRH | Store-Release Exclusive halfword | *STLXRH* on page C6-737 |
| STLXP | Store-Release Exclusive pair | *STLXP* on page C6-728 |

### C3.2.8 Load/Store scalar SIMD and floating-point

The Load/Store scalar SIMD and floating-point instructions operate on scalar values in the SIMD and floating-point register file as described in *SIMD and floating-point scalar register names* on page C1-124. The memory addressing modes available, described in *Load/Store addressing modes* on page C1-126, are identical to the general-purpose register Load/Store instructions, and like those instructions permit arbitrary address alignment unless strict


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

alignment checking is enabled. However, unlike the Load/Store instructions that transfer general-purpose registers, Load/Store scalar SIMD and floating-point instructions make no guarantee of atomicity, even when the address is naturally aligned to the size of the data.

### Load/Store scalar SIMD and floating-point register

The Load/Store scalar SIMD and floating-point register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus unscaled 9-bit signed immediate offset.
- Base plus 64-bit register offset, optionally scaled.
- Base plus 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

For more information on the addressing modes, see *Load/Store addressing modes* on page C1-126.

**Note**

The unscaled 9-bit signed immediate offset address mode requires its own instruction form, see *Load/Store scalar SIMD and floating-point register (unscaled offset)*.

Table C3-19 shows the Load/Store instructions for a single SIMD and floating-point register.

**Table C3-19 Load/Store single SIMD and floating-point register instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDR | Load scalar SIMD&FP register (register offset) | *LDR (register, SIMD&FP)* on page C7-1139 |
| | Load scalar SIMD&FP register (immediate offset) | *LDR (immediate, SIMD&FP)* on page C7-1133 |
| | Load scalar SIMD &FP register (PC-relative literal) | *LDR (literal, SIMD&FP)* on page C7-1137 |
| STR | Store scalar SIMD &FP register (register offset) | *STR (register, SIMD&FP)* on page C7-1386 |
| | Store scalar SIMD &FP register (immediate offset) | *STR (immediate, SIMD&FP)* on page C7-1382 |

### Load/Store scalar SIMD and floating-point register (unscaled offset)

The Load /Store scalar SIMD and floating-point register instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See also *Load/Store addressing modes* on page C1-126.

The Load/Store scalar SIMD and floating-point register (unscaled offset) instructions are required to disambiguate this instruction class from the Load/Store single SIMD and floating-point instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset. This is similar to the Load/Store register (unscaled offset) instructions, that disambiguate this instruction class from the Load/Store register instruction, see *Load/Store register (unscaled offset)* on page C3-143.

Table C3-20 shows the Load/Store SIMD and floating-point register instructions with an unscaled offset.

**Table C3-20 Load/Store SIMD and floating-point register instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDUR | Load scalar SIMD&FP register (unscaled offset) | *LDUR (SIMD&FP)* on page C7-1142 |
| STUR | Store scalar SIMD&FP register (unscaled offset) | *STUR (SIMD&FP)* on page C7-1389 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Load/Store SIMD and Floating-point register pair

The Load/Store SIMD and floating-point register pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also *Load/Store addressing modes* on page C1-126.

If a Load pair instruction specifies the same register for the two registers that are being loaded, then one of the following occurs:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode and the register being loaded takes an UNKNOWN value.

Table C3-21 shows the Load/Store SIMD and floating-point register pair instructions.

**Table C3-21 Load/Store SIMD and floating-point register pair instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDP | Load pair of scalar SIMD&FP registers | *LDP (SIMD&FP)* on page C7-1130 |
| STP | Store pair of scalar SIMD&FP registers | *STP (SIMD&FP)* on page C7-1379 |

### Load/Store SIMD and Floating-point Non-temporal pair

The Load/Store SIMD and Floating-point Non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See also *Load/Store addressing modes* on page C1-126.

The Load/Store Non-temporal pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be preloaded and memory writes to be gathered to accelerate bulk memory transfers.

In addition there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a Load non-temporal pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, those memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a Load Non-temporal pair instruction specifies the same register for the two registers that are being loaded, then one of the following occurs:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-22 shows the Load/Store SIMD and floating-point Non-temporal pair instructions.

**Table C3-22 Load/Store SIMD and floating-point Non-temporal pair instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDNP | Load pair of scalar SIMD&FP registers | *LDNP (SIMD&FP)* on page C7-1128 |
| STNP | Store pair of scalar SIMD&FP registers | *STNP (SIMD&FP)* on page C7-1377 |

### C3.2.9 Load/Store Vector

The Vector Load/Store structure instructions support the following addressing modes:

- Base register only.
- Post-indexed by a 64-bit register.
- Post-indexed by an immediate, equal to the number of bytes transferred.

Load/Store vector instructions, like other Load/Store instructions, allow any address alignment, unless strict alignment checking is enabled. If strict alignment checking is enabled, then alignment checking to the size of the element is performed. However, unlike the Load/Store instructions that transfer general-purpose registers, the Load/Store vector instructions do not guarantee atomicity, even when the address is naturally aligned to the size of the element.

#### Load/Store structures

Table C3-23 shows the Load/Store structure instructions. A post-increment immediate offset, if present, must be 8, 16, 24, 32, 48, or 64, depending on the number of elements transferred.

**Table C3-23 Load/Store multiple structures instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LD1 | • Load single 1-element structure to one lane of one register<br>• Load multiple 1-element structures to one register or to two, three or four consecutive registers | • *LD1 (single structure)* on page C7-1092<br>• *LD1 (multiple structures)* on page C7-1088 |
| LD2 | • Load single 2-element structure to one lane of two consecutive registers<br>• Load multiple 2-element structures to two consecutive registers | • *LD2 (single structure)* on page C7-1101<br>• *LD2 (multiple structures)* on page C7-1098 |
| LD3 | • Load single 3-element structure to one lane of three consecutive registers<br>• Load multiple 3-element structures to three consecutive registers | • *LD3 (single structure)* on page C7-1111<br>• *LD3 (multiple structures)* on page C7-1108 |
| LD4 | • Load single 4-element structure to one lane of four consecutive registers<br>• Load multiple 4-element structures to four consecutive registers | • *LD4 (single structure)* on page C7-1121<br>• *LD4 (multiple structures)* on page C7-1118 |
| ST1 | • Store single 1-element structure from one lane of one register<br>• Store multiple 1-element structures from one register, or from two, three or four consecutive registers | • *ST1 (single structure)* on page C7-1353<br>• *ST1 (multiple structures)* on page C7-1349 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-23 Load/Store multiple structures instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| ST2 | • Store single 2-element structure from one lane of two consecutive registers<br>• Store multiple 2-element structures from two consecutive registers | • *ST2 (single structure)* on page C7-1359<br>• *ST2 (multiple structures)* on page C7-1356 |
| ST3 | • Store single 3-element structure from one lane of three consecutive registers<br>• Store multiple 3-element structures from three consecutive registers | • *ST3 (single structure)* on page C7-1366<br>• *ST3 (multiple structures)* on page C7-1363 |
| ST4 | • Store single 4-element structure from one lane of four consecutive registers<br>• Store multiple 4-element structures from four consecutive registers | • *ST4 (single structure)* on page C7-1373<br>• *ST4 (multiple structures)* on page C7-1370 |

### Load single structure and replicate

Table C3-24 shows the Load single structure and replicate instructions. A post-increment immediate offset, if present, must be 1, 2, 3, 4, 6, 8, 12, 16, 24, or 32, depending on the number of elements transferred.

**Table C3-24 Load single structure and replicate instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LD1R | Load single 1-element structure and replicate to all lanes of one register | *LD1R* on page C7-1095 |
| LD2R | Load single 2-element structure and replicate to all lanes of two registers | *LD2R* on page C7-1105 |
| LD3R | Load single 3-element structure and replicate to all lanes of three registers | *LD3R* on page C7-1115 |
| LD4R | Load single 4-element structure and replicate to all lanes of four registers | *LD4R* on page C7-1125 |

## C3.2.10 Prefetch memory

The Prefetch memory instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset. This can be optionally scaled by 8-bits, for example LSL#3.
- Base plus a 32-bit extended register offset. This can be optionally scaled by 8-bits.
- PC-relative literal.

The prefetch memory instructions signal to the memory system that memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory access when they do occur, such as pre-loading the specified address into one or more caches. Because these signals are only hints, it is valid for the PE to treat any or all prefetch instructions as a NOP.

Because they are hints to the memory system, the operation of a PRFM instruction cannot cause a synchronous exception. However, a memory operation performed as a result of one of these memory system hints might in exceptional cases trigger an asynchronous event, and thereby influence the execution of the PE. An example of an asynchronous event that might be triggered is a SError interrupt.

A PRFM instruction can only have an effect on software visible structures, such as caches and translation lookaside buffers associated with memory locations that can be accessed by reads, writes, or execution as defined in the translation regime of the current Exception level.

A PRFM instruction is guaranteed not to access Device memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A PRFM instruction using a PLI hint must not result in any access that could not be performed by the PE speculatively fetching an instruction. Therefore, if all associated MMUs are disabled, a PLI hint cannot access any memory location that cannot be accessed by instruction fetches.

The PRFM instructions require an additional <prfop> operand to be specified, which must be one of the following:

PLDL1KEEP, PLDL1STRM, PLDL2KEEP, PLDL2STRM, PLDL3KEEP, PLDL3STRM

PSTL1KEEP, PSTL1STRM, PSTL2KEEP, PSTL2STRM, PSTL3KEEP, PSTL3STRM

PLIL1KEEP, PLIL1STRM, PLIL2KEEP, PLIL2STRM, PLIL3KEEP, PLIL3STRM

<prfop> is defined as <type><target><policy>.

Here:

| | | |
|---|---|---|
| <type> | Is one of: | |
| | PLD | Prefetch for load. |
| | PST | Prefetch for store. |
| | PLI | Preload instructions. |
| <target> | Is one of: | |
| | L1 | Level 1 cache. |
| | L2 | Level 2 cache. |
| | L3 | Level 3 cache. |
| <policy> | Is one of: | |
| | KEEP | Retained or temporal prefetch, allocated in the cache normally. |
| | STRM | Streaming or non-temporal prefetch, for data that is used only once. |

PRFUM explicitly uses the unscaled 9-bit signed immediate offset addressing mode, as described in *Load/Store register (unscaled offset)* on page C3-143.

Table C3-25 shows the Prefetch memory instructions.

**Table C3-25 Prefetch memory instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| PRFM | • Prefetch memory (register offset)<br>• Prefetch memory (immediate offset)<br>• Prefetch memory (PC-relative offset) | • *PRFM (register)* on page C6-687<br>• *PRFM (immediate)* on page C6-682<br>• *PRFM (literal)* on page C6-685 |
| PRFUM | Prefetch memory (unscaled offset) | *PRFM (unscaled offset)* on page C6-690 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C3.3 Data processing - immediate

This section describes the instruction groups for data processing with immediate operands. It contains the following subsections:

- *Arithmetic (immediate)*.
- *Logical (immediate)*.
- *Move (wide immediate)* on page C3-154.
- *Move (immediate)* on page C3-154.
- *PC-relative address calculation* on page C3-155.
- *Bitfield move* on page C3-155.
- *Bitfield insert and extract* on page C3-156
- *Extract register* on page C3-156.
- *Shift (immediate)* on page C3-156.
- *Sign-extend and Zero-extend* on page C3-156.

For information about the encoding structure of the instructions in this instruction group, see *Data processing - immediate* on page C4-187.

### C3.3.1 Arithmetic (immediate)

The Arithmetic (immediate) instructions accept a 12-bit unsigned immediate value, optionally shifted left by 12 bits.

The Arithmetic (immediate) instructions that do not set condition flags can read from and write to the current stack pointer. The flag setting instructions can read from the stack pointer, but they cannot write to it.

Table C3-26 shows the Arithmetic instructions with an immediate offset.

**Table C3-26 Arithmetic instructions with an immediate**

| Mnemonic | Instruction | See |
|---|---|---|
| ADD | Add | *ADD (immediate)* on page C6-437 |
| ADDS | Add and set flags | *ADDS (immediate)* on page C6-444 |
| SUB | Subtract | *SUB (immediate)* on page C6-792 |
| SUBS | Subtract and set flags | *SUBS (immediate)* on page C6-799 |
| CMP | Compare | *CMP (immediate)* on page C6-494 |
| CMN | Compare negative | *CMN (immediate)* on page C6-489 |

### C3.3.2 Logical (immediate)

The Logical (immediate) instructions accept a bitmask immediate value that is a 32-bit pattern or a 64-bit pattern viewed as a vector of identical elements of size e = 2, 4, 8, 16, 32 or, 64 bits. Each element contains the same sub-pattern, that is a single run of 1 to (e - 1) nonzero bits from bit 0 followed by zero bits, then rotated by 0 to (e - 1) bits. This mechanism can generate 5334 unique 64-bit patterns as 2667 pairs of pattern and their bitwise inverse.

**Note**

Values that consist of only zeros or only ones cannot be described in this way.

The Logical (immediate) instructions that do not set the condition flags can write to the current stack pointer, for example to align the stack pointer in a function prologue.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Apart from ANDS, and its TST alias, Logical (immediate) instructions do not set the condition flags. However, the final results of a bitwise operation can be tested by a CBZ, CBNZ, TBZ, or TBNZ conditional branch.

Table C3-27 shows the Logical immediate instructions.

**Table C3-27 Logical immediate instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| AND | Bitwise AND | *AND (immediate)* on page C6-450 |
| ANDS | Bitwise AND and set flags | *ANDS (immediate)* on page C6-454 |
| EOR | Bitwise exclusive OR | *EOR (immediate)* on page C6-522 |
| ORR | Bitwise inclusive OR | *ORR (immediate)* on page C6-678 |
| TST | Test bits | *TST (immediate)* on page C6-814 |

### C3.3.3 Move (wide immediate)

The Move (wide immediate) instructions insert a 16-bit immediate, or inverted immediate, into a 16-bit aligned position in the destination register. The value of the other bits in the destination register depends on the variant used. The optional shift amount can be any multiple of 16 that is smaller than the register size.

Table C3-28 shows the Move (wide immediate) instructions.

**Table C3-28 Move (wide immediate) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| MOVZ | Move wide with zero | *MOVZ* on page C6-658 |
| MOVN | Move wide with NOT | *MOVN* on page C6-656 |
| MOVK | Move wide with keep | *MOVK* on page C6-654 |

### C3.3.4 Move (immediate)

The Move (immediate) instructions are aliases for a single MOVZ, MOVN, or ORR (immediate with zero register), instruction to load an immediate value into the destination register. An assembler must permit a signed or unsigned immediate, as long as its binary representation can be generated using one of these instructions, and an assembler error results if the immediate cannot be generated in this way. On disassembly it is unspecified whether the immediate is output as a signed or an unsigned value.

If there is a choice between the MOVZ, MOVN, and ORR instruction to encode the immediate, then an assembler must prefer MOVZ to MOVN, and MOVZ or MOVN to ORR, to ensure reversability. A disassembler must output ORR (immediate with zero register) MOVZ, and MOVN, as a MOV mnemonic except that the underlying instruction must be used when:

- ORR has an immediate that can be generated by a MOVZ or MOVN instruction.
- A MOVN instruction has an immediate that can be encoded by MOVZ.
- MOVZ #0 or MOVN #0 have a shift amount other than LSL #0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-29 shows the Move (immediate) instructions.

**Table C3-29 Move (immediate) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| MOV | • Move (inverted wide immediate)<br>• Move (wide immediate)<br>• Move (bitmask immediate) | • *MOV (inverted wide immediate)* on page C6-650<br>• *MOV (wide immediate)* on page C6-651<br>• *MOV (bitmask immediate)* on page C6-652 |

### C3.3.5 PC-relative address calculation

The ADR instruction adds a signed, 21-bit immediate to the value of the program counter that fetched this instruction, and then writes the result to a general-purpose register. This permits the calculation of any byte address within ±1MB of the current PC.

The ADRP instruction shifts a signed, 21-bit immediate left by 12 bits, adds it to the value of the program counter with the bottom 12 bits cleared to zero, and then writes the result to a general-purpose register. This permits the calculation of the address at a 4KB aligned memory region. In conjunction with an ADD (immediate) instruction, or a Load/Store instruction with a 12-bit immediate offset, this allows for the calculation of, or access to, any address within ±4GB of the current PC.

**Note**

The term *page* used in the ADRP description is short-hand for the 4KB memory region, and is not related to the virtual memory translation granule size.

Table C3-30 shows the instructions used for PC-relative address calculations are as follows:

**Table C3-30 PC-relative address calculation instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ADRP | Compute address of 4KB page at a PC-relative offset | *ADRP* on page C6-449 |
| ADR | Compute address of label at a PC-relative offset. | *ADR* on page C6-448 |

### C3.3.6 Bitfield move

The Bitfield move instructions copy a bitfield of constant width from bit 0 in the source register to a constant bit position in the destination register, or from a constant bit position in the source register to bit 0 in the destination register. The remaining bits in the destination register are set as follows:

- For BFM the remaining bits are unchanged.
- For UBFM the lower bits, if any, and upper bits, if any, are set to zero.
- For SBFM the lower bits, if any, are set to zero, and the upper bits, if any, are set to a copy of the most-significant bit in the copied bitfield.

Table C3-31 shows the Bitfield move instructions.

**Table C3-31 Bitfield move instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| BFM | Bitfield move | *BFM* on page C6-465 |
| SBFM | Signed bitfield move | *SBFM* on page C6-710 |
| UBFM | Unsigned bitfield move (32-bit) | *UBFM* on page C6-818 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C3.3.7 Bitfield insert and extract

The Bitfield insert and extract instructions are implemented as aliases of the Bitfield move instructions. Table C3-32 shows the Bitfield insert and extract aliases.

**Table C3-32 Bitfield insert and extract instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| BFI | Bitfield insert | *BFI* on page C6-464 |
| BFXIL | Bitfield extract and insert low | *BFXIL* on page C6-467 |
| SBFIZ | Signed bitfield insert in zero | *SBFIZ* on page C6-709 |
| SBFX | Signed bitfield extract | *SBFX* on page C6-712 |
| UBFIZ | Unsigned bitfield insert in zero | *UBFIZ* on page C6-817 |
| UBFX | Unsigned bitfield extract | *UBFX* on page C6-820 |

### C3.3.8 Extract register

Depending on the register width of the operands, the Extract register instruction copies a 32-bit or 64-bit field from a constant bit position within a double-width value formed by the concatenation of a pair of source registers to a destination register.

Table C3-33 shows the Extract (immediate) instructions.

**Table C3-33 Extract register instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| EXTR | Extract register from pair | *EXTR* on page C6-527 |

### C3.3.9 Shift (immediate)

Shifts and rotates by a constant amount are implemented as aliases of the Bitfield move or Extract register instructions. The shift or rotate amount must be in the range 0 to one less than the register width of the instruction, inclusive.

Table C3-34 shows the aliases that can be used as immediate shift and rotate instructions.

**Table C3-34 Aliases for immediate shift and rotate instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ASR | Arithmetic shift right | *ASR (immediate)* on page C6-459 |
| LSL | Logical shift left | *LSL (immediate)* on page C6-641 |
| LSR | Logical shift right | *LSR (immediate)* on page C6-644 |
| ROR | Rotate right | *ROR (immediate)* on page C6-702 |

### C3.3.10 Sign-extend and Zero-extend

The Sign-extend and Zero-extend instructions are implemented as aliases of the Bitfield move instructions.

Table C3-35 on page C3-157 shows the aliases that can be used as zero-extend and sign-extend instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-35 Zero-extend and sign-extend instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| SXTB | Sign-extend byte | *SXTB* on page C6-804 |
| SXTH | Sign-extend halfword | *SXTH* on page C6-805 |
| SXTW | Sign-extend word | *SXTW* on page C6-806 |
| UXTB | Unsigned extend byte | *UXTB* on page C6-829 |
| UXTH | Unsigned extend halfword | *UXTH* on page C6-830 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C3.4 Data processing - register

This section describes the instruction groups for data processing with all register operands. It contains the following subsections:

- *Arithmetic (shifted register)*.
- *Arithmetic (extended register)*.
- *Arithmetic with carry* on page C3-159.
- *Logical (shifted register)* on page C3-160.
- *Move (register)* on page C3-161.
- *Shift (register)* on page C3-161.
- *Multiply and divide* on page C3-161.
- *CRC32* on page C3-163.
- *Bit operation* on page C3-163.
- *Conditional select* on page C3-163.
- *Conditional comparison* on page C3-164.

For information about the encoding structure of the instructions in this instruction group, see *Data processing - register* on page C4-218.

### C3.4.1 Arithmetic (shifted register)

The Arithmetic (shifted register) instructions apply an optional shift operator to the second source register value before performing the arithmetic operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators `LSL`, `ASR` and `LSR` accept an immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator implies `LSL #0`, which means that there is no shift. A disassembler must not output `LSL #0`. However, a disassembler must output all other shifts by zero.

The current stack pointer, SP or WSP, cannot be used with this class of instructions. See *Arithmetic (extended register)* for arithmetic instructions that can operate on the current stack pointer.

Table C3-36 shows the Arithmetic (shifted register) instructions.

**Table C3-36 Arithmetic (shifted register) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ADD | Add | *ADD (shifted register)* on page C6-439 |
| ADDS | Add and set flags | *ADDS (shifted register)* on page C6-446 |
| SUB | Subtract | *SUB (shifted register)* on page C6-794 |
| SUBS | Subtract and set flags | *SUBS (shifted register)* on page C6-801 |
| CMN | Compare negative | *CMN (shifted register)* on page C6-490 |
| CMP | Compare | *CMP (shifted register)* on page C6-495 |
| NEG | Negate | *NEG (shifted register)* on page C6-669 |
| NEGS | Negate and set flags | *NEGS* on page C6-671 |

### C3.4.2 Arithmetic (extended register)

The extended register instructions provide an optional sign-extension or zero-extension of a portion of the second source register value, followed by an optional left shift by a constant amount of 1-4, inclusive.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The extended shift is described by the mandatory extend operator SXTB, SXTH, SXTW, UXTB, UXTH, or UXTW. This is followed by an optional left shift amount. If the shift amount is not specified, the default shift amount is zero. A disassembler must not output a shift amount of zero.

For 64-bit instruction forms the additional operators UXTX and SXTX use all 64 bits of the second source register with an optional shift. In that case ARM recommends UXTX as the operator. If and only if at least one register is SP, ARM recommends use of the LSL operator name, rather than UXTX, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

For 32-bit instruction forms the operators UXTW and SXTW both use all 32 bits of the second source register with an optional shift. In that case ARM recommends UXTW as the operator. If and only if at least one register is WSP, ARM recommends use of the LSL operator name, rather than UXTW, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

The non-flag setting variants of the extended register instruction permit the use of the current stack pointer as either the destination register and the first source register. The flag setting variants only permit the stack pointer to be used as the first source register.

In the 64-bit form of these instructions the final register operand is written as Wm for all except the UXTX/LSL and SXTX extend operators. For example:

```
CMP X4, W5, SXTW
ADD X1, X2, W3, UXTB #2
SUB SP, SP, X1              // SUB SP, SP, X1, UXTX #0
```

Table C3-37 shows the Arithmetic (extended register) instructions.

**Table C3-37 Arithmetic (extended register) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ADD | Add | *ADD (extended register)* on page C6-434 |
| ADDS | Add and set flags | *ADDS (extended register)* on page C6-441 |
| SUB | Subtract | *SUB (extended register)* on page C6-789 |
| SUBS | Subtract and set flags | *SUBS (extended register)* on page C6-796 |
| CMN | Compare negative | *CMN (extended register)* on page C6-487 |
| CMP | Compare | *CMP (extended register)* on page C6-492 |

### C3.4.3 Arithmetic with carry

The Arithmetic with carry instructions accept two source registers, with the carry flag as an additional input to the calculation. They do not support shifting of the second source register.

Table C3-38 shows the Arithmetic with carry instructions

**Table C3-38 Arithmetic with carry instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ADC | Add with carry | *ADC* on page C6-432 |
| ADCS | Add with carry and set flags | *ADCS* on page C6-433 |
| SBC | Subtract with carry | *SBC* on page C6-705 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-38 Arithmetic with carry instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| SBCS | Subtract with carry and set flags | *SBCS* on page C6-707 |
| NGC | Negate with carry | *NGC* on page C6-673 |
| NGCS | Negate with carry and set flags | *NGCS* on page C6-674 |

### C3.4.4 Logical (shifted register)

The Logical (shifted register) instructions apply an optional shift operator to the second source register value before performing the main operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators LSL, ASR, LSR and ROR accept a constant immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator and amount implies LSL #0, which means that there is no shift. A disassembler must not output LSL #0. However, a disassembler must output all other shifts by zero.

**Note**

Apart from ANDS, TST and BICS the logical instructions do not set the condition flags, but the final result of a bit operation can usually directly control a CBZ, CBNZ, TBZ, or TBNZ conditional branch.

Table C3-39 shows the Logical (shifted register) instructions.

**Table C3-39 Logical (shifted register) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| AND | Bitwise AND | *AND (shifted register)* on page C6-452 |
| ANDS | Bitwise AND and set flags | *ANDS (shifted register)* on page C6-456 |
| BIC | Bitwise bit clear | *BIC (shifted register)* on page C6-468 |
| BICS | Bitwise bit clear and set flags | *BICS (shifted register)* on page C6-470 |
| EON | Bitwise exclusive OR NOT | *EON (shifted register)* on page C6-520 |
| EOR | Bitwise exclusive OR | *EOR (shifted register)* on page C6-524 |
| ORR | Bitwise inclusive OR | *ORR (shifted register)* on page C6-680 |
| MVN | Bitwise NOT | *MVN* on page C6-667 |
| ORN | Bitwise inclusive OR NOT | *ORN (shifted register)* on page C6-676 |
| TST | Test bits | *TST (shifted register)* on page C6-815 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C3.4.5 Move (register)

The Move (register) instructions are aliases for other data processing instructions. They copy a value from a general-purpose register to another general-purpose register or the current stack pointer, or from the current stack pointer to a general-purpose register.

**Table C3-40 MOV register instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| MOV | • Move register<br>• Move register to SP or move SP to register | • *MOV (register)* on page C6-653<br>• *MOV (to/from SP)* on page C6-649 |

### C3.4.6 Shift (register)

In the Shift (register) instructions, the shift amount is the positive value in the second source register modulo the register size. The register width of the instruction controls whether the new bits are fed into the result on a right shift or rotate at bit[63] or bit[31].

Table C3-41 shows the Shift (register) instructions.

**Table C3-41 Shift (register) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ASRV | Arithmetic shift right variable | *ASRV* on page C6-460 |
| LSLV | Logical shift left variable | *LSLV* on page C6-642 |
| LSRV | Logical shift right variable | *LSRV* on page C6-645 |
| RORV | Rotate right variable | *RORV* on page C6-704 |

However, the Shift (register) instructions have a preferred set of aliases that match the shift immediate aliases described in *Shift (immediate)* on page C3-156.

Table C3-42 shows the aliases for Shift (register) instructions.

**Table C3-42 Aliases for Variable shift instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ASR | Arithmetic shift right | *ASR (register)* on page C6-458 |
| LSL | Logical shift left | *LSL (register)* on page C6-640 |
| LSR | Logical shift right | *LSR (register)* on page C6-643 |
| ROR | Rotate right | *ROR (register)* on page C6-703 |

### C3.4.7 Multiply and divide

This section describes the instructions used for integer multiplication and division. It contains the following subsections:

- *Multiply* on page C3-162.
- *Divide* on page C3-162.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Multiply

The Multiply instructions write to a single 32-bit or 64-bit destination register, and are built around the fundamental four operand multiply-add and multiply-subtract operation, together with 32-bit to 64-bit widening variants. A 64-bit to 128-bit widening multiple can be constructed with two instructions, using `SMULH` or `UMULH` to generate the upper 64 bits. Table C3-43 shows the Multiply instructions.

**Table C3-43 Multiply integer instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| `MADD` | Multiply-add | *MADD* on page C6-646 |
| `MSUB` | Multiply-subtract | *MSUB* on page C6-664 |
| `MNEG` | Multiply-negate | *MNEG* on page C6-648 |
| `MUL` | Multiply | *MUL* on page C6-666 |
| `SMADDL` | Signed multiply-add long | *SMADDL* on page C6-716 |
| `SMSUBL` | Signed multiply-subtract long | *SMSUBL* on page C6-720 |
| `SMNEGL` | Signed multiply-negate long | *SMNEGL* on page C6-719 |
| `SMULL` | Signed multiply long | *SMULL* on page C6-723 |
| `SMULH` | Signed multiply high | *SMULH* on page C6-722 |
| `UMADDL` | Unsigned multiply-add long | *UMADDL* on page C6-822 |
| `UMSUBL` | Unsigned multiply-subtract long | *UMSUBL* on page C6-825 |
| `UMNEGL` | Unsigned multiply-negate long | *UMNEGL* on page C6-824 |
| `UMULL` | Unsigned multiply long | *UMULL* on page C6-828 |
| `UMULH` | Unsigned multiply high | *UMULH* on page C6-827 |

### Divide

The Divide instructions compute the quotient of a division, rounded towards zero. The remainder can then be computed as (numerator - (quotient × denominator)), using the `MSUB` instruction.

If a signed integer division (`INT_MIN / -1`) is performed where `INT_MIN` is the most negative integer value representable in the selected register size, then the result overflows the signed integer range. No indication of this overflow is produced and the result that is written to the destination register is `INT_MIN`.

A division by zero results in a zero being written to the destination register, without any indication that the division by zero occurred.

Table C3-44 shows the Divide instructions.

**Table C3-44 Divide instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| `SDIV` | Signed divide | *SDIV* on page C6-713 |
| `UDIV` | Unsigned divide | *UDIV* on page C6-821 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C3.4.8 CRC32

The optional CRC32 instructions operate on the general-purpose register file to update a 32-bit CRC value from an input value comprising 1, 2, 4, or 8 bytes. There are two different classes of CRC instructions, CRC32 and CRC32C, that support two commonly used 32-bit polynomials, known as CRC-32 and CRC-32C.

To fit with common usage, the bit order of the values is reversed as part of the operation.

When bits[19:16] of ID_AA64ISAR0_EL1 are set to 0b0001 the CRC instructions are implemented.

Table C3-45 shows the CRC instructions.

**Table C3-45 CRC32 instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| CRC32B | CRC-32 sum from byte | *CRC32B, CRC32H, CRC32W, CRC32X* on page C6-498 |
| CRC32H | CRC-32 sum from halfword | *CRC32B, CRC32H, CRC32W, CRC32X* on page C6-498 |
| CRC32W | CRC-32 sum from word | *CRC32B, CRC32H, CRC32W, CRC32X* on page C6-498 |
| CRC32X | CRC-32 sum from doubleword | *CRC32B, CRC32H, CRC32W, CRC32X* on page C6-498 |
| CRC32CB | CRC-32C sum from byte | *CRC32CB, CRC32CH, CRC32CW, CRC32CX* on page C6-500 |
| CRC32CH | CRC-32C sum from halfword | *CRC32CB, CRC32CH, CRC32CW, CRC32CX* on page C6-500 |
| CRC32CW | CRC-32C sum from word | *CRC32CB, CRC32CH, CRC32CW, CRC32CX* on page C6-500 |
| CRC32CX | CRC-32C sum from doubleword | *CRC32CB, CRC32CH, CRC32CW, CRC32CX* on page C6-500 |

### C3.4.9 Bit operation

Table C3-46 shows the Bit operation instructions.

**Table C3-46 Bit operation instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| CLS | Count leading sign bits | *CLS* on page C6-485 |
| CLZ | Count leading zero bits | *CLZ* on page C6-486 |
| RBIT | Reverse bit order | *RBIT* on page C6-693 |
| REV | Reverse bytes in register | *REV* on page C6-696 |
| REV16 | Reverse bytes in halfwords | *REV16* on page C6-698 |
| REV32 | Reverses bytes in words | *REV32* on page C6-700 |

### C3.4.10 Conditional select

The Conditional select instructions select between the first or second source register, depending on the current state of the condition flags. When the named condition is true, the first source register is selected and its value is copied without modification to the destination register. When the condition is false the second source register is selected and its value might not be optionally inverted, negated, or incremented by one, before writing to the destination register.

Other useful conditional set and conditional unary operations are implemented as aliases of the four Conditional select instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-47 shows the Conditional select instructions.

**Table C3-47 Conditional select instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| CSEL | Conditional select | *CSEL* on page C6-502 |
| CSINC | Conditional select increment | *CSINC* on page C6-505 |
| CSINV | Conditional select inversion | *CSINV* on page C6-507 |
| CSNEG | Conditional select negation | *CSNEG* on page C6-509 |
| CSET | Conditional set | *CSET* on page C6-503 |
| CSETM | Conditional set mask | *CSETM* on page C6-504 |
| CINC | Conditional increment | *CINC* on page C6-482 |
| CINV | Conditional invert | *CINV* on page C6-483 |
| CNEG | Conditional negate | *CNEG* on page C6-497 |

### C3.4.11 Conditional comparison

The Conditional comparison instructions provide a conditional select for the NZCV condition flags, setting the flags to the result of an arithmetic comparison of its two source register values if the named input condition is true, or to an immediate value if the input condition is false. There are register and immediate forms. The immediate form compares the source register to a small 5-bit unsigned value.

Table C3-48 shows the Conditional comparison instructions.

**Table C3-48 Conditional comparison instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| CCMN | Conditional compare negative (register) | *CCMN (register)* on page C6-479 |
| CCMN | Conditional compare negative (immediate) | *CCMN (immediate)* on page C6-478 |
| CCMP | Conditional compare (register) | *CCMP (register)* on page C6-481 |
| CCMP | Conditional compare (immediate) | *CCMP (immediate)* on page C6-480 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C3.5 Data processing - SIMD and floating-point

This section describes the instruction groups for data processing with SIMD and floating-point register operands.

It contains the following subsections that describe the scalar floating-point data processing instructions:

- *Floating-point move (register)* on page C3-166.
- *Floating-point move (immediate)* on page C3-166.
- *Floating-point conversion* on page C3-167.
- *Floating-point round to integral* on page C3-168.
- *Floating-point multiply-add* on page C3-169.
- *Floating-point arithmetic (one source)* on page C3-169.
- *Floating-point arithmetic (two sources)* on page C3-169.
- *Floating-point minimum and maximum* on page C3-169.
- *Floating-point comparison* on page C3-170.
- *Floating-point conditional select* on page C3-170.

It also contains the following subsections that describe the SIMD data processing instructions:

- *SIMD move* on page C3-171
- *SIMD arithmetic* on page C3-171.
- *SIMD compare* on page C3-173.
- *SIMD widening and narrowing arithmetic* on page C3-174.
- *SIMD unary arithmetic* on page C3-175.
- *SIMD by element arithmetic* on page C3-177.
- *SIMD permute* on page C3-178.
- *SIMD immediate* on page C3-178.
- *SIMD shift (immediate)* on page C3-179.
- *SIMD floating-point and integer conversion* on page C3-180.
- *SIMD reduce (across vector lanes)* on page C3-181.
- *SIMD pairwise arithmetic* on page C3-181.
- *SIMD table lookup* on page C3-182.
- *The Cryptographic Extensions* on page C3-182.

For information about the encoding structure of the instructions in this instruction group, see *Data processing - SIMD and floating point* on page C4-227.

For information about the floating-point exceptions, see *Floating-point exception traps* on page D1-1568.

### C3.5.1 Common features of SIMD instructions

A number of SIMD instructions come in three forms:

- Wide:
  - This is indicated by the suffix *W*. The element width of the destination register and the first source operand is double that of the second source operand.
- Long:
  - This is indicated by the suffix *L*. The element width of the destination register is double that of both source operands.
- Narrow:
  - This is indicated by the suffix *N*. The element width of the destination register is half that of both source operands.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Furthermore, each vector form of the instruction is part of a pair, with a second and upper half suffix of *2*, to identify the variant of the instruction:

- Where a SIMD operation widens or lengthens a 64-bit vector to a 128-bit vector, the instruction provides a second part operation that can extract the source from the upper 64 bits of the source registers.
- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the instruction provides a second-part operation that can pack the result of a second operation into the upper part of the same destination register.

**Note**

This is referred to as a *lane set specifier*.

### C3.5.2 Floating-point move (register)

The Floating-point move (register) instructions copy a scalar floating-point value from one register to another register without performing any conversion.

Some of the Floating-point move (register) instructions overlap with the functionality provided by the Advanced SIMD instructions `DUP`, `INS`, and `UMOV`. However, ARM recommends using the `FMOV` instructions when operating on scalar floating-point data to avoid the creation of scalar floating-point code that depends on the availability of the Advanced SIMD instruction set.

Table C3-49 shows the Floating-point move (register) instructions.

**Table C3-49 Floating-point move (register) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| `FMOV` | • Floating-point move register without conversion<br>• Floating-point move to or from general-purpose register without conversion | • *FMOV (register)* on page C7-1032<br>• *FMOV (general)* on page C7-1034 |

### C3.5.3 Floating-point move (immediate)

The Floating-point move (immediate) instructions convert a small constant immediate floating-point value into a single-precision or double-precision scalar floating-point value in a SIMD and floating-point register.

The floating-point constant can be specified either in decimal notation, such as 12.0 or -1.2e1, or as a string beginning with `0x` followed by a hexadecimal representation of the IEEE 754 single-precision or double-precision encoding. ARM recommends that a disassembler uses the decimal notation, provided that this displays the value precisely.

The floating-point value must be expressible as $(\pm n/16 \times 2^r)$, where $n$ is an integer in the range $16 \le n \le 31$ and $r$ is an integer in the range of $-3 \le r \le 4$, that is a normalized binary floating-point encoding with one sign bit, four bits of fraction, and a 3-bit exponent.

**Note**

This encoding does not include the floating-point constant 0.0. There are several instructions that can store zero in a SIMD and floating-point register, but ARM recommends that software uses `FMOV Sd, WZR` or `FMOV Dd, XZR` to provide consistency across a range of microarchitectures.

Table C3-50 shows the Floating-point move (immediate) instruction:

**Table C3-50 Floating-point move (immediate) instruction**

| Mnemonic | Instruction | See |
|---|---|---|
| `FMOV` | Floating-point move immediate | *FMOV (scalar, immediate)* on page C7-1037 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C3.5.4 Floating-point conversion

The following subsections describe the conversion of floating-point values:

- *Convert floating-point precision*.
- *Convert between floating-point and integer or fixed-point*.

#### Convert floating-point precision

These instructions convert a floating-point scalar with one precision to a floating-point scalar with a different precision, using the current rounding mode as specified by FPCR.RMode.

Table C3-51 shows the Floating-point precision conversion instruction.

**Table C3-51 Floating-point precision conversion instruction**

| Mnemonic | Instruction | See |
|---|---|---|
| FCVT | Floating-point convert precision (scalar) | *FCVT* on page C7-934 |

#### Convert between floating-point and integer or fixed-point

These instructions convert a floating-point scalar in a SIMD and floating-point register to or from a signed or unsigned integer or fixed-point in a general-purpose register. For a fixed-point value, a final immediate operand indicates that the general-purpose register holds a fixed-point number and `fbits` indicates the number of bits after the binary point. `fbits` is in the range 1- 32 inclusive for a 32-bit general-purpose register name, and 1-64 inclusive for a 64-bit general-purpose register name.

These instructions generate the Invalid Operation exception, in response to a floating-point input of NaN, infinity, or a numerical value that cannot be represented within the destination register. An out-of-range integer or fixed-point result is saturated to the size of the destination register. A numeric result that differs from the input generates an Inexact exception. When flush-to-zero mode is enabled, zero replaces a denormal input and generates an Input Denormal exception.

Table C3-52 shows the Floating-point and fixed-point conversion instructions.

**Table C3-52 Floating-point and integer or fixed-point conversion instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| FCVTAS | Floating-point scalar convert to signed integer, rounding to nearest with ties to away (scalar form) | *FCVTAS (scalar)* on page C7-938 |
| FCVTAU | Floating-point scalar convert to unsigned integer, rounding to nearest with ties to away (scalar form) | *FCVTAU (scalar)* on page C7-942 |
| FCVTMS | Floating-point scalar convert to signed integer, rounding toward minus infinity (scalar form) | *FCVTMS (scalar)* on page C7-947 |
| FCVTMU | Floating-point scalar convert to unsigned integer, rounding toward minus infinity (scalar form) | *FCVTMU (scalar)* on page C7-951 |
| FCVTNS | Floating-point scalar convert to signed integer, rounding to nearest with ties to even (scalar form) | *FCVTNS (scalar)* on page C7-956 |
| FCVTNU | Floating-point scalar convert to unsigned integer, rounding to nearest with ties to even (scalar form) | *FCVTNU (scalar)* on page C7-960 |
| FCVTPS | Floating-point scalar convert to signed integer, rounding toward positive infinity (scalar form) | *FCVTPS (scalar)* on page C7-964 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-52 Floating-point and integer or fixed-point conversion instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| FCVTPU | Floating-point scalar convert to unsigned integer, rounding toward positive infinity (scalar form) | *FCVTPU (scalar)* on page C7-968 |
| FCVTZS | • Floating-point scalar convert to signed integer, rounding toward zero (scalar form)<br>• Floating-point convert to signed fixed-point, rounding toward zero (scalar form) | • *FCVTZS (scalar, integer)* on page C7-978<br>• *FCVTZS (scalar, fixed-point)* on page C7-976 |
| FCVTZU | • Floating-point scalar convert to unsigned integer, rounding toward zero (scalar form)<br>• Floating-point scalar convert to unsigned fixed-point, rounding toward zero (scalar form) | • *FCVTZU (scalar, integer)* on page C7-986<br>• *FCVTZU (scalar, fixed-point)* on page C7-984 |
| SCVTF | • Signed integer scalar convert to floating-point, using the current rounding mode (scalar form)<br>• Signed fixed-point convert to floating-point, using the current rounding mode (scalar form) | • *SCVTF (vector, integer)* on page C7-1214<br>• *SCVTF (scalar, fixed-point)* on page C7-1216 |
| UCVTF | • Unsigned integer scalar convert to floating-point, using the current rounding mode (scalar form)<br>• Unsigned fixed-point convert to floating-point, using the current rounding mode (scalar form) | • *UCVTF (vector, integer)* on page C7-1425<br>• *UCVTF (scalar, fixed-point)* on page C7-1427 |

### C3.5.5 Floating-point round to integral

The Floating-point round to integral instructions round a floating-point value to an integral floating-point value of the same size.

These instructions generate the Invalid Operation exception in response to a signaling NaN input, or the Input Denormal exception in response to a denormal input when flush-to-zero mode is enabled. The FRINTX instruction can also generate the Inexact exception if the result is numeric and does not have the same numerical value as the input. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as in normal floating-point arithmetic.

Table C3-53 shows the Floating-point round to integral instructions.

**Table C3-53 Floating-point round to integral instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| FRINTA | Floating-point round to integral, to nearest with ties to away | *FRINTA (scalar)* on page C7-1062 |
| FRINTI | Floating-point round to integral, using current rounding mode | *FRINTI (scalar)* on page C7-1064 |
| FRINTM | Floating-point round to integral, toward minus infinity | *FRINTM (scalar)* on page C7-1066 |
| FRINTN | Floating-point round to integral, to nearest with ties to even | *FRINTN (scalar)* on page C7-1068 |
| FRINTP | Floating-point round to integral, toward positive infinity | *FRINTP (scalar)* on page C7-1070 |
| FRINTX | Floating-point round to integral exact, using current rounding mode | *FRINTX (scalar)* on page C7-1072 |
| FRINTZ | Floating-point round to integral, toward zero | *FRINTZ (scalar)* on page C7-1074 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C3.5.6 Floating-point multiply-add

Table C3-54 shows the Floating-point multiply-add instructions that require three source register operands.

**Table C3-54 Floating-point multiply-add instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| FMADD | Floating-point scalar fused multiply-add | *FMADD* on page C7-990 |
| FMSUB | Floating-point scalar fused multiply-subtract | *FMSUB* on page C7-1038 |
| FNMADD | Floating-point scalar negated fused multiply-add | *FNMADD* on page C7-1051 |
| FNMSUB | Floating-point scalar negated fused multiply-subtract | *FNMSUB* on page C7-1053 |

### C3.5.7 Floating-point arithmetic (one source)

Table C3-55 shows the Floating-point arithmetic instructions that require a single source register operand.

**Table C3-55 Floating-point arithmetic instructions with one source register**

| Mnemonic | Instructions | See |
|---|---|---|
| FABS | Floating-point scalar absolute value | *FABS (scalar)* on page C7-899 |
| FNEG | Floating-point scalar negate | *FNEG (scalar)* on page C7-1049 |
| FSQRT | Floating-point scalar square root | *FSQRT (scalar)* on page C7-1080 |

### C3.5.8 Floating-point arithmetic (two sources)

Table C3-56 shows the Floating-point arithmetic instructions that require two source register operands.

**Table C3-56 Floating-point arithmetic instructions with two source registers**

| Mnemonic | Instruction | See |
|---|---|---|
| FADD | Floating-point scalar add | *FADD (scalar)* on page C7-906 |
| FDIV | Floating-point scalar divide | *FDIV (scalar)* on page C7-989 |
| FMUL | Floating-point scalar multiply | *FMUL (scalar)* on page C7-1043 |
| FNMUL | Floating-point scalar multiply-negate | *FNMUL* on page C7-1055 |
| FSUB | Floating-point scalar subtract | *FSUB (scalar)* on page C7-1083 |

### C3.5.9 Floating-point minimum and maximum

The `min(x,y)` and `max(x,y)` operations return a quiet NaN when either *x* or *y* is NaN. In flush-to-zero mode denormal operands are flushed to zero before comparison, and if the result of the comparison is the flushed value, then a zero value is returned. Where both *x* and *y* are zero, or denormal values flushed to zero, with different signs, then +0.0 is returned by `max()` and -0.0 by `min()`.

The `minNum(x,y)` and `maxNum(x,y)` operations follow the IEEE 754-2008 standard and return the numerical operand when one operand is numerical and the other a quiet NaN. Apart from this additional handling of a single quiet NaN the result is then identical to `min(x,y)` and `max(x,y)`.

Table C3-57 on page C3-170 shows the Floating-point instructions that can perform floating-point minimum and maximum operations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-57 Floating-point minimum and maximum instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| FMAX | Floating-point scalar maximum | *FMAX (scalar)* on page C7-994 |
| FMAXNM | Floating-point scalar maximum number | *FMAXNM (scalar)* on page C7-998 |
| FMIN | Floating-point scalar minimum | *FMIN (scalar)* on page C7-1010 |
| FMINNM | Floating-point scalar minimum number | *FMINNM (scalar)* on page C7-1014 |

### C3.5.10 Floating-point comparison

These instructions set the NZCV condition flags in PSTATE, based on the result of a comparison of two operands. If the floating-point comparisons are *unordered*, where one or both operands are a form of NaN, the C and V bits are set to 1 and the N and Z bits are cleared to 0.

**Note**

The NZCV flags in the FPSR are associated with AArch32 state. The A64 floating-point comparison instructions do not change the condition flags in the FPSR.

For the conditional Floating-point comparison instructions, if the condition is TRUE, the flags are updated to the result of the comparison, otherwise the flags are updated to the immediate value that is defined in the instruction encoding.

The quiet compare instructions generate an Invalid Operation exception if either of the source operands is a signaling NaN. The signaling compare instructions generate an Invalid Operation exception if either of the source operands is any type of NaN.

Table C3-58 shows the Floating-point comparison instructions.

**Table C3-58 Floating-point comparison instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| FCMP | Floating-point quiet compare | *FCMP* on page C7-929 |
| FCMPE | Floating-point signaling compare | *FCMPE* on page C7-931 |
| FCCMP | Floating-point conditional quiet compare | *FCCMP* on page C7-909 |
| FCCMPE | Floating-point conditional signaling compare | *FCCMPE* on page C7-911 |

### C3.5.11 Floating-point conditional select

Table C3-59 shows the Floating-point conditional select instructions.

**Table C3-59 Floating-point conditional select instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| FCSEL | Floating-point scalar conditional select | *FCSEL* on page C7-933 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C3.5.12 SIMD move

The functionality of some data movement instructions overlaps with that provided by the scalar floating-point FMOV instructions described in *Floating-point move (register)* on page C3-166.

Table C3-60 shows the SIMD move instructions.

**Table C3-60 SIMD move instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| DUP | • Duplicate vector element to vector or scalar<br>• Duplicate general-purpose register to vector | • *DUP (element)* on page C7-888<br>• *DUP (general)* on page C7-891 |
| INS | • Insert vector element from another vector element<br>• Insert vector element from general-purpose register<br>**Note**<br>Normally disassembled as MOV. | • *INS (element)* on page C7-1084<br>• *INS (general)* on page C7-1086 |
| MOV | • Move vector element to vector element<br>• Move general-purpose register to vector element<br>• Move vector element to scalar<br>• Move vector element to general-purpose register | • *MOV (element)* on page C7-1153<br>• *MOV (from general)* on page C7-1155<br>• *MOV (scalar)* on page C7-1152<br>• *MOV (to general)* on page C7-1158 |
| UMOV | Unsigned move vector element to general-purpose register | *UMOV* on page C7-1453 |
| SMOV | Signed move vector element to general-purpose register | *SMOV* on page C7-1260 |

### C3.5.13 SIMD arithmetic

Table C3-61 shows the SIMD arithmetic instructions.

**Table C3-61 SIMD arithmetic instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ADD | Add (vector and scalar form) | *ADD (vector)* on page C7-842 |
| AND | Bitwise AND (vector form) | *AND (vector)* on page C7-853 |
| BIC | Bitwise bit clear (register) (vector form) | *BIC (vector, register)* on page C7-856 |
| BIF | Bitwise insert if false (vector form) | *BIF* on page C7-857 |
| BIT | Bitwise insert if true (vector form) | *BIT* on page C7-859 |
| BSL | Bitwise select (vector form) | *BSL* on page C7-861 |
| EOR | Bitwise exclusive OR (vector form) | *EOR (vector)* on page C7-893 |
| FABD | Floating-point absolute difference (vector and scalar form) | *FABD* on page C7-896 |
| FADD | Floating-point add (vector form) | *FADD (scalar)* on page C7-906 |
| FDIV | Floating-point divide (vector form) | *FDIV (vector)* on page C7-988 |
| FMAX | Floating-point maximum (vector form) | *FMAXP (vector)* on page C7-1005 |
| FMAXNM | Floating-point maximum number (vector form) | *FMAXNM (vector)* on page C7-996 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-61 SIMD arithmetic instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| FMIN | Floating-point minimum (vector form) | *FMIN (vector)* on page C7-1008 |
| FMINNM | Floating-point minimum number (vector form) | *FMINNM (vector)* on page C7-1012 |
| FMLA | Floating-point fused multiply-add (vector form) | *FMLA (vector)* on page C7-1026 |
| FMLS | Floating-point fused multiply-subtract (vector form) | *FMLS (vector)* on page C7-1029 |
| FMUL | Floating-point multiply (vector form) | *FMUL (vector)* on page C7-1042 |
| FMULX | Floating-point multiply extended (vector and scalar form) | *FMULX* on page C7-1046 |
| FRECPS | Floating-point reciprocal step (vector and scalar form) | *FRECPS* on page C7-1058 |
| FRSQRTS | Floating-point reciprocal square root step (vector and scalar form) | *FRSQRTS* on page C7-1077 |
| FSUB | Floating-point subtract (vector form) | *FSUB (vector)* on page C7-1082 |
| MLA | Multiply-add (vector form) | *MLA (vector)* on page C7-1146 |
| MLS | Multiply-subtract (vector form) | *MLS (vector)* on page C7-1150 |
| MUL | Multiply (vector form) | *MUL (vector)* on page C7-1164 |
| MOV | Move vector register (vector form) | *MOV (vector)* on page C7-1157 |
| ORN | Bitwise inclusive OR NOT (vector form) | *ORN (vector)* on page C7-1172 |
| ORR | Bitwise inclusive OR (register) (vector form) | *ORR (vector, register)* on page C7-1175 |
| PMUL | Polynomial multiply (vector form) | *PMUL* on page C7-1177 |
| SABA | Signed absolute difference and accumulate (vector form) | *SABA* on page C7-1194 |
| SABD | Signed absolute difference (vector form) | *SABD* on page C7-1198 |
| SHADD | Signed halving add (vector form) | *SHADD* on page C7-1230 |
| SHSUB | Signed halving subtract (vector form) | *SHSUB* on page C7-1237 |
| SMAX | Signed maximum (vector form) | *SMAX* on page C7-1240 |
| SMIN | Signed minimum (vector form) | *SMIN* on page C7-1246 |
| SQADD | Signed saturating add (vector and scalar form) | *SQADD* on page C7-1268 |
| SQDMULH | Signed saturating doubling multiply returning high half (vector and scalar form) | *SQDMULH (vector)* on page C7-1285 |
| SQRSHL | Signed saturating rounding shift left (register) (vector and scalar form) | *SQRSHL* on page C7-1300 |
| SQRDMULH | Signed saturating rounding doubling multiply returning high half (vector and scalar form) | *SQRDMULH (vector)* on page C7-1298 |
| SQSHL | Signed saturating shift left (register) (vector and scalar form) | *SQSHL (register)* on page C7-1311 |
| SQSUB | Signed saturating subtract (vector and scalar form) | *SQSUB* on page C7-1322 |
| SRHADD | Signed rounding halving add (vector form) | *SRHADD* on page C7-1328 |
| SRSHL | Signed rounding shift left (register) (vector and scalar form) | *SRSHL* on page C7-1331 |
| SSHL | Signed shift left (register) (vector and scalar form) | *SSHL* on page C7-1337 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-61 SIMD arithmetic instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| SUB | Subtract (vector and scalar form) | *SUB (vector)* on page C7-1391 |
| UABA | Unsigned absolute difference and accumulate (vector form) | *UABA* on page C7-1405 |
| UABD | Unsigned absolute difference (vector form) | *UABD* on page C7-1409 |
| UHADD | Unsigned halving add (vector form) | *UHADD* on page C7-1431 |
| UHSUB | Unsigned halving subtract (vector form) | *UHSUB* on page C7-1432 |
| UMAX | Unsigned maximum (vector form) | *UMAX* on page C7-1433 |
| UMIN | Unsigned minimum (vector form) | *UMIN* on page C7-1439 |
| UQADD | Unsigned saturating add (vector and scalar form) | *UQADD* on page C7-1459 |
| UQRSHL | Unsigned saturating rounding shift left (register) (vector and scalar form) | *UQRSHL* on page C7-1461 |
| UQSHL | Unsigned saturating shift left (register) (vector and scalar form) | *UQSHL (register)* on page C7-1469 |
| UQSUB | Unsigned saturating subtract (vector and scalar form) | *UQSUB* on page C7-1474 |
| URHADD | Unsigned rounding halving add (vector form) | *URHADD* on page C7-1479 |
| URSHL | Unsigned rounding shift left (register) (vector and scalar form) | *URSHL* on page C7-1480 |
| USHL | Unsigned shift left (register) (vector and scalar form) | *USHL* on page C7-1487 |

### C3.5.14 SIMD compare

The SIMD compare instructions compare vector or scalar elements according to the specified condition and set the destination vector element to all ones if the condition holds, or to zero if the condition does not hold.

**Note**

Some of the comparisons, such as LS, LE, LO, and LT, can be made by reversing the operands and using the opposite comparison, HS, GE, HI, or GT.

Table C3-62 shows that SIMD compare instructions.

**Table C3-62 SIMD compare instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| CMEQ | • Compare bitwise equal (vector and scalar form)<br>• Compare bitwise equal to zero (vector and scalar form) | • *CMEQ (register)* on page C7-865<br>• *CMEQ (zero)* on page C7-867 |
| CMHS | Compare unsigned higher or same (vector and scalar form) | *CMHS (register)* on page C7-879 |
| CMGE | • Compare signed greater than or equal (vector and scalar form)<br>• Compare signed greater than or equal to zero (vector and scalar form) | • *CMGE (register)* on page C7-869<br>• *CMGE (zero)* on page C7-871 |
| CMHI | Compare unsigned higher (vector and scalar form) | *CMHI (register)* on page C7-877 |
| CMGT | • Compare signed greater than (vector and scalar form)<br>• Compare signed greater than zero (vector and scalar form) | • *CMGT (register)* on page C7-873<br>• *CMGT (zero)* on page C7-875 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-62 SIMD compare instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| CMLE | Compare signed less than or equal to zero (vector and scalar form) | *CMLE (zero)* on page C7-881 |
| CMLT | Compare signed less than zero (vector and scalar form) | *CMLT (zero)* on page C7-883 |
| CMTST | Compare bitwise test bits nonzero (vector and scalar form) | *CMTST* on page C7-885 |
| FCMEQ | • Floating-point compare equal (vector and scalar form)<br>• Floating-point compare equal to zero (vector and scalar form) | • *FCMEQ (register)* on page C7-913<br>• *FCMEQ (zero)* on page C7-915 |
| FCMGE | • Floating-point compare greater than or equal (vector and scalar form)<br>• Floating-point compare greater than or equal to zero (vector and scalar form) | • *FCMGE (register)* on page C7-917<br>• *FCMGE (zero)* on page C7-919 |
| FCMGT | • Floating-point compare greater than (vector and scalar form)<br>• Floating-point compare greater than zero (vector and scalar form) | • *FCMGT (register)* on page C7-921<br>• *FCMGT (zero)* on page C7-923 |
| FCMLE | Floating-point compare less than or equal to zero (vector and scalar form) | *FCMLE (zero)* on page C7-925 |
| FCMLT | Floating-point compare less than zero (vector and scalar form) | *FCMLT (zero)* on page C7-927 |
| FACGE | Floating-point absolute compare greater than or equal (vector and scalar form) | *FACGE* on page C7-901 |
| FACGT | Floating-point absolute compare greater than (vector and scalar form) | *FACGT* on page C7-903 |

### C3.5.15 SIMD widening and narrowing arithmetic

For information about the variants of these instructions, see *Common features of SIMD instructions* on page C3-165.

Table C3-63 shows the SIMD widening and narrowing arithmetic instructions.

**Table C3-63 SIMD widening and narrowing arithmetic instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ADDHN, ADDHN2 | Add returning high, narrow (vector form) | *ADDHN, ADDHN2* on page C7-844 |
| PMULL, PMULL2 | Polynomial multiply long (vector form) | *PMULL, PMULL2* on page C7-1179<br>See also *The Cryptographic Extensions* on page C3-182 |
| RADDHN, RADDHN2 | Rounding add returning high, narrow (vector form) | *RADDHN, RADDHN2* on page C7-1181 |
| RSUBHN, RSUBHN2 | Rounding subtract returning high, narrow (vector form) | *RSUBHN, RSUBHN2* on page C7-1192 |
| SABAL, SABAL2 | Signed absolute difference and accumulate long (vector form) | *SABAL, SABAL2* on page C7-1196 |
| SABDL, SABDL2 | Signed absolute difference long (vector form) | *SABDL, SABDL2* on page C7-1200 |
| SADDL, SADDL2 | Signed add long (vector form) | *SADDL, SADDL2* on page C7-1204 |
| SADDW, SADDW2 | Signed add wide (vector form) | *SADDW, SADDW2* on page C7-1210 |
| SMLAL, SMLAL2 | Signed multiply-add long (vector form) | *SMLAL, SMLAL2 (vector)* on page C7-1254 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-63 SIMD widening and narrowing arithmetic instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| SMLSL, SMLSL2 | Signed multiply-subtract long (vector form) | *SMLSL, SMLSL2 (vector)* on page C7-1258 |
| SMULL, SMULL2 | Signed multiply long (vector form) | *SMULL, SMULL2 (vector)* on page C7-1264 |
| SQDMLAL, SQDMLAL2 | Signed saturating doubling multiply-add long (vector and scalar form) | *SQDMLAL, SQDMLAL2 (vector)* on page C7-1273 |
| SQDMLSL, SQDMLSL2 | Signed saturating doubling multiply-subtract long (vector and scalar form) | *SQDMLSL, SQDMLSL2 (vector)* on page C7-1279 |
| SQDMULL, SQDMULL2 | Signed saturating doubling multiply long (vector and scalar form) | *SQDMULL, SQDMULL2 (vector)* on page C7-1290 |
| SSUBL, SSUBL2 | Signed subtract long (vector form) | *SSUBL, SSUBL2* on page C7-1345 |
| SSUBW, SSUBW2 | Signed subtract wide (vector form) | *SSUBW, SSUBW2* on page C7-1347 |
| SUBHN, SUBHN2 | Subtract returning high, narrow (vector form) | *SUBHN, SUBHN2* on page C7-1393 |
| UABAL, UABAL2 | Unsigned absolute difference and accumulate long (vector form) | *UABAL, UABAL2* on page C7-1407 |
| UABDL, UABDL2 | Unsigned absolute difference long (vector form) | *UABDL, UABDL2* on page C7-1411 |
| UADDL, UADDL2 | Unsigned add long (vector form) | *UADDL, UADDL2* on page C7-1415 |
| UADDW, UADDW2 | Unsigned add wide (vector form) | *UADDW, UADDW2* on page C7-1421 |
| UMLAL, UMLAL2 | Unsigned multiply-add long (vector form) | *UMLAL, UMLAL2 (vector)* on page C7-1447 |
| UMLSL, UMLSL2 | Unsigned multiply-subtract long (vector form) | *UMLSL, UMLSL2 (vector)* on page C7-1451 |
| UMULL, UMULL2 | Unsigned multiply long (vector form) | *UMULL, UMULL2 (vector)* on page C7-1457 |
| USUBL, USUBL2 | Unsigned subtract long (vector form) | *USUBL, USUBL2* on page C7-1497 |
| USUBW, USUBW2 | Unsigned subtract wide (vector form) | *USUBW, USUBW2* on page C7-1499 |

### C3.5.16 SIMD unary arithmetic

For information about the variants of these instructions, see *Common features of SIMD instructions* on page C3-165.

Table C3-64 shows the SIMD unary arithmetic instructions.

**Table C3-64 SIMD unary arithmetic instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ABS | Absolute value (vector and scalar form) | *ABS* on page C7-840 |
| CLS | Count leading sign bits (vector form) | *CLS (vector)* on page C7-863 |
| CLZ | Count leading zero bits (vector form) | *CLZ (vector)* on page C7-864 |
| CNT | Population count per byte (vector form) | *CNT* on page C7-887 |
| FABS | Floating-point absolute (vector form) | *FABS (vector)* on page C7-898 |
| FCVTL, FCVTL2 | Floating-point convert to higher precision long (vector form) | *FCVTL, FCVTL2* on page C7-944 |
| FCVTN, FCVTN2 | Floating-point convert to lower precision narrow (vector form) | *FCVTN, FCVTN2* on page C7-953 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-64 SIMD unary arithmetic instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| FCVTXN, FCVTXN2 | Floating-point convert to lower precision narrow, rounding to odd (vector and scalar form) | *FCVTXN, FCVTXN2* on page C7-970 |
| FNEG | Floating-point negate (vector form) | *FNEG (vector)* on page C7-1048 |
| FRECPE | Floating-point reciprocal estimate (vector and scalar form) | *FRECPE* on page C7-1056 |
| FRECPX | Floating-point reciprocal square root (scalar form) | *FRECPX* on page C7-1060 |
| FRINTA | Floating-point round to integral, to nearest with ties to away (vector form) | *FRINTA (scalar)* on page C7-1062 |
| FRINTI | Floating-point round to integral, using current rounding mode (vector form) | *FRINTI (vector)* on page C7-1063 |
| FRINTM | Floating-point round to integral, toward minus infinity (vector form) | *FRINTM (vector)* on page C7-1065 |
| FRINTN | Floating-point round to integral, to nearest with ties to even (vector form) | *FRINTN (vector)* on page C7-1067 |
| FRINTP | Floating-point round to integral, toward positive infinity (vector form) | *FRINTP (vector)* on page C7-1069 |
| FRINTX | Floating-point round to integral exact, using current rounding mode (vector form) | *FRINTX (vector)* on page C7-1071 |
| FRINTZ | Floating-point round to integral, toward zero (vector form) | *FRINTZ (vector)* on page C7-1073 |
| FRSQRTE | Floating-point reciprocal square root estimate (vector and scalar form) | *FRSQRTE* on page C7-1075 |
| FSQRT | Floating-point square root (vector form) | *FSQRT (vector)* on page C7-1079 |
| MVN | Bitwise NOT (vector form) | *MVN* on page C7-1166 |
| NEG | Negate (vector and scalar form) | *NEG (vector)* on page C7-1169 |
| NOT | Bitwise NOT (vector form) | *NOT* on page C7-1171 |
| RBIT | Bitwise reverse (vector form) | *RBIT (vector)* on page C7-1183 |
| REV16 | Reverse elements in 16-bit halfwords (vector form) | *REV16 (vector)* on page C7-1184 |
| REV32 | Reverse elements in 32-bit words (vector form) | *REV32 (vector)* on page C7-1186 |
| REV64 | Reverse elements in 64-bit doublewords (vector form) | *REV64* on page C7-1188 |
| SADALP | Signed add and accumulate long pairwise (vector form) | *SADALP* on page C7-1202 |
| SADDLP | Signed add long pairwise (vector form) | *SADDLP* on page C7-1206 |
| SQABS | Signed saturating absolute value (vector and scalar form) | *SQABS* on page C7-1266 |
| SQNEG | Signed saturating negate (vector and scalar form) | *SQNEG* on page C7-1293 |
| SQXTN, SQXTN2 | Signed saturating extract narrow (vector form) | *SQXTN, SQXTN2* on page C7-1324 |
| SQXTUN, SQXTUN2 | Signed saturating extract unsigned narrow (vector and scalar form) | *SQXTUN, SQXTUN2* on page C7-1326 |
| SUQADD | Signed saturating accumulate of unsigned value (vector and scalar form) | *SUQADD* on page C7-1395 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-64 SIMD unary arithmetic instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| SXTL, SXTL2 | Signed extend long | *SXTL, SXTL2* on page C7-1397 |
| UADALP | Unsigned add and accumulate long pairwise (vector form) | *UADALP* on page C7-1413 |
| UADDLP | Unsigned add long pairwise (vector form) | *UADDLP* on page C7-1417 |
| UQXTN, UQXTN2 | Unsigned saturating extract narrow (vector form) | *UQXTN, UQXTN2* on page C7-1476 |
| URECPE | Unsigned reciprocal estimate (vector form) | *URECPE* on page C7-1478 |
| URSQRTE | Unsigned reciprocal square root estimate (vector form) | *URSQRTE* on page C7-1484 |
| USQADD | Unsigned saturating accumulate of signed value (vector and scalar form) | *USQADD* on page C7-1493 |
| UXTL, UXTL2 | Unsigned extend long | *UXTL, UXTL2* on page C7-1501 |
| XTN, XTN2 | Extract narrow (vector form) | *XTN, XTN2* on page C7-1505 |

### C3.5.17 SIMD by element arithmetic

For information about the variants of these instructions, see *Common features of SIMD instructions* on page C3-165.

Table C3-65 shows the SIMD by element arithmetic instructions.

**Table C3-65 SIMD by element arithmetic instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| FMLA | Floating-point fused multiply-add (vector and scalar form) | *FMLA (by element)* on page C7-1024 |
| FMLS | Floating-point fused multiply-subtract (vector and scalar form) | *FMLS (by element)* on page C7-1027. |
| FMUL | Floating-point multiply (vector and scalar form) | *FMUL (by element)* on page C7-1040 |
| FMULX | Floating-point multiply extended (vector and scalar form) | *FMULX (by element)* on page C7-1044 |
| MLA | Multiply-add (vector form) | *MLA (by element)* on page C7-1144 |
| MLS | Multiply-subtract (vector form) | *MLS (by element)* on page C7-1148 |
| MUL | Multiply (vector form) | *MUL (by element)* on page C7-1162 |
| SMLAL, SMLAL2 | Signed multiply-add long (vector form) | *SMLAL, SMLAL2 (by element)* on page C7-1252 |
| SMLSL, SMLSL2 | Signed multiply-subtract long (vector form) | *SMLSL, SMLSL2 (by element)* on page C7-1256 |
| SMULL, SMULL2 | Signed multiply long (vector form) | *SMULL, SMULL2 (by element)* on page C7-1262 |
| SQDMLAL, SQDMLAL2 | Signed saturating doubling multiply-add long (vector and scalar form) | *SQDMLAL, SQDMLAL2 (by element)* on page C7-1270 |
| SQDMLSL, SQDMLSL2 | Signed saturating doubling multiply-subtract long (vector form) | *SQDMLSL, SQDMLSL2 (by element)* on page C7-1276 |
| SQDMULH | Signed saturating doubling multiply returning high half (vector and scalar form) | *SQDMULH (by element)* on page C7-1282 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-65 SIMD by element arithmetic instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| SQDMULL, SQDMULL2 | Signed saturating doubling multiply long (vector and scalar form) | *SQDMULL, SQDMULL2 (by element)* on page C7-1287 |
| SQRDMULH | Signed saturating rounding doubling multiply returning high half (vector and scalar form) | *SQRDMULH (by element)* on page C7-1295 |
| UMLAL, UMLAL2 | Unsigned multiply-add long (vector form) | *UMLAL, UMLAL2 (by element)* on page C7-1445 |
| UMLSL, UMLSL2 | Unsigned multiply-subtract long (vector form) | *UMLSL, UMLSL2 (by element)* on page C7-1449 |
| UMULL, UMULL2 | Unsigned multiply long (vector form) | *UMULL, UMULL2 (by element)* on page C7-1455 |

### C3.5.18 SIMD permute

Table C3-66 shows the SIMD permute instructions.

**Table C3-66 SIMD permute instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| EXT | Extract vector from a pair of vectors | *EXT* on page C7-895 |
| TRN1 | Transpose vectors (primary) | *TRN1* on page C7-1403 |
| TRN2 | Transpose vectors (secondary) | *TRN2* on page C7-1404 |
| UZP1 | Unzip vectors (primary) | *UZP1* on page C7-1503 |
| UZP2 | Unzip vectors (secondary) | *UZP2* on page C7-1504 |
| ZIP1 | Zip vectors (primary) | *ZIP1* on page C7-1507 |
| ZIP2 | Zip vectors (secondary) | *ZIP2* on page C7-1509 |

### C3.5.19 SIMD immediate

Table C3-67 shows the SIMD immediate instructions.

**Table C3-67 SIMD immediate instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| BIC | Bitwise bit clear immediate | *BIC (vector, immediate)* on page C7-854 |
| FMOV | Floating-point move immediate | *FMOV (vector, immediate)* on page C7-1030 |
| MOVI | Move immediate | *MOVI* on page C7-1159 |
| MVNI | Move inverted immediate | *MVNI* on page C7-1167 |
| ORR | Bitwise inclusive OR immediate | *ORR (vector, immediate)* on page C7-1173 |

### C3.5.20 SIMD shift (immediate)

For information about the variants of these instructions, see *Common features of SIMD instructions* on page C3-165.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-68 shows the SIMD shift immediate instructions.

**Table C3-68 SIMD shift (immediate) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| RSHRN, RSHRN2 | Rounding shift right narrow immediate (vector form) | *RSHRN, RSHRN2* on page C7-1190 |
| SHL | Shift left immediate (vector and scalar form) | *SHL* on page C7-1231 |
| SHLL, SHLL2 | Shift left long (by element size) (vector form) | *SHLL, SHLL2* on page C7-1233 |
| SHRN, SHRN2 | Shift right narrow immediate (vector form) | *SHRN, SHRN2* on page C7-1235 |
| SLI | Shift left and insert immediate (vector and scalar form) | *SLI* on page C7-1238 |
| SQRSHRN, SQRSHRN2 | Signed saturating rounded shift right narrow immediate (vector and scalar form) | *SQRSHRN, SQRSHRN2* on page C7-1302 |
| SQRSHRUN, SQRSHRUN2 | Signed saturating shift right unsigned narrow immediate (vector and scalar form) | *SQRSHRUN, SQRSHRUN2* on page C7-1305 |
| SQSHL | Signed saturating shift left immediate (vector and scalar form) | *SQSHL (immediate)* on page C7-1308 |
| SQSHLU | Signed saturating shift left unsigned immediate (vector and scalar form) | *SQSHLU* on page C7-1313 |
| SQSHRN, SQSHRN2 | Signed saturating shift right narrow immediate (vector and scalar form) | *SQSHRN, SQSHRN2* on page C7-1316 |
| SQSHRUN, SQSHRUN2 | Signed saturating shift right unsigned narrow immediate (vector and scalar form) | *SQSHRUN, SQSHRUN2* on page C7-1319 |
| SRI | Shift right and insert immediate (vector and scalar form) | *SRI* on page C7-1329 |
| SRSHR | Signed rounding shift right immediate (vector and scalar form) | *SRSHR* on page C7-1333 |
| SRSRA | Signed rounding shift right and accumulate immediate (vector and scalar form) | *SRSRA* on page C7-1335. |
| SSHLL, SSHLL2 | Signed shift left long immediate (vector form) | *SSHLL, SSHLL2* on page C7-1339 |
| SSHR | Signed shift right immediate (vector and scalar form) | *SSHR* on page C7-1341 |
| SSRA | Signed integer shift right and accumulate immediate (vector and scalar form) | *SSRA* on page C7-1343 |
| SXTL, SXTL2 | Signed integer extend (vector only) | *SXTL, SXTL2* on page C7-1397 |
| UQRSHRN, UQRSHRN2 | Unsigned saturating rounded shift right narrow immediate (vector and scalar form) | *UQRSHRN, UQRSHRN2* on page C7-1463 |
| UQSHL | Unsigned saturating shift left immediate (vector and scalar form) | *UQSHL (immediate)* on page C7-1466 |
| UQSHRN, UQSHRN2 | Unsigned saturating shift right narrow immediate (vector and scalar form) | *UQSHRN, UQSHRN2* on page C7-1471 |
| URSHR | Unsigned rounding shift right immediate (vector and scalar form) | *URSHR* on page C7-1482 |
| URSRA | Unsigned integer rounding shift right and accumulate immediate (vector and scalar form) | *URSRA* on page C7-1485 |
| USHLL, USHLL2 | Unsigned shift left long immediate (vector form) | *USHLL, USHLL2* on page C7-1489 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-68 SIMD shift (immediate) instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| USHR | Unsigned shift right immediate (vector and scalar form) | *USHR* on page C7-1491 |
| USRA | Unsigned shift right and accumulate immediate (vector and scalar form) | *USRA* on page C7-1495 |
| UXTL, UXTL2 | Unsigned integer extend (vector only) | *UXTL, UXTL2* on page C7-1501 |

### C3.5.21 SIMD floating-point and integer conversion

The SIMD floating-point and integer conversion instructions generate the Invalid Operation exception in response to a floating-point input of NaN, infinity, or a numerical value that cannot be represented within the destination register. An out-of-range integer or a fixed-point result is saturated to the size of the destination register. A numeric result that differs from the input raises the Inexact exception.

Table C3-69 shows the SIMD floating-point and integer conversion instructions.

**Table C3-69 SIMD floating-point and integer conversion instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| FCVTAS | Floating-point convert to signed integer, rounding to nearest with ties to away (vector and scalar form) | *FCVTAS (vector)* on page C7-936 |
| FCVTAU | Floating-point convert to unsigned integer, rounding to nearest with ties to away (vector and scalar form) | *FCVTAU (vector)* on page C7-940 |
| FCVTMS | Floating-point convert to signed integer, rounding toward minus infinity (vector and scalar form) | *FCVTMS (vector)* on page C7-945 |
| FCVTMU | Floating-point convert to unsigned integer, rounding toward minus infinity (vector and scalar form) | *FCVTMU (vector)* on page C7-949 |
| FCVTNS | Floating-point convert to signed integer, rounding to nearest with ties to even (vector and scalar form) | *FCVTNS (vector)* on page C7-954 |
| FCVTNU | Floating-point convert to unsigned integer, rounding to nearest with ties to even (vector and scalar form) | *FCVTNU (vector)* on page C7-958 |
| FCVTPS | Floating-point convert to signed integer, rounding toward positive infinity (vector and scalar form) | *FCVTPS (vector)* on page C7-962 |
| FCVTPU | Floating-point convert to unsigned integer, rounding toward positive infinity (vector and scalar form) | *FCVTPU (vector)* on page C7-966 |
| FCVTZS | • Floating-point convert to signed integer, rounding toward zero (vector and scalar form)<br>• Floating-point convert to signed fixed-point, rounding toward zero (vector and scalar form) | • *FCVTZS (vector, integer)* on page C7-974<br>• *FCVTZS (vector, fixed-point)* on page C7-972 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C3-69 SIMD floating-point and integer conversion instructions (continued)**

| Mnemonic | Instruction | See |
|---|---|---|
| FCVTZU | • Floating-point convert to unsigned integer, rounding toward zero (vector and scalar form)<br>• Floating-point convert to unsigned fixed-point, rounding toward zero, (vector and scalar form) | • *FCVTZU (vector, integer)* on page C7-982<br>• *FCVTZU (vector, fixed-point)* on page C7-980 |
| SCVTF | • Signed integer convert to floating-point (vector and scalar form)<br>• Signed fixed-point convert to floating-point (vector and scalar form) | • *SCVTF (vector, integer)* on page C7-1214<br>• *SCVTF (vector, fixed-point)* on page C7-1212 |
| UCVTF | • Unsigned integer convert to floating-point (vector and scalar form)<br>• Unsigned fixed-point convert to floating-point (vector and scalar form) | • *UCVTF (vector, integer)* on page C7-1425<br>• *UCVTF (vector, fixed-point)* on page C7-1423 |

### C3.5.22 SIMD reduce (across vector lanes)

The SIMD reduce (across vector lanes) instructions perform arithmetic operations horizontally, that is across all lanes of the input vector. They deliver a single scalar result.

Table C3-70 shows the SIMD reduce (across vector lanes) instructions.

**Table C3-70 SIMD reduce (across vector lanes) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ADDV | Add (across vector) | *ADDV* on page C7-848 |
| FMAXNMV | Floating-point maximum number (across vector) | *FMAXNMV* on page C7-1003 |
| FMAXV | Floating-point maximum (across vector) | *FMAXV* on page C7-1007 |
| FMINNMV | Floating-point minimum number (across vector) | *FMINNMV* on page C7-1019 |
| FMINV | Floating-point minimum (across vector) | *FMINV* on page C7-1023 |
| SADDLV | Signed add long (across vector) | *SADDLV* on page C7-1208 |
| SMAXV | Signed maximum (across vector) | *SMAXV* on page C7-1244 |
| SMINV | Signed minimum (across vector) | *SMINV* on page C7-1250 |
| UADDLV | Unsigned add long (across vector) | *UADDLV* on page C7-1419 |
| UMAXV | Unsigned maximum (across vector) | *UMAXV* on page C7-1437 |
| UMINV | Unsigned minimum (across vector) | *UMINV* on page C7-1443 |

### C3.5.23 SIMD pairwise arithmetic

The SIMD pairwise arithmetic instructions perform operations on pairs of adjacent elements and deliver a vector result.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-71 shows the SIMD pairwise arithmetic instructions.

**Table C3-71 SIMD pairwise arithmetic instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| ADDP | Add pairwise (vector and scalar form) | • *ADDP (vector)* on page C7-847<br>• *ADDP (scalar)* on page C7-846 |
| FADDP | Floating-point add pairwise (vector and scalar form) | • *FADDP (vector)* on page C7-908<br>• *FADDP (scalar)* on page C7-907 |
| FMAXNMP | Floating-point maximum number pairwise (vector and scalar form) | • *FMAXNMP (vector)* on page C7-1001<br>• *FMAXNMP (scalar)* on page C7-1000 |
| FMAXP | Floating-point maximum pairwise (vector and scalar form) | • *FMAXP (vector)* on page C7-1005<br>• *FMAXP (scalar)* on page C7-1004 |
| FMINNMP | Floating-point minimum number pairwise (vector and scalar form) | • *FMINNMP (vector)* on page C7-1017<br>• *FMINNMP (scalar)* on page C7-1016 |
| FMINP | Floating-point minimum pairwise (vector and scalar form) | • *FMINP (vector)* on page C7-1021<br>• *FMINP (scalar)* on page C7-1020 |
| SMAXP | Signed maximum pairwise | *SMAXP* on page C7-1242 |
| SMINP | Signed minimum pairwise | *SMINP* on page C7-1248 |
| UMAXP | Unsigned maximum pairwise | *UMAXP* on page C7-1435 |
| UMINP | Unsigned minimum pairwise | *UMINP* on page C7-1441 |

### C3.5.24 SIMD table lookup

Table C3-72 shows the SIMD table lookup instructions.

**Table C3-72 SIMD table lookup instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| TBL | Table vector lookup | *TBL* on page C7-1399 |
| TBX | Table vector lookup extension | *TBX* on page C7-1401 |

### C3.5.25 The Cryptographic Extensions

The instructions provided by the optional Cryptographic Extension share the SIMD and floating-point register file. For more information see:

- *Announcing the Advanced Encryption Standard.*
- *The Galois/Counter Mode of Operation*.
- *Announcing the Secure Hash Standard.*


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C3-73 shows the Cryptographic Extension instructions.

**Table C3-73 Cryptographic Extension instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| AESD | AES single round decryption | *AESD* on page C7-849 |
| AESE | AES single round encryption | *AESE* on page C7-850 |
| AESIMC | AES inverse mix columns | *AESIMC* on page C7-851 |
| AESMC | AES mix columns | *AESMC* on page C7-852 |
| PMULL | Polynomial multiply long | *PMULL, PMULL2* on page C7-1179 |
| SHA1C | SHA1 hash update (choose) | *SHA1C* on page C7-1220 |
| SHA1H | SHA1 fixed rotate | *SHA1H* on page C7-1221 |
| SHA1M | SHA1 hash update (majority) | *SHA1M* on page C7-1222 |
| SHA1P | SHA1 hash update (parity) | *SHA1P* on page C7-1223 |
| SHA1SU0 | SHA1 schedule update 0 | *SHA1SU0* on page C7-1224 |
| SHA1SU1 | SHA1 schedule update 1 | *SHA1SU1* on page C7-1225 |
| SHA256H | SHA256 hash update (part 1) | *SHA256H* on page C7-1227 |
| SHA256H2 | SHA256 hash update (part 2) | *SHA256H2* on page C7-1226 |
| SHA256SU0 | SHA256 schedule update 0 | *SHA256SU0* on page C7-1228 |
| SHA256SU1 | SHA256 schedule update 1 | *SHA256SU1* on page C7-1229 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter C4
# A64 Instruction Set Encoding

This chapter describes the encoding of the A64 instruction set. It contains the following sections:


- *A64 instruction index by encoding* on page C4-186.
- *Data processing - immediate* on page C4-187.
- *Branches, exception generating and system instructions* on page C4-191.
- *Loads and stores* on page C4-196.
- *Data processing - register* on page C4-218.
- *Data processing - SIMD and floating point* on page C4-227.


**Note**

In the decode tables in this chapter, an entry of - for a field value means the value of the field does not affect the decoding.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## C4.1 A64 instruction index by encoding

The encoding of an A64 instruction is:

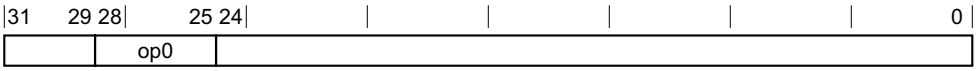



**Table C4-1 Main encoding table for the A64 instruction set**

| Decode fields<br>op0 | Decode group or instruction page |
|---|---|
| 00xx | Unallocated. |
| 100x | *Data processing - immediate* on page C4-187 |
| 101x | *Branches, exception generating and system instructions* on page C4-191 |
| x1x0 | *Loads and stores* on page C4-196 |
| x101 | *Data processing - register* on page C4-218 |
| 0111 | *Data processing - SIMD and floating point* on page C4-227 |
| 1111 | *Data processing - SIMD and floating point* on page C4-227 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C4.2 Data processing - immediate

This section describes the encoding of the Data processing (immediate) instruction group. This section is decoded from *A64 instruction index by encoding* on page C4-186. For additional information on this functional group of instructions, see *Data processing - immediate* on page C3-153.

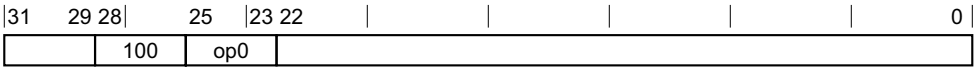
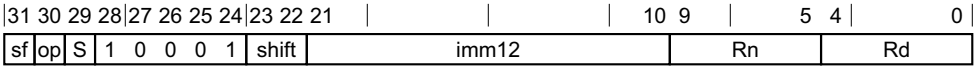



**Table C4-2 Encoding table for the Data Processing -- Immediate group**

| Decode fields | Decode group or instruction page |
|---|---|
| op0 | |
| 00x | *PC-rel. addressing* on page C4-190 |
| 01x | *Add/subtract (immediate)* |
| 100 | *Logical (immediate)* on page C4-189 |
| 101 | *Move wide (immediate)* on page C4-189 |
| 110 | *Bitfield* on page C4-188 |
| 111 | *Extract* on page C4-188 |

### C4.2.1 Add/subtract (immediate)

This section describes the encoding of the Add/subtract (immediate) instruction class. This section is decoded from *Data processing - immediate*.

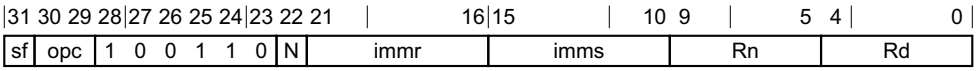



| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| sf | op | S | shift | |
| - | - | - | 1x | Unallocated. |
| 0 | 0 | 0 | - | ADD (immediate) - 32-bit variant |
| 0 | 0 | 1 | - | ADDS (immediate) - 32-bit variant |
| 0 | 1 | 0 | - | SUB (immediate) - 32-bit variant |
| 0 | 1 | 1 | - | SUBS (immediate) - 32-bit variant |
| 1 | 0 | 0 | - | ADD (immediate) - 64-bit variant |
| 1 | 0 | 1 | - | ADDS (immediate) - 64-bit variant |
| 1 | 1 | 0 | - | SUB (immediate) - 64-bit variant |
| 1 | 1 | 1 | - | SUBS (immediate) - 64-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C4.2.2 Bitfield

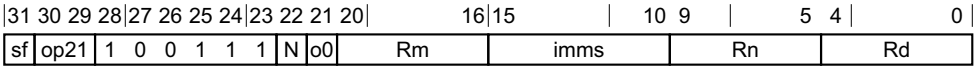
This section describes the encoding of the Bitfield instruction class. This section is decoded from *Data processing - immediate* on page C4-187.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| sf | opc | N | |
| - | 11 | - | Unallocated. |
| 0 | - | 1 | Unallocated. |
| 0 | 00 | 0 | SBFM - 32-bit variant |
| 0 | 01 | 0 | BFM - 32-bit variant |
| 0 | 10 | 0 | UBFM - 32-bit variant |
| 1 | - | 0 | Unallocated. |
| 1 | 00 | 1 | SBFM - 64-bit variant |
| 1 | 01 | 1 | BFM - 64-bit variant |
| 1 | 10 | 1 | UBFM - 64-bit variant |

### C4.2.3 Extract

This section describes the encoding of the Extract instruction class. This section is decoded from *Data processing - immediate* on page C4-187.

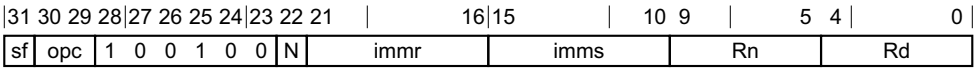



| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| sf | op21 | N | o0 | imms | |
| - | x1 | - | - | - | Unallocated. |
| - | 00 | - | 1 | - | Unallocated. |
| - | 1x | - | - | - | Unallocated. |
| 0 | - | - | - | 1xxxxx | Unallocated. |
| 0 | - | 1 | - | - | Unallocated. |
| 0 | 00 | 0 | 0 | 0xxxxx | EXTR - 32-bit variant |
| 1 | - | 0 | - | - | Unallocated. |
| 1 | 00 | 1 | 0 | - | EXTR - 64-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C4.2.4 Logical (immediate)

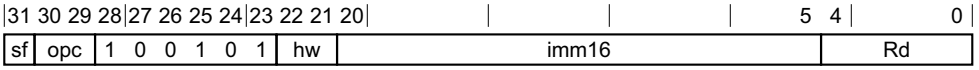
This section describes the encoding of the Logical (immediate) instruction class. This section is decoded from *Data processing - immediate* on page C4-187.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| sf | opc | N | |
| 0 | - | 1 | Unallocated. |
| 0 | 00 | 0 | AND (immediate) - 32-bit variant |
| 0 | 01 | 0 | ORR (immediate) - 32-bit variant |
| 0 | 10 | 0 | EOR (immediate) - 32-bit variant |
| 0 | 11 | 0 | ANDS (immediate) - 32-bit variant |
| 1 | 00 | - | AND (immediate) - 64-bit variant |
| 1 | 01 | - | ORR (immediate) - 64-bit variant |
| 1 | 10 | - | EOR (immediate) - 64-bit variant |
| 1 | 11 | - | ANDS (immediate) - 64-bit variant |

### C4.2.5 Move wide (immediate)

This section describes the encoding of the Move wide (immediate) instruction class. This section is decoded from *Data processing - immediate* on page C4-187.

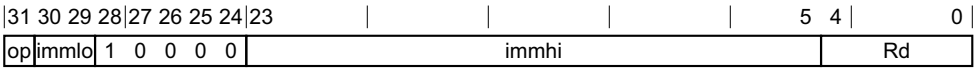



| Decode fields | | | Instruction Page |
|---|---|---|---|
| sf | opc | hw | |
| - | 01 | - | Unallocated. |
| 0 | - | 1x | Unallocated. |
| 0 | 00 | - | MOVN - 32-bit variant |
| 0 | 10 | - | MOVZ - 32-bit variant |
| 0 | 11 | - | MOVK - 32-bit variant |
| 1 | 00 | - | MOVN - 64-bit variant |
| 1 | 10 | - | MOVZ - 64-bit variant |
| 1 | 11 | - | MOVK - 64-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C4.2.6 PC-rel. addressing

This section describes the encoding of the PC-rel. addressing instruction class. This section is decoded from *Data processing - immediate* on page C4-187.




| Decode fields | Instruction Page |
|---|---|
| op | |
| 0 | ADR |
| 1 | ADRP |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C4.3 Branches, exception generating and system instructions

This section describes the encoding of the Branch, exception generation and system instruction group. This section is decoded from *A64 instruction index by encoding* on page C4-186. For additional information on this functional group of instructions, see *Branches, Exception generating, and System instructions* on page C3-138.

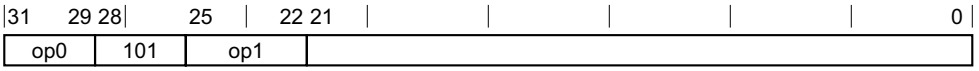
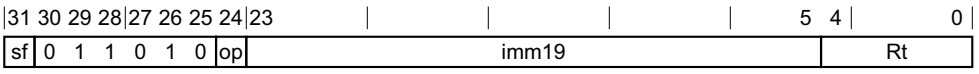



**Table C4-3 Encoding table for the Branches, Exception Generating and System instructions group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 010 | 0xxx | *Conditional branch (immediate)* on page C4-192 |
| 010 | 1xxx | Unallocated. |
| 110 | 00xx | *Exception generation* on page C4-192 |
| 110 | 0100 | *System* on page C4-193 |
| 110 | 0101 | Unallocated. |
| 110 | 011x | Unallocated. |
| 110 | 1xxx | *Unconditional branch (register)* on page C4-194 |
| x00 | - | *Unconditional branch (immediate)* on page C4-194 |
| x01 | 0xxx | *Compare & branch (immediate)* |
| x01 | 1xxx | *Test & branch (immediate)* on page C4-194 |
| x11 | - | Unallocated. |

### C4.3.1 Compare & branch (immediate)

This section describes the encoding of the Compare & branch (immediate) instruction class. This section is decoded from *Branches, exception generating and system instructions*.

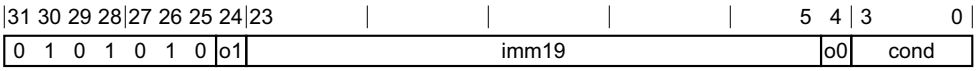



| Decode fields | | Instruction Page |
|---|---|---|
| sf | op | |
| 0 | 0 | CBZ - 32-bit variant |
| 0 | 1 | CBNZ - 32-bit variant |
| 1 | 0 | CBZ - 64-bit variant |
| 1 | 1 | CBNZ - 64-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C4.3.2 Conditional branch (immediate)

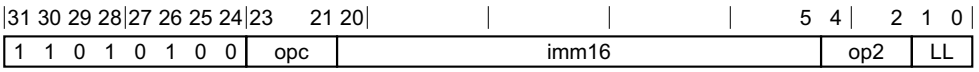
This section describes the encoding of the Conditional branch (immediate) instruction class. This section is decoded from *Branches, exception generating and system instructions* on page C4-191.




| Decode fields | | Instruction Page |
|---|---|---|
| o1 | o0 | |
| 0 | 0 | B.cond |
| 0 | 1 | Unallocated. |
| 1 | - | Unallocated. |

### C4.3.3 Exception generation

This section describes the encoding of the Exception generation instruction class. This section is decoded from *Branches, exception generating and system instructions* on page C4-191.



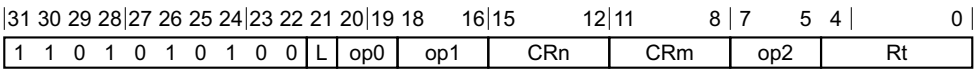

| Decode fields | | | Instruction Page |
|---|---|---|---|
| opc | op2 | LL | |
| - | xx1 | - | Unallocated. |
| - | x1x | - | Unallocated. |
| - | 1xx | - | Unallocated. |
| 000 | 000 | 00 | Unallocated. |
| 000 | 000 | 01 | SVC |
| 000 | 000 | 10 | HVC |
| 000 | 000 | 11 | SMC |
| 001 | 000 | x1 | Unallocated. |
| 001 | 000 | 00 | BRK |
| 001 | 000 | 1x | Unallocated. |
| 010 | 000 | x1 | Unallocated. |
| 010 | 000 | 00 | HLT |
| 010 | 000 | 1x | Unallocated. |
| 011 | 000 | - | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| opc | op2 | LL | |
| 100 | 000 | - | Unallocated. |
| 101 | 000 | 00 | Unallocated. |
| 101 | 000 | 01 | DCPS1 |
| 101 | 000 | 10 | DCPS2 |
| 101 | 000 | 11 | DCPS3 |
| 11x | 000 | - | Unallocated. |

### C4.3.4 System

This section describes the encoding of the System instruction class. This section is decoded from *Branches, exception generating and system instructions* on page C4-191.



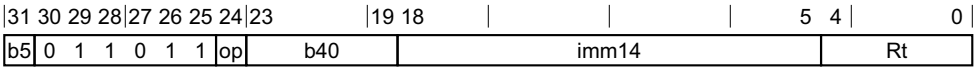

| Decode fields | | | | | | Instruction Page |
|---|---|---|---|---|---|---|
| L | op0 | op1 | CRn | op2 | Rt | |
| 0 | 00 | - | 000x | - | - | Unallocated. |
| 0 | 00 | - | 0100 | - | != 11111 | Unallocated. |
| 0 | 00 | - | 0100 | - | 11111 | MSR (immediate) |
| 0 | 00 | - | 0101 | - | - | Unallocated. |
| 0 | 00 | - | 011x | - | - | Unallocated. |
| 0 | 00 | - | 1xxx | - | - | Unallocated. |
| 0 | 00 | xx0 | 001x | - | - | Unallocated. |
| 0 | 00 | x0x | 001x | - | - | Unallocated. |
| 0 | 00 | 011 | 001x | - | != 11111 | Unallocated. |
| 0 | 00 | 011 | 0010 | - | 11111 | HINT |
| 0 | 00 | 011 | 0011 | 000 | - | Unallocated. |
| 0 | 00 | 011 | 0011 | 001 | - | Unallocated. |
| 0 | 00 | 011 | 0011 | 010 | 11111 | CLREX |
| 0 | 00 | 011 | 0011 | 011 | - | Unallocated. |
| 0 | 00 | 011 | 0011 | 100 | 11111 | DSB |
| 0 | 00 | 011 | 0011 | 101 | 11111 | DMB |
| 0 | 00 | 011 | 0011 | 110 | 11111 | ISB |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | | Instruction Page |
|---|---|---|---|---|---|---|
| L | op0 | op1 | CRn | op2 | Rt | |
| 0 | 00 | 011 | 0011 | 111 | - | Unallocated. |
| 0 | 00 | 1xx | 001x | - | - | Unallocated. |
| 0 | 01 | - | - | - | - | SYS |
| 0 | 1x | - | - | - | - | MSR (register) |
| 1 | 00 | - | - | - | - | Unallocated. |
| 1 | 01 | - | - | - | - | SYSL |
| 1 | 1x | - | - | - | - | MRS |

### C4.3.5 Test & branch (immediate)

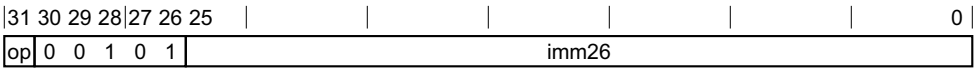
This section describes the encoding of the Test & branch (immediate) instruction class. This section is decoded from *Branches, exception generating and system instructions* on page C4-191.




| Decode fields | Instruction Page |
|---|---|
| op | |
| 0 | TBZ |
| 1 | TBNZ |

### C4.3.6 Unconditional branch (immediate)

This section describes the encoding of the Unconditional branch (immediate) instruction class. This section is decoded from *Branches, exception generating and system instructions* on page C4-191.

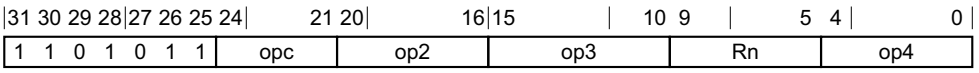



| Decode fields | Instruction Page |
|---|---|
| op | |
| 0 | B |
| 1 | BL |

### C4.3.7 Unconditional branch (register)

This section describes the encoding of the Unconditional branch (register) instruction class. This section is decoded from *Branches, exception generating and system instructions* on page C4-191.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| **opc** | **op2** | **op3** | **Rn** | **op4** | |
| - | - | - | - | != 00000 | Unallocated. |
| - | - | != 000000 | - | - | Unallocated. |
| - | != 11111 | - | - | - | Unallocated. |
| 0000 | 11111 | 000000 | - | 00000 | BR |
| 0001 | 11111 | 000000 | - | 00000 | BLR |
| 0010 | 11111 | 000000 | - | 00000 | RET |
| 0011 | - | - | - | - | Unallocated. |
| 010x | - | - | != 11111 | - | Unallocated. |
| 0100 | 11111 | 000000 | 11111 | 00000 | ERET |
| 0101 | 11111 | 000000 | 11111 | 00000 | DRPS |
| 011x | - | - | - | - | Unallocated. |
| 1xxx | - | - | - | - | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C4.4 Loads and stores

This section describes the encoding of the Loads and stores instruction group. This section is decoded from *A64 instruction index by encoding* on page C4-186. For additional information on this functional group of instructions, see *Loads and stores* on page C3-142.

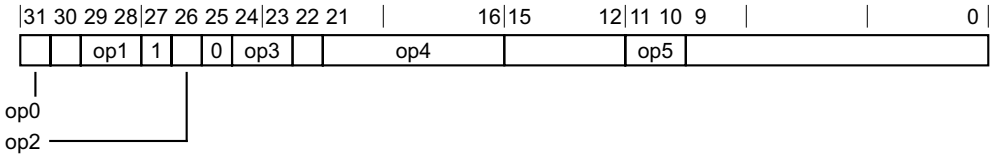



**Table C4-4 Encoding table for the Loads and Stores group**

| Decode fields | | | | | | Decode group or instruction page |
|---|---|---|---|---|---|---|
| op0 | op1 | op2 | op3 | op4 | op5 | |
| 0 | 00 | 1 | 00 | 000000 | - | *Advanced SIMD load/store multiple structures* on page C4-197 |
| 0 | 00 | 1 | 01 | 0xxxxx | - | *Advanced SIMD load/store multiple structures (post-indexed)* on page C4-198 |
| 0 | 00 | 1 | 0x | 1xxxxx | - | Unallocated. |
| 0 | 00 | 1 | 10 | x00000 | - | *Advanced SIMD load/store single structure* on page C4-199 |
| 0 | 00 | 1 | 11 | - | - | *Advanced SIMD load/store single structure (post-indexed)* on page C4-202 |
| 0 | 00 | 1 | x0 | x1xxxx | - | Unallocated. |
| 0 | 00 | 1 | x0 | xx1xxx | - | Unallocated. |
| 0 | 00 | 1 | x0 | xxx1xx | - | Unallocated. |
| 0 | 00 | 1 | x0 | xxxx1x | - | Unallocated. |
| 0 | 00 | 1 | x0 | xxxxx1 | - | Unallocated. |
| 1 | 00 | 1 | - | - | - | Unallocated. |
| - | 00 | 0 | 0x | - | - | *Load/store exclusive* on page C4-206 |
| - | 00 | 0 | 1x | - | - | Unallocated. |
| - | 01 | - | 0x | - | - | *Load register (literal)* on page C4-206 |
| - | 01 | - | 1x | - | - | Unallocated. |
| - | 10 | - | 00 | - | - | *Load/store no-allocate pair (offset)* on page C4-207 |
| - | 10 | - | 01 | - | - | *Load/store register pair (post-indexed)* on page C4-215 |
| - | 10 | - | 10 | - | - | *Load/store register pair (offset)* on page C4-215 |
| - | 10 | - | 11 | - | - | *Load/store register pair (pre-indexed)* on page C4-216 |
| - | 11 | - | 0x | 0xxxxx | 00 | *Load/store register (unscaled immediate)* on page C4-213 |
| - | 11 | - | 0x | 0xxxxx | 01 | *Load/store register (immediate post-indexed)* on page C4-208 |
| - | 11 | - | 0x | 0xxxxx | 10 | *Load/store register (unprivileged)* on page C4-212 |
| - | 11 | - | 0x | 0xxxxx | 11 | *Load/store register (immediate pre-indexed)* on page C4-209 |
| - | 11 | - | 0x | 1xxxxx | 00 | Unallocated. |

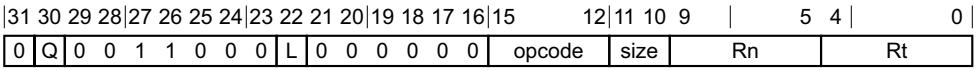

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C4-4 Encoding table for the Loads and Stores group (continued)**

| Decode fields | | | | | | Decode group or instruction page |
|---|---|---|---|---|---|---|
| op0 | op1 | op2 | op3 | op4 | op5 | |
| - | 11 | - | 0x | 1xxxxx | 01 | Unallocated. |
| - | 11 | - | 0x | 1xxxxx | 10 | *Load/store register (register offset)* on page C4-210 |
| - | 11 | - | 0x | 1xxxxx | 11 | Unallocated. |
| - | 11 | - | 1x | - | - | *Load/store register (unsigned immediate)* on page C4-214 |

## C4.4.1 Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. This section is decoded from *Loads and stores* on page C4-196.



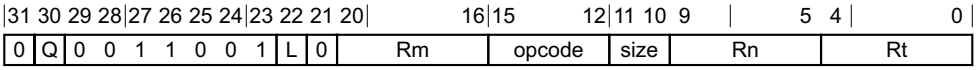

| Decode fields | | Instruction Page |
|---|---|---|
| L | opcode | |
| 0 | 0000 | ST4 (multiple structures) |
| 0 | 0001 | Unallocated. |
| 0 | 0010 | ST1 (multiple structures) - Four registers variant |
| 0 | 0011 | Unallocated. |
| 0 | 0100 | ST3 (multiple structures) |
| 0 | 0101 | Unallocated. |
| 0 | 0110 | ST1 (multiple structures) - Three registers variant |
| 0 | 0111 | ST1 (multiple structures) - One register variant |
| 0 | 1000 | ST2 (multiple structures) |
| 0 | 1001 | Unallocated. |
| 0 | 1010 | ST1 (multiple structures) - Two registers variant |
| 0 | 1011 | Unallocated. |
| 0 | 11xx | Unallocated. |
| 1 | 0000 | LD4 (multiple structures) |
| 1 | 0001 | Unallocated. |
| 1 | 0010 | LD1 (multiple structures) - Four registers variant |
| 1 | 0011 | Unallocated. |
| 1 | 0100 | LD3 (multiple structures) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | Instruction Page |
|---|---|---|
| L | opcode | |
| 1 | 0101 | Unallocated. |
| 1 | 0110 | LD1 (multiple structures) - Three registers variant |
| 1 | 0111 | LD1 (multiple structures) - One register variant |
| 1 | 1000 | LD2 (multiple structures) |
| 1 | 1001 | Unallocated. |
| 1 | 1010 | LD1 (multiple structures) - Two registers variant |
| 1 | 1011 | Unallocated. |
| 1 | 11xx | Unallocated. |

### C4.4.2 Advanced SIMD load/store multiple structures (post-indexed)

This section describes the encoding of the Advanced SIMD load/store multiple structures (post-indexed) instruction class. This section is decoded from *Loads and stores* on page C4-196.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| L | Rm | opcode | |
| 0 | - | 0001 | Unallocated. |
| 0 | - | 0011 | Unallocated. |
| 0 | - | 0101 | Unallocated. |
| 0 | - | 1001 | Unallocated. |
| 0 | - | 1011 | Unallocated. |
| 0 | - | 11xx | Unallocated. |
| 0 | != 11111 | 0000 | ST4 (multiple structures) - Register offset variant |
| 0 | != 11111 | 0010 | ST1 (multiple structures) - Four registers, register offset variant |
| 0 | != 11111 | 0100 | ST3 (multiple structures) - Register offset variant |
| 0 | != 11111 | 0110 | ST1 (multiple structures) - Three registers, register offset variant |
| 0 | != 11111 | 0111 | ST1 (multiple structures) - One register, register offset variant |
| 0 | != 11111 | 1000 | ST2 (multiple structures) - Register offset variant |
| 0 | != 11111 | 1010 | ST1 (multiple structures) - Two registers, register offset variant |
| 0 | 11111 | 0000 | ST4 (multiple structures) - Immediate offset variant |
| 0 | 11111 | 0010 | ST1 (multiple structures) - Four registers, immediate offset variant |

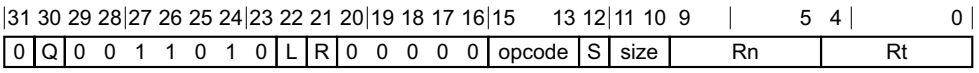

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| L | Rm | opcode | |
| 0 | 11111 | 0100 | ST3 (multiple structures) - Immediate offset variant |
| 0 | 11111 | 0110 | ST1 (multiple structures) - Three registers, immediate offset variant |
| 0 | 11111 | 0111 | ST1 (multiple structures) - One register, immediate offset variant |
| 0 | 11111 | 1000 | ST2 (multiple structures) - Immediate offset variant |
| 0 | 11111 | 1010 | ST1 (multiple structures) - Two registers, immediate offset variant |
| 1 | - | 0001 | Unallocated. |
| 1 | - | 0011 | Unallocated. |
| 1 | - | 0101 | Unallocated. |
| 1 | - | 1001 | Unallocated. |
| 1 | - | 1011 | Unallocated. |
| 1 | - | 11xx | Unallocated. |
| 1 | != 11111 | 0000 | LD4 (multiple structures) - Register offset variant |
| 1 | != 11111 | 0010 | LD1 (multiple structures) - Four registers, register offset variant |
| 1 | != 11111 | 0100 | LD3 (multiple structures) - Register offset variant |
| 1 | != 11111 | 0110 | LD1 (multiple structures) - Three registers, register offset variant |
| 1 | != 11111 | 0111 | LD1 (multiple structures) - One register, register offset variant |
| 1 | != 11111 | 1000 | LD2 (multiple structures) - Register offset variant |
| 1 | != 11111 | 1010 | LD1 (multiple structures) - Two registers, register offset variant |
| 1 | 11111 | 0000 | LD4 (multiple structures) - Immediate offset variant |
| 1 | 11111 | 0010 | LD1 (multiple structures) - Four registers, immediate offset variant |
| 1 | 11111 | 0100 | LD3 (multiple structures) - Immediate offset variant |
| 1 | 11111 | 0110 | LD1 (multiple structures) - Three registers, immediate offset variant |
| 1 | 11111 | 0111 | LD1 (multiple structures) - One register, immediate offset variant |
| 1 | 11111 | 1000 | LD2 (multiple structures) - Immediate offset variant |
| 1 | 11111 | 1010 | LD1 (multiple structures) - Two registers, immediate offset variant |

### C4.4.3 Advanced SIMD load/store single structure

This section describes the encoding of the Advanced SIMD load/store single structure instruction class. This section is decoded from *Loads and stores* on page C4-196.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| L | R | opcode | S | size | |
| 0 | - | 11x | - | - | Unallocated. |
| 0 | 0 | 000 | - | - | ST1 (single structure) - 8-bit variant |
| 0 | 0 | 001 | - | - | ST3 (single structure) - 8-bit variant |
| 0 | 0 | 010 | - | x0 | ST1 (single structure) - 16-bit variant |
| 0 | 0 | 010 | - | x1 | Unallocated. |
| 0 | 0 | 011 | - | x0 | ST3 (single structure) - 16-bit variant |
| 0 | 0 | 011 | - | x1 | Unallocated. |
| 0 | 0 | 100 | - | 00 | ST1 (single structure) - 32-bit variant |
| 0 | 0 | 100 | - | 1x | Unallocated. |
| 0 | 0 | 100 | 0 | 01 | ST1 (single structure) - 64-bit variant |
| 0 | 0 | 100 | 1 | 01 | Unallocated. |
| 0 | 0 | 101 | - | 00 | ST3 (single structure) - 32-bit variant |
| 0 | 0 | 101 | - | 10 | Unallocated. |
| 0 | 0 | 101 | 0 | 01 | ST3 (single structure) - 64-bit variant |
| 0 | 0 | 101 | 0 | 11 | Unallocated. |
| 0 | 0 | 101 | 1 | x1 | Unallocated. |
| 0 | 1 | 000 | - | - | ST2 (single structure) - 8-bit variant |
| 0 | 1 | 001 | - | - | ST4 (single structure) - 8-bit variant |
| 0 | 1 | 010 | - | x0 | ST2 (single structure) - 16-bit variant |
| 0 | 1 | 010 | - | x1 | Unallocated. |
| 0 | 1 | 011 | - | x0 | ST4 (single structure) - 16-bit variant |
| 0 | 1 | 011 | - | x1 | Unallocated. |
| 0 | 1 | 100 | - | 00 | ST2 (single structure) - 32-bit variant |
| 0 | 1 | 100 | - | 10 | Unallocated. |
| 0 | 1 | 100 | 0 | 01 | ST2 (single structure) - 64-bit variant |
| 0 | 1 | 100 | 0 | 11 | Unallocated. |
| 0 | 1 | 100 | 1 | x1 | Unallocated. |
| 0 | 1 | 101 | - | 00 | ST4 (single structure) - 32-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

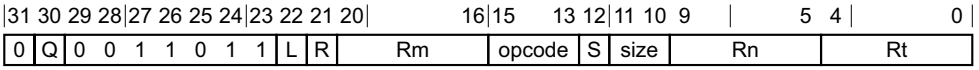
| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| L | R | opcode | S | size | |
| 0 | 1 | 101 | - | 10 | Unallocated. |
| 0 | 1 | 101 | 0 | 01 | ST4 (single structure) - 64-bit variant |
| 0 | 1 | 101 | 0 | 11 | Unallocated. |
| 0 | 1 | 101 | 1 | x1 | Unallocated. |
| 1 | 0 | 000 | - | - | LD1 (single structure) - 8-bit variant |
| 1 | 0 | 001 | - | - | LD3 (single structure) - 8-bit variant |
| 1 | 0 | 010 | - | x0 | LD1 (single structure) - 16-bit variant |
| 1 | 0 | 010 | - | x1 | Unallocated. |
| 1 | 0 | 011 | - | x0 | LD3 (single structure) - 16-bit variant |
| 1 | 0 | 011 | - | x1 | Unallocated. |
| 1 | 0 | 100 | - | 00 | LD1 (single structure) - 32-bit variant |
| 1 | 0 | 100 | - | 1x | Unallocated. |
| 1 | 0 | 100 | 0 | 01 | LD1 (single structure) - 64-bit variant |
| 1 | 0 | 100 | 1 | 01 | Unallocated. |
| 1 | 0 | 101 | - | 00 | LD3 (single structure) - 32-bit variant |
| 1 | 0 | 101 | - | 10 | Unallocated. |
| 1 | 0 | 101 | 0 | 01 | LD3 (single structure) - 64-bit variant |
| 1 | 0 | 101 | 0 | 11 | Unallocated. |
| 1 | 0 | 101 | 1 | x1 | Unallocated. |
| 1 | 0 | 110 | 0 | - | LD1R |
| 1 | 0 | 110 | 1 | - | Unallocated. |
| 1 | 0 | 111 | 0 | - | LD3R |
| 1 | 0 | 111 | 1 | - | Unallocated. |
| 1 | 1 | 000 | - | - | LD2 (single structure) - 8-bit variant |
| 1 | 1 | 001 | - | - | LD4 (single structure) - 8-bit variant |
| 1 | 1 | 010 | - | x0 | LD2 (single structure) - 16-bit variant |
| 1 | 1 | 010 | - | x1 | Unallocated. |
| 1 | 1 | 011 | - | x0 | LD4 (single structure) - 16-bit variant |
| 1 | 1 | 011 | - | x1 | Unallocated. |
| 1 | 1 | 100 | - | 00 | LD2 (single structure) - 32-bit variant |
| 1 | 1 | 100 | - | 10 | Unallocated. |
| 1 | 1 | 100 | 0 | 01 | LD2 (single structure) - 64-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| L | R | opcode | S | size | |
| 1 | 1 | 100 | 0 | 11 | Unallocated. |
| 1 | 1 | 100 | 1 | x1 | Unallocated. |
| 1 | 1 | 101 | - | 00 | LD4 (single structure) - 32-bit variant |
| 1 | 1 | 101 | - | 10 | Unallocated. |
| 1 | 1 | 101 | 0 | 01 | LD4 (single structure) - 64-bit variant |
| 1 | 1 | 101 | 0 | 11 | Unallocated. |
| 1 | 1 | 101 | 1 | x1 | Unallocated. |
| 1 | 1 | 110 | 0 | - | LD2R |
| 1 | 1 | 110 | 1 | - | Unallocated. |
| 1 | 1 | 111 | 0 | - | LD4R |
| 1 | 1 | 111 | 1 | - | Unallocated. |

### C4.4.4 Advanced SIMD load/store single structure (post-indexed)

This section describes the encoding of the Advanced SIMD load/store single structure (post-indexed) instruction class. This section is decoded from *Loads and stores* on page C4-196.




| Decode fields | | | | | | Instruction Page |
|---|---|---|---|---|---|---|
| L | R | Rm | opcode | S | size | |
| 0 | - | - | 11x | - | - | Unallocated. |
| 0 | 0 | - | 010 | - | x1 | Unallocated. |
| 0 | 0 | - | 011 | - | x1 | Unallocated. |
| 0 | 0 | - | 100 | - | 1x | Unallocated. |
| 0 | 0 | - | 100 | 1 | 01 | Unallocated. |
| 0 | 0 | - | 101 | - | 10 | Unallocated. |
| 0 | 0 | - | 101 | 0 | 11 | Unallocated. |
| 0 | 0 | - | 101 | 1 | x1 | Unallocated. |
| 0 | 0 | != 11111 | 000 | - | - | ST1 (single structure) - 8-bit, register offset variant |
| 0 | 0 | != 11111 | 001 | - | - | ST3 (single structure) - 8-bit, register offset variant |
| 0 | 0 | != 11111 | 010 | - | x0 | ST1 (single structure) - 16-bit, register offset variant |
| 0 | 0 | != 11111 | 011 | - | x0 | ST3 (single structure) - 16-bit, register offset variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | | Instruction Page |
|---|---|---|---|---|---|---|
| L | R | Rm | opcode | S | size | |
| 0 | 0 | != 11111 | 100 | - | 00 | ST1 (single structure) - 32-bit, register offset variant |
| 0 | 0 | != 11111 | 100 | 0 | 01 | ST1 (single structure) - 64-bit, register offset variant |
| 0 | 0 | != 11111 | 101 | - | 00 | ST3 (single structure) - 32-bit, register offset variant |
| 0 | 0 | != 11111 | 101 | 0 | 01 | ST3 (single structure) - 64-bit, register offset variant |
| 0 | 0 | 11111 | 000 | - | - | ST1 (single structure) - 8-bit, immediate offset variant |
| 0 | 0 | 11111 | 001 | - | - | ST3 (single structure) - 8-bit, immediate offset variant |
| 0 | 0 | 11111 | 010 | - | x0 | ST1 (single structure) - 16-bit, immediate offset variant |
| 0 | 0 | 11111 | 011 | - | x0 | ST3 (single structure) - 16-bit, immediate offset variant |
| 0 | 0 | 11111 | 100 | - | 00 | ST1 (single structure) - 32-bit, immediate offset variant |
| 0 | 0 | 11111 | 100 | 0 | 01 | ST1 (single structure) - 64-bit, immediate offset variant |
| 0 | 0 | 11111 | 101 | - | 00 | ST3 (single structure) - 32-bit, immediate offset variant |
| 0 | 0 | 11111 | 101 | 0 | 01 | ST3 (single structure) - 64-bit, immediate offset variant |
| 0 | 1 | - | 010 | - | x1 | Unallocated. |
| 0 | 1 | - | 011 | - | x1 | Unallocated. |
| 0 | 1 | - | 100 | - | 10 | Unallocated. |
| 0 | 1 | - | 100 | 0 | 11 | Unallocated. |
| 0 | 1 | - | 100 | 1 | x1 | Unallocated. |
| 0 | 1 | - | 101 | - | 10 | Unallocated. |
| 0 | 1 | - | 101 | 0 | 11 | Unallocated. |
| 0 | 1 | - | 101 | 1 | x1 | Unallocated. |
| 0 | 1 | != 11111 | 000 | - | - | ST2 (single structure) - 8-bit, register offset variant |
| 0 | 1 | != 11111 | 001 | - | - | ST4 (single structure) - 8-bit, register offset variant |
| 0 | 1 | != 11111 | 010 | - | x0 | ST2 (single structure) - 16-bit, register offset variant |
| 0 | 1 | != 11111 | 011 | - | x0 | ST4 (single structure) - 16-bit, register offset variant |
| 0 | 1 | != 11111 | 100 | - | 00 | ST2 (single structure) - 32-bit, register offset variant |
| 0 | 1 | != 11111 | 100 | 0 | 01 | ST2 (single structure) - 64-bit, register offset variant |
| 0 | 1 | != 11111 | 101 | - | 00 | ST4 (single structure) - 32-bit, register offset variant |
| 0 | 1 | != 11111 | 101 | 0 | 01 | ST4 (single structure) - 64-bit, register offset variant |
| 0 | 1 | 11111 | 000 | - | - | ST2 (single structure) - 8-bit, immediate offset variant |
| 0 | 1 | 11111 | 001 | - | - | ST4 (single structure) - 8-bit, immediate offset variant |
| 0 | 1 | 11111 | 010 | - | x0 | ST2 (single structure) - 16-bit, immediate offset variant |
| 0 | 1 | 11111 | 011 | - | x0 | ST4 (single structure) - 16-bit, immediate offset variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | | Instruction Page |
|---|---|---|---|---|---|---|
| L | R | Rm | opcode | S | size | |
| 0 | 1 | 11111 | 100 | - | 00 | ST2 (single structure) - 32-bit, immediate offset variant |
| 0 | 1 | 11111 | 100 | 0 | 01 | ST2 (single structure) - 64-bit, immediate offset variant |
| 0 | 1 | 11111 | 101 | - | 00 | ST4 (single structure) - 32-bit, immediate offset variant |
| 0 | 1 | 11111 | 101 | 0 | 01 | ST4 (single structure) - 64-bit, immediate offset variant |
| 1 | 0 | - | 010 | - | x1 | Unallocated. |
| 1 | 0 | - | 011 | - | x1 | Unallocated. |
| 1 | 0 | - | 100 | - | 1x | Unallocated. |
| 1 | 0 | - | 100 | 1 | 01 | Unallocated. |
| 1 | 0 | - | 101 | - | 10 | Unallocated. |
| 1 | 0 | - | 101 | 0 | 11 | Unallocated. |
| 1 | 0 | - | 101 | 1 | x1 | Unallocated. |
| 1 | 0 | - | 110 | 1 | - | Unallocated. |
| 1 | 0 | - | 111 | 1 | - | Unallocated. |
| 1 | 0 | != 11111 | 000 | - | - | LD1 (single structure) - 8-bit, register offset variant |
| 1 | 0 | != 11111 | 001 | - | - | LD3 (single structure) - 8-bit, register offset variant |
| 1 | 0 | != 11111 | 010 | - | x0 | LD1 (single structure) - 16-bit, register offset variant |
| 1 | 0 | != 11111 | 011 | - | x0 | LD3 (single structure) - 16-bit, register offset variant |
| 1 | 0 | != 11111 | 100 | - | 00 | LD1 (single structure) - 32-bit, register offset variant |
| 1 | 0 | != 11111 | 100 | 0 | 01 | LD1 (single structure) - 64-bit, register offset variant |
| 1 | 0 | != 11111 | 101 | - | 00 | LD3 (single structure) - 32-bit, register offset variant |
| 1 | 0 | != 11111 | 101 | 0 | 01 | LD3 (single structure) - 64-bit, register offset variant |
| 1 | 0 | != 11111 | 110 | 0 | - | LD1R - Register offset variant |
| 1 | 0 | != 11111 | 111 | 0 | - | LD3R - Register offset variant |
| 1 | 0 | 11111 | 000 | - | - | LD1 (single structure) - 8-bit, immediate offset variant |
| 1 | 0 | 11111 | 001 | - | - | LD3 (single structure) - 8-bit, immediate offset variant |
| 1 | 0 | 11111 | 010 | - | x0 | LD1 (single structure) - 16-bit, immediate offset variant |
| 1 | 0 | 11111 | 011 | - | x0 | LD3 (single structure) - 16-bit, immediate offset variant |
| 1 | 0 | 11111 | 100 | - | 00 | LD1 (single structure) - 32-bit, immediate offset variant |
| 1 | 0 | 11111 | 100 | 0 | 01 | LD1 (single structure) - 64-bit, immediate offset variant |
| 1 | 0 | 11111 | 101 | - | 00 | LD3 (single structure) - 32-bit, immediate offset variant |
| 1 | 0 | 11111 | 101 | 0 | 01 | LD3 (single structure) - 64-bit, immediate offset variant |
| 1 | 0 | 11111 | 110 | 0 | - | LD1R - Immediate offset variant |

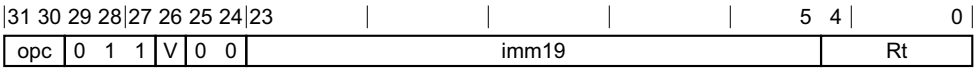

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | | Instruction Page |
|---|---|---|---|---|---|---|
| L | R | Rm | opcode | S | size | |
| 1 | 0 | 11111 | 111 | 0 | - | LD3R - Immediate offset variant |
| 1 | 1 | - | 010 | - | x1 | Unallocated. |
| 1 | 1 | - | 011 | - | x1 | Unallocated. |
| 1 | 1 | - | 100 | - | 10 | Unallocated. |
| 1 | 1 | - | 100 | 0 | 11 | Unallocated. |
| 1 | 1 | - | 100 | 1 | x1 | Unallocated. |
| 1 | 1 | - | 101 | - | 10 | Unallocated. |
| 1 | 1 | - | 101 | 0 | 11 | Unallocated. |
| 1 | 1 | - | 101 | 1 | x1 | Unallocated. |
| 1 | 1 | - | 110 | 1 | - | Unallocated. |
| 1 | 1 | - | 111 | 1 | - | Unallocated. |
| 1 | 1 | != 11111 | 000 | - | - | LD2 (single structure) - 8-bit, register offset variant |
| 1 | 1 | != 11111 | 001 | - | - | LD4 (single structure) - 8-bit, register offset variant |
| 1 | 1 | != 11111 | 010 | - | x0 | LD2 (single structure) - 16-bit, register offset variant |
| 1 | 1 | != 11111 | 011 | - | x0 | LD4 (single structure) - 16-bit, register offset variant |
| 1 | 1 | != 11111 | 100 | - | 00 | LD2 (single structure) - 32-bit, register offset variant |
| 1 | 1 | != 11111 | 100 | 0 | 01 | LD2 (single structure) - 64-bit, register offset variant |
| 1 | 1 | != 11111 | 101 | - | 00 | LD4 (single structure) - 32-bit, register offset variant |
| 1 | 1 | != 11111 | 101 | 0 | 01 | LD4 (single structure) - 64-bit, register offset variant |
| 1 | 1 | != 11111 | 110 | 0 | - | LD2R - Register offset variant |
| 1 | 1 | != 11111 | 111 | 0 | - | LD4R - Register offset variant |
| 1 | 1 | 11111 | 000 | - | - | LD2 (single structure) - 8-bit, immediate offset variant |
| 1 | 1 | 11111 | 001 | - | - | LD4 (single structure) - 8-bit, immediate offset variant |
| 1 | 1 | 11111 | 010 | - | x0 | LD2 (single structure) - 16-bit, immediate offset variant |
| 1 | 1 | 11111 | 011 | - | x0 | LD4 (single structure) - 16-bit, immediate offset variant |
| 1 | 1 | 11111 | 100 | - | 00 | LD2 (single structure) - 32-bit, immediate offset variant |
| 1 | 1 | 11111 | 100 | 0 | 01 | LD2 (single structure) - 64-bit, immediate offset variant |
| 1 | 1 | 11111 | 101 | - | 00 | LD4 (single structure) - 32-bit, immediate offset variant |
| 1 | 1 | 11111 | 101 | 0 | 01 | LD4 (single structure) - 64-bit, immediate offset variant |
| 1 | 1 | 11111 | 110 | 0 | - | LD2R - Immediate offset variant |
| 1 | 1 | 11111 | 111 | 0 | - | LD4R - Immediate offset variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C4.4.5 Load register (literal)

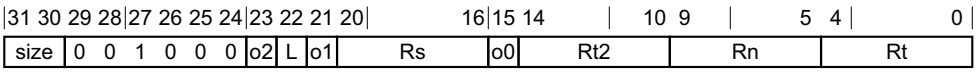
This section describes the encoding of the Load register (literal) instruction class. This section is decoded from *Loads and stores* on page C4-196.




| Decode fields | | Instruction Page |
|---|---|---|
| opc | V | |
| 00 | 0 | LDR (literal) - 32-bit variant |
| 00 | 1 | LDR (literal, SIMD&FP) - 32-bit variant |
| 01 | 0 | LDR (literal) - 64-bit variant |
| 01 | 1 | LDR (literal, SIMD&FP) - 64-bit variant |
| 10 | 0 | LDRSW (literal) |
| 10 | 1 | LDR (literal, SIMD&FP) - 128-bit variant |
| 11 | 0 | PRFM (literal) |
| 11 | 1 | Unallocated. |

### C4.4.6 Load/store exclusive

This section describes the encoding of the Load/store exclusive instruction class. This section is decoded from *Loads and stores* on page C4-196.




| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| size | o2 | L | o1 | o0 | |
| - | 1 | - | 0 | 0 | Unallocated. |
| - | 1 | - | 1 | - | Unallocated. |
| 0x | - | - | 1 | - | Unallocated. |
| 00 | 0 | 0 | 0 | 0 | STXRB |
| 00 | 0 | 0 | 0 | 1 | STLXRB |
| 00 | 0 | 1 | 0 | 0 | LDXRB |
| 00 | 0 | 1 | 0 | 1 | LDAXRB |
| 00 | 1 | 0 | 0 | 1 | STLRB |
| 00 | 1 | 1 | 0 | 1 | LDARB |

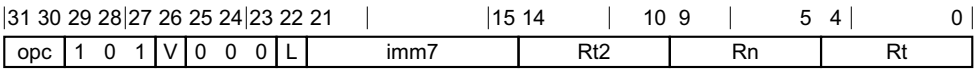

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| size | o2 | L | o1 | o0 | |
| 01 | 0 | 0 | 0 | 0 | STXRH |
| 01 | 0 | 0 | 0 | 1 | STLXRH |
| 01 | 0 | 1 | 0 | 0 | LDXRH |
| 01 | 0 | 1 | 0 | 1 | LDAXRH |
| 01 | 1 | 0 | 0 | 1 | STLRH |
| 01 | 1 | 1 | 0 | 1 | LDARH |
| 10 | 0 | 0 | 0 | 0 | STXR - 32-bit variant |
| 10 | 0 | 0 | 0 | 1 | STLXR - 32-bit variant |
| 10 | 0 | 0 | 1 | 0 | STXP - 32-bit variant |
| 10 | 0 | 0 | 1 | 1 | STLXP - 32-bit variant |
| 10 | 0 | 1 | 0 | 0 | LDXR - 32-bit variant |
| 10 | 0 | 1 | 0 | 1 | LDAXR - 32-bit variant |
| 10 | 0 | 1 | 1 | 0 | LDXP - 32-bit variant |
| 10 | 0 | 1 | 1 | 1 | LDAXP - 32-bit variant |
| 10 | 1 | 0 | 0 | 1 | STLR - 32-bit variant |
| 10 | 1 | 1 | 0 | 1 | LDAR - 32-bit variant |
| 11 | 0 | 0 | 0 | 0 | STXR - 64-bit variant |
| 11 | 0 | 0 | 0 | 1 | STLXR - 64-bit variant |
| 11 | 0 | 0 | 1 | 0 | STXP - 64-bit variant |
| 11 | 0 | 0 | 1 | 1 | STLXP - 64-bit variant |
| 11 | 0 | 1 | 0 | 0 | LDXR - 64-bit variant |
| 11 | 0 | 1 | 0 | 1 | LDAXR - 64-bit variant |
| 11 | 0 | 1 | 1 | 0 | LDXP - 64-bit variant |
| 11 | 0 | 1 | 1 | 1 | LDAXP - 64-bit variant |
| 11 | 1 | 0 | 0 | 1 | STLR - 64-bit variant |
| 11 | 1 | 1 | 0 | 1 | LDAR - 64-bit variant |

### C4.4.7 Load/store no-allocate pair (offset)

This section describes the encoding of the Load/store no-allocate pair (offset) instruction class. This section is decoded from *Loads and stores* on page C4-196.

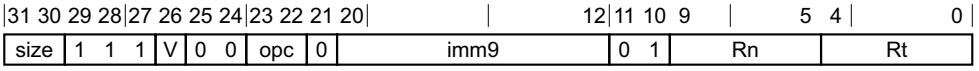

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| opc | V | L | |
| 00 | 0 | 0 | STNP - 32-bit variant |
| 00 | 0 | 1 | LDNP - 32-bit variant |
| 00 | 1 | 0 | STNP (SIMD&FP) - 32-bit variant |
| 00 | 1 | 1 | LDNP (SIMD&FP) - 32-bit variant |
| 01 | 0 | - | Unallocated. |
| 01 | 1 | 0 | STNP (SIMD&FP) - 64-bit variant |
| 01 | 1 | 1 | LDNP (SIMD&FP) - 64-bit variant |
| 10 | 0 | 0 | STNP - 64-bit variant |
| 10 | 0 | 1 | LDNP - 64-bit variant |
| 10 | 1 | 0 | STNP (SIMD&FP) - 128-bit variant |
| 10 | 1 | 1 | LDNP (SIMD&FP) - 128-bit variant |
| 11 | - | - | Unallocated. |

### C4.4.8 Load/store register (immediate post-indexed)

This section describes the encoding of the Load/store register (immediate post-indexed) instruction class. This section is decoded from *Loads and stores* on page C4-196.



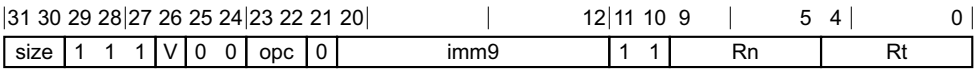

| Decode fields | | | Instruction Page |
|---|---|---|---|
| size | V | opc | |
| x1 | 1 | 1x | Unallocated. |
| 00 | 0 | 00 | STRB (immediate) |
| 00 | 0 | 01 | LDRB (immediate) |
| 00 | 0 | 10 | LDRSB (immediate) - 64-bit variant |
| 00 | 0 | 11 | LDRSB (immediate) - 32-bit variant |
| 00 | 1 | 00 | STR (immediate, SIMD&FP) - 8-bit variant |
| 00 | 1 | 01 | LDR (immediate, SIMD&FP) - 8-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| size | V | opc | |
| 00 | 1 | 10 | STR (immediate, SIMD&FP) - 128-bit variant |
| 00 | 1 | 11 | LDR (immediate, SIMD&FP) - 128-bit variant |
| 01 | 0 | 00 | STRH (immediate) |
| 01 | 0 | 01 | LDRH (immediate) |
| 01 | 0 | 10 | LDRSH (immediate) - 64-bit variant |
| 01 | 0 | 11 | LDRSH (immediate) - 32-bit variant |
| 01 | 1 | 00 | STR (immediate, SIMD&FP) - 16-bit variant |
| 01 | 1 | 01 | LDR (immediate, SIMD&FP) - 16-bit variant |
| 1x | 0 | 11 | Unallocated. |
| 1x | 1 | 1x | Unallocated. |
| 10 | 0 | 00 | STR (immediate) - 32-bit variant |
| 10 | 0 | 01 | LDR (immediate) - 32-bit variant |
| 10 | 0 | 10 | LDRSW (immediate) |
| 10 | 1 | 00 | STR (immediate, SIMD&FP) - 32-bit variant |
| 10 | 1 | 01 | LDR (immediate, SIMD&FP) - 32-bit variant |
| 11 | 0 | 00 | STR (immediate) - 64-bit variant |
| 11 | 0 | 01 | LDR (immediate) - 64-bit variant |
| 11 | 0 | 10 | Unallocated. |
| 11 | 1 | 00 | STR (immediate, SIMD&FP) - 64-bit variant |
| 11 | 1 | 01 | LDR (immediate, SIMD&FP) - 64-bit variant |

### C4.4.9 Load/store register (immediate pre-indexed)

This section describes the encoding of the Load/store register (immediate pre-indexed) instruction class. This section is decoded from *Loads and stores* on page C4-196.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| size | V | opc | |
| x1 | 1 | 1x | Unallocated. |
| 00 | 0 | 00 | STRB (immediate) |
| 00 | 0 | 01 | LDRB (immediate) |

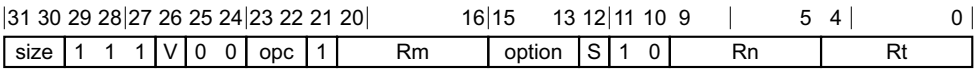

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| size | V | opc | |
| 00 | 0 | 10 | LDRSB (immediate) - 64-bit variant |
| 00 | 0 | 11 | LDRSB (immediate) - 32-bit variant |
| 00 | 1 | 00 | STR (immediate, SIMD&FP) - 8-bit variant |
| 00 | 1 | 01 | LDR (immediate, SIMD&FP) - 8-bit variant |
| 00 | 1 | 10 | STR (immediate, SIMD&FP) - 128-bit variant |
| 00 | 1 | 11 | LDR (immediate, SIMD&FP) - 128-bit variant |
| 01 | 0 | 00 | STRH (immediate) |
| 01 | 0 | 01 | LDRH (immediate) |
| 01 | 0 | 10 | LDRSH (immediate) - 64-bit variant |
| 01 | 0 | 11 | LDRSH (immediate) - 32-bit variant |
| 01 | 1 | 00 | STR (immediate, SIMD&FP) - 16-bit variant |
| 01 | 1 | 01 | LDR (immediate, SIMD&FP) - 16-bit variant |
| 1x | 0 | 11 | Unallocated. |
| 1x | 1 | 1x | Unallocated. |
| 10 | 0 | 00 | STR (immediate) - 32-bit variant |
| 10 | 0 | 01 | LDR (immediate) - 32-bit variant |
| 10 | 0 | 10 | LDRSW (immediate) |
| 10 | 1 | 00 | STR (immediate, SIMD&FP) - 32-bit variant |
| 10 | 1 | 01 | LDR (immediate, SIMD&FP) - 32-bit variant |
| 11 | 0 | 00 | STR (immediate) - 64-bit variant |
| 11 | 0 | 01 | LDR (immediate) - 64-bit variant |
| 11 | 0 | 10 | Unallocated. |
| 11 | 1 | 00 | STR (immediate, SIMD&FP) - 64-bit variant |
| 11 | 1 | 01 | LDR (immediate, SIMD&FP) - 64-bit variant |

### C4.4.10 Load/store register (register offset)

This section describes the encoding of the Load/store register (register offset) instruction class. This section is decoded from *Loads and stores* on page C4-196.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



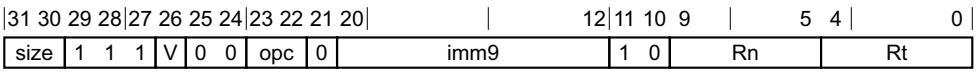

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| size | V | opc | option | |
| - | - | - | x0x | Unallocated. |
| x1 | 1 | 1x | - | Unallocated. |
| 00 | 0 | 00 | != 011 | STRB (register) - Extended register variant |
| 00 | 0 | 00 | 011 | STRB (register) - Shifted register variant |
| 00 | 0 | 01 | != 011 | LDRB (register) - Extended register variant |
| 00 | 0 | 01 | 011 | LDRB (register) - Shifted register variant |
| 00 | 0 | 10 | != 011 | LDRSB (register) - 64-bit with extended register offset variant |
| 00 | 0 | 10 | 011 | LDRSB (register) - 64-bit with shifted register offset variant |
| 00 | 0 | 11 | != 011 | LDRSB (register) - 32-bit with extended register offset variant |
| 00 | 0 | 11 | 011 | LDRSB (register) - 32-bit with shifted register offset variant |
| 00 | 1 | 00 | != 011 | STR (register, SIMD&FP) |
| 00 | 1 | 00 | 011 | STR (register, SIMD&FP) |
| 00 | 1 | 01 | != 011 | LDR (register, SIMD&FP) |
| 00 | 1 | 01 | 011 | LDR (register, SIMD&FP) |
| 00 | 1 | 10 | - | STR (register, SIMD&FP) |
| 00 | 1 | 11 | - | LDR (register, SIMD&FP) |
| 01 | 0 | 00 | - | STRH (register) |
| 01 | 0 | 01 | - | LDRH (register) |
| 01 | 0 | 10 | - | LDRSH (register) - 64-bit variant |
| 01 | 0 | 11 | - | LDRSH (register) - 32-bit variant |
| 01 | 1 | 00 | - | STR (register, SIMD&FP) |
| 01 | 1 | 01 | - | LDR (register, SIMD&FP) |
| 1x | 0 | 11 | - | Unallocated. |
| 1x | 1 | 1x | - | Unallocated. |
| 10 | 0 | 00 | - | STR (register) - 32-bit variant |
| 10 | 0 | 01 | - | LDR (register) - 32-bit variant |
| 10 | 0 | 10 | - | LDRSW (register) |
| 10 | 1 | 00 | - | STR (register, SIMD&FP) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| size | V | opc | option | |
| 10 | 1 | 01 | - | LDR (register, SIMD&FP) |
| 11 | 0 | 00 | - | STR (register) - 64-bit variant |
| 11 | 0 | 01 | - | LDR (register) - 64-bit variant |
| 11 | 0 | 10 | - | PRFM (register) |
| 11 | 1 | 00 | - | STR (register, SIMD&FP) |
| 11 | 1 | 01 | - | LDR (register, SIMD&FP) |

### C4.4.11 Load/store register (unprivileged)

This section describes the encoding of the Load/store register (unprivileged) instruction class. This section is decoded from *Loads and stores* on page C4-196.

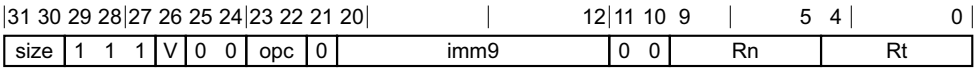



| Decode fields | | | Instruction Page |
|---|---|---|---|
| size | V | opc | |
| - | 1 | - | Unallocated. |
| 00 | 0 | 00 | STTRB |
| 00 | 0 | 01 | LDTRB |
| 00 | 0 | 10 | LDTRSB - 64-bit variant |
| 00 | 0 | 11 | LDTRSB - 32-bit variant |
| 01 | 0 | 00 | STTRH |
| 01 | 0 | 01 | LDTRH |
| 01 | 0 | 10 | LDTRSH - 64-bit variant |
| 01 | 0 | 11 | LDTRSH - 32-bit variant |
| 1x | 0 | 11 | Unallocated. |
| 10 | 0 | 00 | STTR - 32-bit variant |
| 10 | 0 | 01 | LDTR - 32-bit variant |
| 10 | 0 | 10 | LDTRSW |
| 11 | 0 | 00 | STTR - 64-bit variant |
| 11 | 0 | 01 | LDTR - 64-bit variant |
| 11 | 0 | 10 | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C4.4.12 Load/store register (unscaled immediate)

This section describes the encoding of the Load/store register (unscaled immediate) instruction class. This section is decoded from *Loads and stores* on page C4-196.



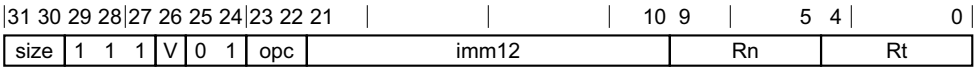

| Decode fields | | | Instruction Page |
|---|---|---|---|
| size | V | opc | |
| x1 | 1 | 1x | Unallocated. |
| 00 | 0 | 00 | STURB |
| 00 | 0 | 01 | LDURB |
| 00 | 0 | 10 | LDURSB - 64-bit variant |
| 00 | 0 | 11 | LDURSB - 32-bit variant |
| 00 | 1 | 00 | STUR (SIMD&FP) - 8-bit variant |
| 00 | 1 | 01 | LDUR (SIMD&FP) - 8-bit variant |
| 00 | 1 | 10 | STUR (SIMD&FP) - 128-bit variant |
| 00 | 1 | 11 | LDUR (SIMD&FP) - 128-bit variant |
| 01 | 0 | 00 | STURH |
| 01 | 0 | 01 | LDURH |
| 01 | 0 | 10 | LDURSH - 64-bit variant |
| 01 | 0 | 11 | LDURSH - 32-bit variant |
| 01 | 1 | 00 | STUR (SIMD&FP) - 16-bit variant |
| 01 | 1 | 01 | LDUR (SIMD&FP) - 16-bit variant |
| 1x | 0 | 11 | Unallocated. |
| 1x | 1 | 1x | Unallocated. |
| 10 | 0 | 00 | STUR - 32-bit variant |
| 10 | 0 | 01 | LDUR - 32-bit variant |
| 10 | 0 | 10 | LDURSW |
| 10 | 1 | 00 | STUR (SIMD&FP) - 32-bit variant |
| 10 | 1 | 01 | LDUR (SIMD&FP) - 32-bit variant |
| 11 | 0 | 00 | STUR - 64-bit variant |
| 11 | 0 | 01 | LDUR - 64-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| size | V | opc | |
| 11 | 0 | 10 | PRFM (unscaled offset) |
| 11 | 1 | 00 | STUR (SIMD&FP) - 64-bit variant |
| 11 | 1 | 01 | LDUR (SIMD&FP) - 64-bit variant |

### C4.4.13 Load/store register (unsigned immediate)

This section describes the encoding of the Load/store register (unsigned immediate) instruction class. This section is decoded from *Loads and stores* on page C4-196.



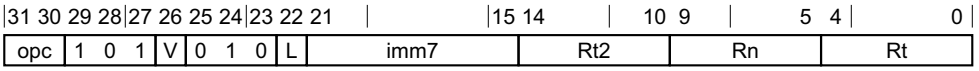

| Decode fields | | | Instruction Page |
|---|---|---|---|
| size | V | opc | |
| x1 | 1 | 1x | Unallocated. |
| 00 | 0 | 00 | STRB (immediate) |
| 00 | 0 | 01 | LDRB (immediate) |
| 00 | 0 | 10 | LDRSB (immediate) - 64-bit variant |
| 00 | 0 | 11 | LDRSB (immediate) - 32-bit variant |
| 00 | 1 | 00 | STR (immediate, SIMD&FP) - 8-bit variant |
| 00 | 1 | 01 | LDR (immediate, SIMD&FP) - 8-bit variant |
| 00 | 1 | 10 | STR (immediate, SIMD&FP) - 128-bit variant |
| 00 | 1 | 11 | LDR (immediate, SIMD&FP) - 128-bit variant |
| 01 | 0 | 00 | STRH (immediate) |
| 01 | 0 | 01 | LDRH (immediate) |
| 01 | 0 | 10 | LDRSH (immediate) - 64-bit variant |
| 01 | 0 | 11 | LDRSH (immediate) - 32-bit variant |
| 01 | 1 | 00 | STR (immediate, SIMD&FP) - 16-bit variant |
| 01 | 1 | 01 | LDR (immediate, SIMD&FP) - 16-bit variant |
| 1x | 0 | 11 | Unallocated. |
| 1x | 1 | 1x | Unallocated. |
| 10 | 0 | 00 | STR (immediate) - 32-bit variant |
| 10 | 0 | 01 | LDR (immediate) - 32-bit variant |
| 10 | 0 | 10 | LDRSW (immediate) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| size | V | opc | |
| 10 | 1 | 00 | STR (immediate, SIMD&FP) - 32-bit variant |
| 10 | 1 | 01 | LDR (immediate, SIMD&FP) - 32-bit variant |
| 11 | 0 | 00 | STR (immediate) - 64-bit variant |
| 11 | 0 | 01 | LDR (immediate) - 64-bit variant |
| 11 | 0 | 10 | PRFM (immediate) |
| 11 | 1 | 00 | STR (immediate, SIMD&FP) - 64-bit variant |
| 11 | 1 | 01 | LDR (immediate, SIMD&FP) - 64-bit variant |

### C4.4.14 Load/store register pair (offset)

This section describes the encoding of the Load/store register pair (offset) instruction class. This section is decoded from *Loads and stores* on page C4-196.

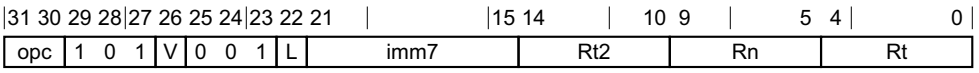



| Decode fields | | | Instruction Page |
|---|---|---|---|
| opc | V | L | |
| 00 | 0 | 0 | STP - 32-bit variant |
| 00 | 0 | 1 | LDP - 32-bit variant |
| 00 | 1 | 0 | STP (SIMD&FP) - 32-bit variant |
| 00 | 1 | 1 | LDP (SIMD&FP) - 32-bit variant |
| 01 | 0 | 0 | Unallocated. |
| 01 | 0 | 1 | LDPSW |
| 01 | 1 | 0 | STP (SIMD&FP) - 64-bit variant |
| 01 | 1 | 1 | LDP (SIMD&FP) - 64-bit variant |
| 10 | 0 | 0 | STP - 64-bit variant |
| 10 | 0 | 1 | LDP - 64-bit variant |
| 10 | 1 | 0 | STP (SIMD&FP) - 128-bit variant |
| 10 | 1 | 1 | LDP (SIMD&FP) - 128-bit variant |
| 11 | - | - | Unallocated. |

### C4.4.15 Load/store register pair (post-indexed)

This section describes the encoding of the Load/store register pair (post-indexed) instruction class. This section is decoded from *Loads and stores* on page C4-196.

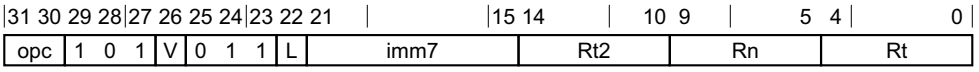

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| opc | V | L | |
| 00 | 0 | 0 | STP - 32-bit variant |
| 00 | 0 | 1 | LDP - 32-bit variant |
| 00 | 1 | 0 | STP (SIMD&FP) - 32-bit variant |
| 00 | 1 | 1 | LDP (SIMD&FP) - 32-bit variant |
| 01 | 0 | 0 | Unallocated. |
| 01 | 0 | 1 | LDPSW |
| 01 | 1 | 0 | STP (SIMD&FP) - 64-bit variant |
| 01 | 1 | 1 | LDP (SIMD&FP) - 64-bit variant |
| 10 | 0 | 0 | STP - 64-bit variant |
| 10 | 0 | 1 | LDP - 64-bit variant |
| 10 | 1 | 0 | STP (SIMD&FP) - 128-bit variant |
| 10 | 1 | 1 | LDP (SIMD&FP) - 128-bit variant |
| 11 | - | - | Unallocated. |

### C4.4.16 Load/store register pair (pre-indexed)

This section describes the encoding of the Load/store register pair (pre-indexed) instruction class. This section is decoded from *Loads and stores* on page C4-196.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| opc | V | L | |
| 00 | 0 | 0 | STP - 32-bit variant |
| 00 | 0 | 1 | LDP - 32-bit variant |
| 00 | 1 | 0 | STP (SIMD&FP) - 32-bit variant |
| 00 | 1 | 1 | LDP (SIMD&FP) - 32-bit variant |
| 01 | 0 | 0 | Unallocated. |
| 01 | 0 | 1 | LDPSW |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| opc | V | L | |
| 01 | 1 | 0 | STP (SIMD&FP) - 64-bit variant |
| 01 | 1 | 1 | LDP (SIMD&FP) - 64-bit variant |
| 10 | 0 | 0 | STP - 64-bit variant |
| 10 | 0 | 1 | LDP - 64-bit variant |
| 10 | 1 | 0 | STP (SIMD&FP) - 128-bit variant |
| 10 | 1 | 1 | LDP (SIMD&FP) - 128-bit variant |
| 11 | - | - | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C4.5 Data processing - register

This section describes the encoding of the Data processing (register) instruction group. This section is decoded from *A64 instruction index by encoding* on page C4-186. For additional information on this functional group of instructions, see *Data processing - register* on page C3-158.

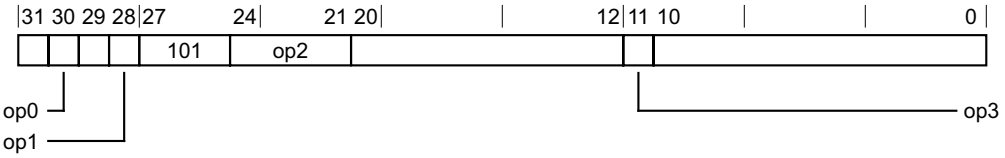
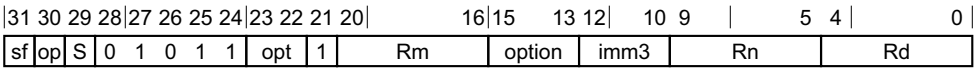



**Table C4-5 Encoding table for the Data Processing -- Register group**

| Decode fields | | | | Decode group or instruction page |
|---|---|---|---|---|
| op0 | op1 | op2 | op3 | |
| 0 | 1 | 0110 | - | *Data-processing (2 source)* on page C4-223 |
| 1 | 1 | 0110 | - | *Data-processing (1 source)* on page C4-222 |
| - | 0 | 0xxx | - | *Logical (shifted register)* on page C4-225 |
| - | 0 | 1xx0 | - | *Add/subtract (shifted register)* on page C4-219 |
| - | 0 | 1xx1 | - | *Add/subtract (extended register)* |
| - | 1 | 0000 | - | *Add/subtract (with carry)* on page C4-219 |
| - | 1 | 0010 | 0 | *Conditional compare (register)* on page C4-221 |
| - | 1 | 0010 | 1 | *Conditional compare (immediate)* on page C4-220 |
| - | 1 | 0100 | - | *Conditional select* on page C4-221 |
| - | 1 | 0xx1 | - | Unallocated. |
| - | 1 | 1xxx | - | *Data-processing (3 source)* on page C4-224 |

### C4.5.1 Add/subtract (extended register)

This section describes the encoding of the Add/subtract (extended register) instruction class. This section is decoded from *Data processing - register*.



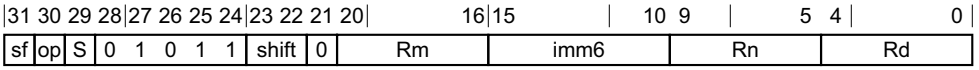

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| sf | op | S | opt | imm3 | |
| - | - | - | - | 1x1 | Unallocated. |
| - | - | - | - | 11x | Unallocated. |
| - | - | - | x1 | - | Unallocated. |
| - | - | - | 1x | - | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| sf | op | S | opt | imm3 | |
| 0 | 0 | 0 | 00 | - | ADD (extended register) - 32-bit variant |
| 0 | 0 | 1 | 00 | - | ADDS (extended register) - 32-bit variant |
| 0 | 1 | 0 | 00 | - | SUB (extended register) - 32-bit variant |
| 0 | 1 | 1 | 00 | - | SUBS (extended register) - 32-bit variant |
| 1 | 0 | 0 | 00 | - | ADD (extended register) - 64-bit variant |
| 1 | 0 | 1 | 00 | - | ADDS (extended register) - 64-bit variant |
| 1 | 1 | 0 | 00 | - | SUB (extended register) - 64-bit variant |
| 1 | 1 | 1 | 00 | - | SUBS (extended register) - 64-bit variant |

### C4.5.2 Add/subtract (shifted register)

This section describes the encoding of the Add/subtract (shifted register) instruction class. This section is decoded from *Data processing - register* on page C4-218.

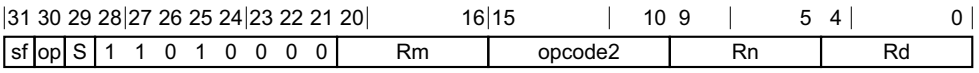



| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| sf | op | S | shift | imm6 | |
| - | - | - | 11 | - | Unallocated. |
| 0 | - | - | - | 1xxxxx | Unallocated. |
| 0 | 0 | 0 | - | - | ADD (shifted register) - 32-bit variant |
| 0 | 0 | 1 | - | - | ADDS (shifted register) - 32-bit variant |
| 0 | 1 | 0 | - | - | SUB (shifted register) - 32-bit variant |
| 0 | 1 | 1 | - | - | SUBS (shifted register) - 32-bit variant |
| 1 | 0 | 0 | - | - | ADD (shifted register) - 64-bit variant |
| 1 | 0 | 1 | - | - | ADDS (shifted register) - 64-bit variant |
| 1 | 1 | 0 | - | - | SUB (shifted register) - 64-bit variant |
| 1 | 1 | 1 | - | - | SUBS (shifted register) - 64-bit variant |

### C4.5.3 Add/subtract (with carry)

This section describes the encoding of the Add/subtract (with carry) instruction class. This section is decoded from *Data processing - register* on page C4-218.

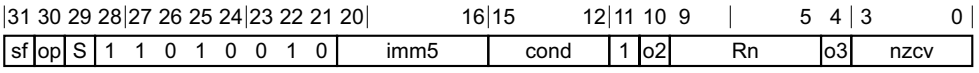

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| sf | op | S | opcode2 | |
| - | - | - | xxxxx1 | Unallocated. |
| - | - | - | xxxx1x | Unallocated. |
| - | - | - | xxx1xx | Unallocated. |
| - | - | - | xx1xxx | Unallocated. |
| - | - | - | x1xxxx | Unallocated. |
| - | - | - | 1xxxxx | Unallocated. |
| 0 | 0 | 0 | 000000 | ADC - 32-bit variant |
| 0 | 0 | 1 | 000000 | ADCS - 32-bit variant |
| 0 | 1 | 0 | 000000 | SBC - 32-bit variant |
| 0 | 1 | 1 | 000000 | SBCS - 32-bit variant |
| 1 | 0 | 0 | 000000 | ADC - 64-bit variant |
| 1 | 0 | 1 | 000000 | ADCS - 64-bit variant |
| 1 | 1 | 0 | 000000 | SBC - 64-bit variant |
| 1 | 1 | 1 | 000000 | SBCS - 64-bit variant |

### C4.5.4 Conditional compare (immediate)

This section describes the encoding of the Conditional compare (immediate) instruction class. This section is decoded from *Data processing - register* on page C4-218.



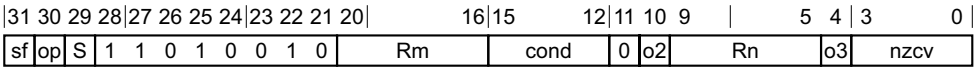

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| sf | op | S | o2 | o3 | |
| - | - | - | - | 1 | Unallocated. |
| - | - | - | 1 | - | Unallocated. |
| - | - | 0 | - | - | Unallocated. |
| 0 | 0 | 1 | 0 | 0 | CCMN (immediate) - 32-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| sf | op | S | o2 | o3 | |
| 0 | 1 | 1 | 0 | 0 | CCMP (immediate) - 32-bit variant |
| 1 | 0 | 1 | 0 | 0 | CCMN (immediate) - 64-bit variant |
| 1 | 1 | 1 | 0 | 0 | CCMP (immediate) - 64-bit variant |

### C4.5.5 Conditional compare (register)

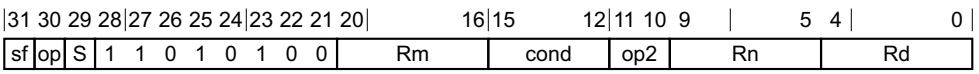
This section describes the encoding of the Conditional compare (register) instruction class. This section is decoded from *Data processing - register* on page C4-218.




| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| sf | op | S | o2 | o3 | |
| - | - | - | - | 1 | Unallocated. |
| - | - | - | 1 | - | Unallocated. |
| - | - | 0 | - | - | Unallocated. |
| 0 | 0 | 1 | 0 | 0 | CCMN (register) - 32-bit variant |
| 0 | 1 | 1 | 0 | 0 | CCMP (register) - 32-bit variant |
| 1 | 0 | 1 | 0 | 0 | CCMN (register) - 64-bit variant |
| 1 | 1 | 1 | 0 | 0 | CCMP (register) - 64-bit variant |

### C4.5.6 Conditional select

This section describes the encoding of the Conditional select instruction class. This section is decoded from *Data processing - register* on page C4-218.



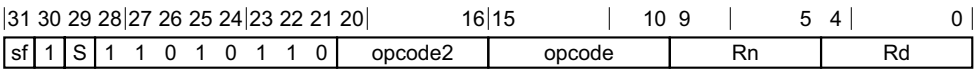

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| sf | op | S | op2 | |
| - | - | - | 1x | Unallocated. |
| - | - | 1 | - | Unallocated. |
| 0 | 0 | 0 | 00 | CSEL - 32-bit variant |
| 0 | 0 | 0 | 01 | CSINC - 32-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| sf | op | S | op2 | |
| 0 | 1 | 0 | 00 | CSINV - 32-bit variant |
| 0 | 1 | 0 | 01 | CSNEG - 32-bit variant |
| 1 | 0 | 0 | 00 | CSEL - 64-bit variant |
| 1 | 0 | 0 | 01 | CSINC - 64-bit variant |
| 1 | 1 | 0 | 00 | CSINV - 64-bit variant |
| 1 | 1 | 0 | 01 | CSNEG - 64-bit variant |

### C4.5.7 Data-processing (1 source)

This section describes the encoding of the Data-processing (1 source) instruction class. This section is decoded from *Data processing - register* on page C4-218.



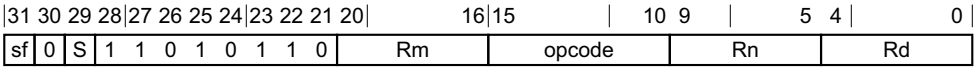

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| sf | S | opcode2 | opcode | |
| - | - | - | xx1xxx | Unallocated. |
| - | - | - | x1xxxx | Unallocated. |
| - | - | - | 1xxxxx | Unallocated. |
| - | - | xxxx1 | - | Unallocated. |
| - | - | xxx1x | - | Unallocated. |
| - | - | xx1xx | - | Unallocated. |
| - | - | x1xxx | - | Unallocated. |
| - | - | 1xxxx | - | Unallocated. |
| - | 0 | 00000 | 00011x | Unallocated. |
| - | 1 | - | - | Unallocated. |
| 0 | 0 | 00000 | 000000 | RBIT - 32-bit variant |
| 0 | 0 | 00000 | 000001 | REV16 - 32-bit variant |
| 0 | 0 | 00000 | 000010 | REV - 32-bit variant |
| 0 | 0 | 00000 | 000011 | Unallocated. |
| 0 | 0 | 00000 | 000100 | CLZ - 32-bit variant |
| 0 | 0 | 00000 | 000101 | CLS - 32-bit variant |
| 1 | 0 | 00000 | 000000 | RBIT - 64-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| sf | S | opcode2 | opcode | |
| 1 | 0 | 00000 | 000001 | REV16 - 64-bit variant |
| 1 | 0 | 00000 | 000010 | REV32 |
| 1 | 0 | 00000 | 000011 | REV - 64-bit variant |
| 1 | 0 | 00000 | 000100 | CLZ - 64-bit variant |
| 1 | 0 | 00000 | 000101 | CLS - 64-bit variant |

### C4.5.8 Data-processing (2 source)

This section describes the encoding of the Data-processing (2 source) instruction class. This section is decoded from *Data processing - register* on page C4-218.



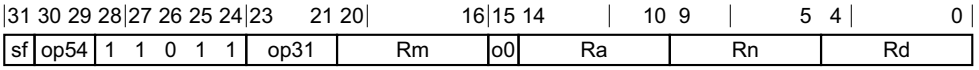

| Decode fields | | | Instruction Page |
|---|---|---|---|
| sf | S | opcode | |
| - | - | 00000x | Unallocated. |
| - | - | 011xxx | Unallocated. |
| - | - | 1xxxxx | Unallocated. |
| - | 0 | 0001xx | Unallocated. |
| - | 0 | 0011xx | Unallocated. |
| - | 1 | - | Unallocated. |
| 0 | 0 | 000010 | UDIV - 32-bit variant |
| 0 | 0 | 000011 | SDIV - 32-bit variant |
| 0 | 0 | 001000 | LSLV - 32-bit variant |
| 0 | 0 | 001001 | LSRV - 32-bit variant |
| 0 | 0 | 001010 | ASRV - 32-bit variant |
| 0 | 0 | 001011 | RORV - 32-bit variant |
| 0 | 0 | 010x11 | Unallocated. |
| 0 | 0 | 010000 | CRC32B, CRC32H, CRC32W, CRC32X - CRC32B variant |
| 0 | 0 | 010001 | CRC32B, CRC32H, CRC32W, CRC32X - CRC32H variant |
| 0 | 0 | 010010 | CRC32B, CRC32H, CRC32W, CRC32X - CRC32W variant |
| 0 | 0 | 010100 | CRC32CB, CRC32CH, CRC32CW, CRC32CX - CRC32CB variant |
| 0 | 0 | 010101 | CRC32CB, CRC32CH, CRC32CW, CRC32CX - CRC32CH variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| sf | S | opcode | |
| 0 | 0 | 010110 | CRC32CB, CRC32CH, CRC32CW, CRC32CX - CRC32CW variant |
| 1 | 0 | 000010 | UDIV - 64-bit variant |
| 1 | 0 | 000011 | SDIV - 64-bit variant |
| 1 | 0 | 001000 | LSLV - 64-bit variant |
| 1 | 0 | 001001 | LSRV - 64-bit variant |
| 1 | 0 | 001010 | ASRV - 64-bit variant |
| 1 | 0 | 001011 | RORV - 64-bit variant |
| 1 | 0 | 010xx0 | Unallocated. |
| 1 | 0 | 010x0x | Unallocated. |
| 1 | 0 | 010011 | CRC32B, CRC32H, CRC32W, CRC32X - CRC32X variant |
| 1 | 0 | 010111 | CRC32CB, CRC32CH, CRC32CW, CRC32CX - CRC32CX variant |

### C4.5.9 Data-processing (3 source)

This section describes the encoding of the Data-processing (3 source) instruction class. This section is decoded from *Data processing - register* on page C4-218.



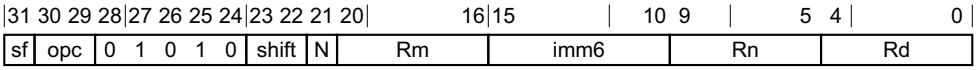

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| sf | op54 | op31 | o0 | |
| - | 00 | 010 | 1 | Unallocated. |
| - | 00 | 011 | - | Unallocated. |
| - | 00 | 100 | - | Unallocated. |
| - | 00 | 110 | 1 | Unallocated. |
| - | 00 | 111 | - | Unallocated. |
| - | 01 | - | - | Unallocated. |
| - | 1x | - | - | Unallocated. |
| 0 | 00 | 000 | 0 | MADD - 32-bit variant |
| 0 | 00 | 000 | 1 | MSUB - 32-bit variant |
| 0 | 00 | 001 | 0 | Unallocated. |
| 0 | 00 | 001 | 1 | Unallocated. |
| 0 | 00 | 010 | 0 | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| sf | op54 | op31 | o0 | |
| 0 | 00 | 101 | 0 | Unallocated. |
| 0 | 00 | 101 | 1 | Unallocated. |
| 0 | 00 | 110 | 0 | Unallocated. |
| 1 | 00 | 000 | 0 | MADD - 64-bit variant |
| 1 | 00 | 000 | 1 | MSUB - 64-bit variant |
| 1 | 00 | 001 | 0 | SMADDL |
| 1 | 00 | 001 | 1 | SMSUBL |
| 1 | 00 | 010 | 0 | SMULH |
| 1 | 00 | 101 | 0 | UMADDL |
| 1 | 00 | 101 | 1 | UMSUBL |
| 1 | 00 | 110 | 0 | UMULH |

### C4.5.10 Logical (shifted register)

This section describes the encoding of the Logical (shifted register) instruction class. This section is decoded from *Data processing - register* on page C4-218.




| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| sf | opc | N | imm6 | |
| 0 | - | - | 1xxxxx | Unallocated. |
| 0 | 00 | 0 | - | AND (shifted register) - 32-bit variant |
| 0 | 00 | 1 | - | BIC (shifted register) - 32-bit variant |
| 0 | 01 | 0 | - | ORR (shifted register) - 32-bit variant |
| 0 | 01 | 1 | - | ORN (shifted register) - 32-bit variant |
| 0 | 10 | 0 | - | EOR (shifted register) - 32-bit variant |
| 0 | 10 | 1 | - | EON (shifted register) - 32-bit variant |
| 0 | 11 | 0 | - | ANDS (shifted register) - 32-bit variant |
| 0 | 11 | 1 | - | BICS (shifted register) - 32-bit variant |
| 1 | 00 | 0 | - | AND (shifted register) - 64-bit variant |
| 1 | 00 | 1 | - | BIC (shifted register) - 64-bit variant |
| 1 | 01 | 0 | - | ORR (shifted register) - 64-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| sf | opc | N | imm6 | |
| 1 | 01 | 1 | - | ORN (shifted register) - 64-bit variant |
| 1 | 10 | 0 | - | EOR (shifted register) - 64-bit variant |
| 1 | 10 | 1 | - | EON (shifted register) - 64-bit variant |
| 1 | 11 | 0 | - | ANDS (shifted register) - 64-bit variant |
| 1 | 11 | 1 | - | BICS (shifted register) - 64-bit variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C4.6 Data processing - SIMD and floating point

This section describes the encoding of the Data processing (SIMD and floating-point) instruction group. This section is decoded from *A64 instruction index by encoding* on page C4-186. For additional information on this functional group of instructions, see *Data processing - SIMD and floating-point* on page C3-165.

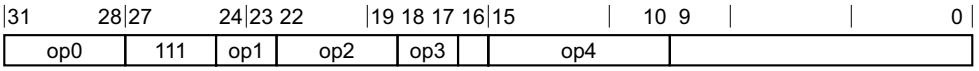



**Table C4-6 Encoding table for the Data Processing -- Scalar Floating-Point and Advanced SIMD group**

| Decode fields | | | | | Decode group or instruction page |
|---|---|---|---|---|---|
| op0 | op1 | op2 | op3 | op4 | |
| 0000 | 0x | x101 | 00 | xxxx10 | Unallocated. |
| 0010 | 0x | x101 | 00 | xxxx10 | Unallocated. |
| 0100 | 0x | x101 | 00 | xxxx10 | *Cryptographic AES* on page C4-252 |
| 0101 | 0x | x0xx | - | 0xxx00 | *Cryptographic three-register SHA* on page C4-252 |
| 0101 | 0x | x0xx | - | 0xxx10 | Unallocated. |
| 0101 | 0x | x101 | 00 | xxxx10 | *Cryptographic two-register SHA* on page C4-253 |
| 0110 | 0x | x101 | 00 | xxxx10 | Unallocated. |
| 0111 | 0x | x0xx | - | 0xxxx0 | Unallocated. |
| 0111 | 0x | x101 | 00 | xxxx10 | Unallocated. |
| 01x1 | 00 | 00xx | - | 0xxxx1 | *Advanced SIMD scalar copy* on page C4-232 |
| 01x1 | 01 | x0xx | - | 0xxxx1 | Unallocated. |
| 01x1 | 0x | 10xx | - | 0xxxx1 | Unallocated. |
| 01x1 | 0x | x0xx | - | 1xxxxx | Unallocated. |
| 01x1 | 0x | x100 | 00 | xxxx10 | *Advanced SIMD scalar two-register miscellaneous* on page C4-238 |
| 01x1 | 0x | x110 | 00 | xxxx10 | *Advanced SIMD scalar pairwise* on page C4-233 |
| 01x1 | 0x | x111 | 00 | xxxx10 | Unallocated. |
| 01x1 | 0x | x1xx | 1x | xxxx10 | Unallocated. |
| 01x1 | 0x | x1xx | x1 | xxxx10 | Unallocated. |
| 01x1 | 0x | x1xx | - | xxxx00 | *Advanced SIMD scalar three different* on page C4-235 |
| 01x1 | 0x | x1xx | - | xxxxx1 | *Advanced SIMD scalar three same* on page C4-236 |
| 01x1 | 10 | - | - | xxxxx1 | *Advanced SIMD scalar shift by immediate* on page C4-234 |
| 01x1 | 11 | - | - | xxxxx1 | Unallocated. |
| 01x1 | 1x | - | - | xxxxx0 | *Advanced SIMD scalar x indexed element* on page C4-240 |
| 0x00 | 0x | x0xx | - | 0xxx00 | *Advanced SIMD table lookup* on page C4-243 |
| 0x00 | 0x | x0xx | - | 0xxx10 | *Advanced SIMD permute* on page C4-232 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

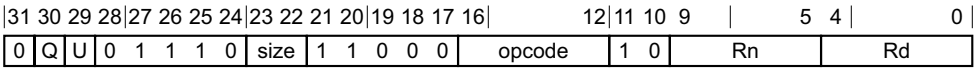
**Table C4-6 Encoding table for the Data Processing -- Scalar Floating-Point and Advanced SIMD group (continued)**

| Decode fields | | | | | Decode group or instruction page |
|---|---|---|---|---|---|
| op0 | op1 | op2 | op3 | op4 | |
| 0x10 | 0x | x0xx | - | 0xxxx0 | *Advanced SIMD extract* on page C4-230 |
| 0xx0 | 00 | 00xx | - | 0xxxx1 | *Advanced SIMD copy* on page C4-230 |
| 0xx0 | 01 | x0xx | - | 0xxxx1 | Unallocated. |
| 0xx0 | 0x | 10xx | - | 0xxxx1 | Unallocated. |
| 0xx0 | 0x | x0xx | - | 1xxxxx | Unallocated. |
| 0xx0 | 0x | x100 | 00 | xxxx10 | *Advanced SIMD two-register miscellaneous* on page C4-248 |
| 0xx0 | 0x | x110 | 00 | xxxx10 | *Advanced SIMD across lanes* on page C4-229 |
| 0xx0 | 0x | x111 | 00 | xxxx10 | Unallocated. |
| 0xx0 | 0x | x1xx | 1x | xxxx10 | Unallocated. |
| 0xx0 | 0x | x1xx | x1 | xxxx10 | Unallocated. |
| 0xx0 | 0x | x1xx | - | xxxx00 | *Advanced SIMD three different* on page C4-244 |
| 0xx0 | 0x | x1xx | - | xxxxx1 | *Advanced SIMD three same* on page C4-245 |
| 0xx0 | 10 | 0000 | - | xxxxx1 | *Advanced SIMD modified immediate* on page C4-231 |
| 0xx0 | 10 | != 0000 | - | xxxxx1 | *Advanced SIMD shift by immediate* on page C4-241 |
| 0xx0 | 11 | - | - | xxxxx1 | Unallocated. |
| 0xx0 | 1x | - | - | xxxxx0 | *Advanced SIMD vector x indexed element* on page C4-251 |
| 11x1 | - | - | - | - | Unallocated. |
| 1xx0 | - | - | - | - | Unallocated. |
| x0x1 | 0x | x0xx | - | - | *Conversion between floating-point and fixed-point* on page C4-259 |
| x0x1 | 0x | x1xx | - | 000000 | *Conversion between floating-point and integer* on page C4-260 |
| x0x1 | 0x | x1xx | - | 100000 | Unallocated. |
| x0x1 | 0x | x1xx | - | x10000 | *Floating-point data-processing (1 source)* on page C4-255 |
| x0x1 | 0x | x1xx | - | xx1000 | *Floating-point compare* on page C4-254 |
| x0x1 | 0x | x1xx | - | xxx100 | *Floating-point immediate* on page C4-259 |
| x0x1 | 0x | x1xx | - | xxxx01 | *Floating-point conditional compare* on page C4-254 |
| x0x1 | 0x | x1xx | - | xxxx10 | *Floating-point data-processing (2 source)* on page C4-257 |
| x0x1 | 0x | x1xx | - | xxxx11 | *Floating-point conditional select* on page C4-255 |
| x0x1 | 1x | - | - | - | *Floating-point data-processing (3 source)* on page C4-258 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C4.6.1 Advanced SIMD across lanes

This section describes the encoding of the Advanced SIMD across lanes instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

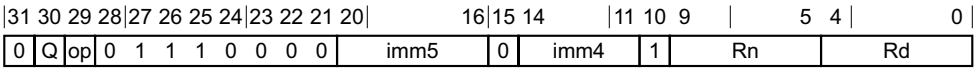



| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| - | - | 0000x | Unallocated. |
| - | - | 00010 | Unallocated. |
| - | - | 001xx | Unallocated. |
| - | - | 0100x | Unallocated. |
| - | - | 01011 | Unallocated. |
| - | - | 01101 | Unallocated. |
| - | - | 01110 | Unallocated. |
| - | - | 10xxx | Unallocated. |
| - | - | 1100x | Unallocated. |
| - | - | 111xx | Unallocated. |
| 0 | - | 00011 | SADDLV |
| 0 | - | 01010 | SMAXV |
| 0 | - | 11010 | SMINV |
| 0 | - | 11011 | ADDV |
| 0 | 0x | 01100 | Unallocated. |
| 0 | 0x | 01111 | Unallocated. |
| 0 | 1x | 01100 | Unallocated. |
| 0 | 1x | 01111 | Unallocated. |
| 1 | - | 00011 | UADDLV |
| 1 | - | 01010 | UMAXV |
| 1 | - | 11010 | UMINV |
| 1 | - | 11011 | Unallocated. |
| 1 | 0x | 01100 | FMAXNMV |
| 1 | 0x | 01111 | FMAXV |
| 1 | 1x | 01100 | FMINNMV |
| 1 | 1x | 01111 | FMINV |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C4.6.2 Advanced SIMD copy

This section describes the encoding of the Advanced SIMD copy instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

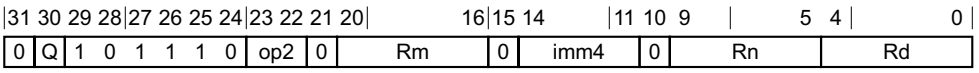



| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| Q | op | imm5 | imm4 | |
| - | - | x0000 | - | Unallocated. |
| - | 0 | - | 0000 | DUP (element) |
| - | 0 | - | 0001 | DUP (general) |
| - | 0 | - | 0010 | Unallocated. |
| - | 0 | - | 0100 | Unallocated. |
| - | 0 | - | 0110 | Unallocated. |
| - | 0 | - | 1xxx | Unallocated. |
| 0 | 0 | - | 0011 | Unallocated. |
| 0 | 0 | - | 0101 | SMOV |
| 0 | 0 | - | 0111 | UMOV |
| 0 | 1 | - | - | Unallocated. |
| 1 | 0 | - | 0011 | INS (general) |
| 1 | 0 | - | 0101 | SMOV |
| 1 | 0 | x1000 | 0111 | UMOV |
| 1 | 1 | - | - | INS (element) |

### C4.6.3 Advanced SIMD extract

This section describes the encoding of the Advanced SIMD extract instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

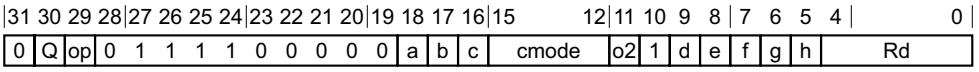

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | Instruction Page |
|---|---|
| op2 | |
| x1 | Unallocated. |
| 00 | EXT |
| 1x | Unallocated. |

### C4.6.4 Advanced SIMD modified immediate

This section describes the encoding of the Advanced SIMD modified immediate instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.



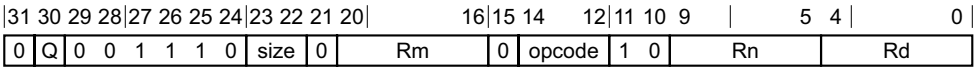

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| Q | op | cmode | o2 | |
| - | - | - | 1 | Unallocated. |
| - | 0 | 0xx0 | 0 | MOVI - 32-bit shifted immediate variant |
| - | 0 | 0xx1 | 0 | ORR (vector, immediate) - 32-bit variant |
| - | 0 | 10x0 | 0 | MOVI - 16-bit shifted immediate variant |
| - | 0 | 10x1 | 0 | ORR (vector, immediate) - 16-bit variant |
| - | 0 | 110x | 0 | MOVI - 32-bit shifting ones variant |
| - | 0 | 1110 | 0 | MOVI - 8-bit variant |
| - | 0 | 1111 | 0 | FMOV (vector, immediate) - Single-precision variant |
| - | 1 | 0xx0 | 0 | MVNI - 32-bit shifted immediate variant |
| - | 1 | 0xx1 | 0 | BIC (vector, immediate) - 32-bit variant |
| - | 1 | 10x0 | 0 | MVNI - 16-bit shifted immediate variant |
| - | 1 | 10x1 | 0 | BIC (vector, immediate) - 16-bit variant |
| - | 1 | 110x | 0 | MVNI - 32-bit shifting ones variant |
| 0 | 1 | 1110 | 0 | MOVI - 64-bit scalar variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| Q | op | cmode | o2 | |
| 0 | 1 | 1111 | 0 | Unallocated. |
| 1 | 1 | 1110 | 0 | MOVI - 64-bit vector variant |
| 1 | 1 | 1111 | 0 | FMOV (vector, immediate) - Double-precision variant |

### C4.6.5 Advanced SIMD permute

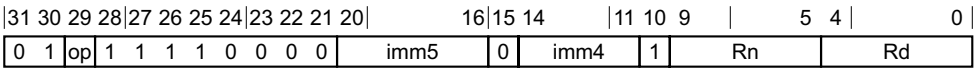
This section describes the encoding of the Advanced SIMD permute instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.




| Decode fields | Instruction Page |
|---|---|
| opcode | |
| 000 | Unallocated. |
| 001 | UZP1 |
| 010 | TRN1 |
| 011 | ZIP1 |
| 100 | Unallocated. |
| 101 | UZP2 |
| 110 | TRN2 |
| 111 | ZIP2 |

### C4.6.6 Advanced SIMD scalar copy

This section describes the encoding of the Advanced SIMD scalar copy instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.



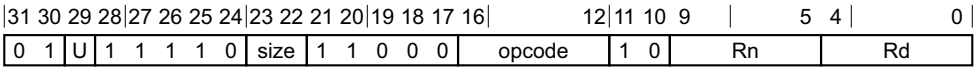

| Decode fields | | | Instruction Page |
|---|---|---|---|
| op | imm5 | imm4 | |
| 0 | - | xxx1 | Unallocated. |
| 0 | - | xx1x | Unallocated. |
| 0 | - | x1xx | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| op | imm5 | imm4 | |
| 0 | - | 0000 | DUP (element) |
| 0 | - | 1xxx | Unallocated. |
| 0 | x0000 | 0000 | Unallocated. |
| 1 | - | - | Unallocated. |

## C4.6.7 Advanced SIMD scalar pairwise

This section describes the encoding of the Advanced SIMD scalar pairwise instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.



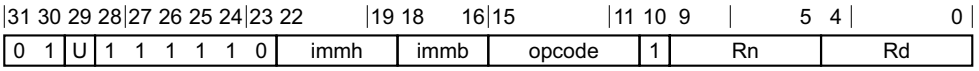

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| - | - | 00xxx | Unallocated. |
| - | - | 010xx | Unallocated. |
| - | - | 01110 | Unallocated. |
| - | - | 10xxx | Unallocated. |
| - | - | 1100x | Unallocated. |
| - | - | 11010 | Unallocated. |
| - | - | 111xx | Unallocated. |
| - | 1x | 01101 | Unallocated. |
| 0 | - | 11011 | ADDP (scalar) |
| 0 | 0x | 01100 | Unallocated. |
| 0 | 0x | 01101 | Unallocated. |
| 0 | 0x | 01111 | Unallocated. |
| 0 | 1x | 01100 | Unallocated. |
| 0 | 1x | 01111 | Unallocated. |
| 1 | - | 11011 | Unallocated. |
| 1 | 0x | 01100 | FMAXNMP (scalar) |
| 1 | 0x | 01101 | FADDP (scalar) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 1 | 0x | 01111 | FMAXP (scalar) |
| 1 | 1x | 01100 | FMINNMP (scalar) |
| 1 | 1x | 01111 | FMINP (scalar) |

### C4.6.8 Advanced SIMD scalar shift by immediate

This section describes the encoding of the Advanced SIMD scalar shift by immediate instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | immh | opcode | |
| - | != 0000 | 00001 | Unallocated. |
| - | != 0000 | 00011 | Unallocated. |
| - | != 0000 | 00101 | Unallocated. |
| - | != 0000 | 00111 | Unallocated. |
| - | != 0000 | 01001 | Unallocated. |
| - | != 0000 | 01011 | Unallocated. |
| - | != 0000 | 01101 | Unallocated. |
| - | != 0000 | 01111 | Unallocated. |
| - | != 0000 | 101xx | Unallocated. |
| - | != 0000 | 110xx | Unallocated. |
| - | != 0000 | 11101 | Unallocated. |
| - | != 0000 | 11110 | Unallocated. |
| - | 0000 | - | Unallocated. |
| 0 | != 0000 | 00000 | SSHR |
| 0 | != 0000 | 00010 | SSRA |
| 0 | != 0000 | 00100 | SRSHR |
| 0 | != 0000 | 00110 | SRSRA |
| 0 | != 0000 | 01000 | Unallocated. |
| 0 | != 0000 | 01010 | SHL |
| 0 | != 0000 | 01100 | Unallocated. |

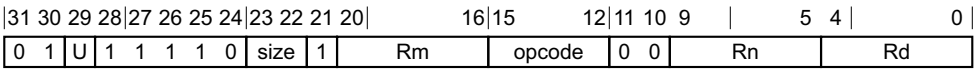

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | immh | opcode | |
| 0 | != 0000 | 01110 | SQSHL (immediate) |
| 0 | != 0000 | 10000 | Unallocated. |
| 0 | != 0000 | 10001 | Unallocated. |
| 0 | != 0000 | 10010 | SQSHRN, SQSHRN2 |
| 0 | != 0000 | 10011 | SQRSHRN, SQRSHRN2 |
| 0 | != 0000 | 11100 | SCVTF (vector, fixed-point) |
| 0 | != 0000 | 11111 | FCVTZS (vector, fixed-point) |
| 1 | != 0000 | 00000 | USHR |
| 1 | != 0000 | 00010 | USRA |
| 1 | != 0000 | 00100 | URSHR |
| 1 | != 0000 | 00110 | URSRA |
| 1 | != 0000 | 01000 | SRI |
| 1 | != 0000 | 01010 | SLI |
| 1 | != 0000 | 01100 | SQSHLU |
| 1 | != 0000 | 01110 | UQSHL (immediate) |
| 1 | != 0000 | 10000 | SQSHRUN, SQSHRUN2 |
| 1 | != 0000 | 10001 | SQRSHRUN, SQRSHRUN2 |
| 1 | != 0000 | 10010 | UQSHRN, UQSHRN2 |
| 1 | != 0000 | 10011 | UQRSHRN, UQRSHRN2 |
| 1 | != 0000 | 11100 | UCVTF (vector, fixed-point) |
| 1 | != 0000 | 11111 | FCVTZU (vector, fixed-point) |

### C4.6.9 Advanced SIMD scalar three different

This section describes the encoding of the Advanced SIMD scalar three different instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

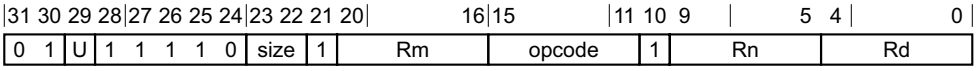

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | Instruction Page |
|---|---|---|
| U | opcode | |
| - | 00xx | Unallocated. |
| - | 01xx | Unallocated. |
| - | 1000 | Unallocated. |
| - | 1010 | Unallocated. |
| - | 1100 | Unallocated. |
| - | 111x | Unallocated. |
| 0 | 1001 | SQDMLAL, SQDMLAL2 (vector) |
| 0 | 1011 | SQDMLSL, SQDMLSL2 (vector) |
| 0 | 1101 | SQDMULL, SQDMULL2 (vector) |
| 1 | 1001 | Unallocated. |
| 1 | 1011 | Unallocated. |
| 1 | 1101 | Unallocated. |

### C4.6.10 Advanced SIMD scalar three same

This section describes the encoding of the Advanced SIMD scalar three same instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| - | - | 00000 | Unallocated. |
| - | - | 0001x | Unallocated. |
| - | - | 00100 | Unallocated. |
| - | - | 011xx | Unallocated. |
| - | - | 1001x | Unallocated. |
| - | 1x | 11011 | Unallocated. |
| 0 | - | 00001 | SQADD |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 0 | - | 00101 | SQSUB |
| 0 | - | 00110 | CMGT (register) |
| 0 | - | 00111 | CMGE (register) |
| 0 | - | 01000 | SSHL |
| 0 | - | 01001 | SQSHL (register) |
| 0 | - | 01010 | SRSHL |
| 0 | - | 01011 | SQRSHL |
| 0 | - | 10000 | ADD (vector) |
| 0 | - | 10001 | CMTST |
| 0 | - | 10100 | Unallocated. |
| 0 | - | 10101 | Unallocated. |
| 0 | - | 10110 | SQDMULH (vector) |
| 0 | - | 10111 | Unallocated. |
| 0 | 0x | 11000 | Unallocated. |
| 0 | 0x | 11001 | Unallocated. |
| 0 | 0x | 11010 | Unallocated. |
| 0 | 0x | 11011 | FMULX |
| 0 | 0x | 11100 | FCMEQ (register) |
| 0 | 0x | 11101 | Unallocated. |
| 0 | 0x | 11110 | Unallocated. |
| 0 | 0x | 11111 | FRECPS |
| 0 | 1x | 11000 | Unallocated. |
| 0 | 1x | 11001 | Unallocated. |
| 0 | 1x | 11010 | Unallocated. |
| 0 | 1x | 11100 | Unallocated. |
| 0 | 1x | 11101 | Unallocated. |
| 0 | 1x | 11110 | Unallocated. |
| 0 | 1x | 11111 | FRSQRTS |
| 1 | - | 00001 | UQADD |
| 1 | - | 00101 | UQSUB |
| 1 | - | 00110 | CMHI (register) |
| 1 | - | 00111 | CMHS (register) |

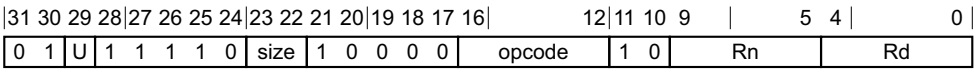

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 1 | - | 01000 | USHL |
| 1 | - | 01001 | UQSHL (register) |
| 1 | - | 01010 | URSHL |
| 1 | - | 01011 | UQRSHL |
| 1 | - | 10000 | SUB (vector) |
| 1 | - | 10001 | CMEQ (register) |
| 1 | - | 10100 | Unallocated. |
| 1 | - | 10101 | Unallocated. |
| 1 | - | 10110 | SQRDMULH (vector) |
| 1 | - | 10111 | Unallocated. |
| 1 | 0x | 11000 | Unallocated. |
| 1 | 0x | 11001 | Unallocated. |
| 1 | 0x | 11010 | Unallocated. |
| 1 | 0x | 11011 | Unallocated. |
| 1 | 0x | 11100 | FCMGE (register) |
| 1 | 0x | 11101 | FACGE |
| 1 | 0x | 11110 | Unallocated. |
| 1 | 0x | 11111 | Unallocated. |
| 1 | 1x | 11000 | Unallocated. |
| 1 | 1x | 11001 | Unallocated. |
| 1 | 1x | 11010 | FABD |
| 1 | 1x | 11100 | FCMGT (register) |
| 1 | 1x | 11101 | FACGT |
| 1 | 1x | 11110 | Unallocated. |
| 1 | 1x | 11111 | Unallocated. |

### C4.6.11 Advanced SIMD scalar two-register miscellaneous

This section describes the encoding of the Advanced SIMD scalar two-register miscellaneous instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| - | - | 0000x | Unallocated. |
| - | - | 00010 | Unallocated. |
| - | - | 0010x | Unallocated. |
| - | - | 00110 | Unallocated. |
| - | - | 01111 | Unallocated. |
| - | - | 1000x | Unallocated. |
| - | - | 10011 | Unallocated. |
| - | - | 10101 | Unallocated. |
| - | - | 10111 | Unallocated. |
| - | - | 1100x | Unallocated. |
| - | - | 11110 | Unallocated. |
| - | 0x | 011xx | Unallocated. |
| - | 0x | 11111 | Unallocated. |
| - | 1x | 10110 | Unallocated. |
| - | 1x | 11100 | Unallocated. |
| 0 | - | 00011 | SUQADD |
| 0 | - | 00111 | SQABS |
| 0 | - | 01000 | CMGT (zero) |
| 0 | - | 01001 | CMEQ (zero) |
| 0 | - | 01010 | CMLT (zero) |
| 0 | - | 01011 | ABS |
| 0 | - | 10010 | Unallocated. |
| 0 | - | 10100 | SQXTN, SQXTN2 |
| 0 | 0x | 10110 | Unallocated. |
| 0 | 0x | 11010 | FCVTNS (vector) |
| 0 | 0x | 11011 | FCVTMS (vector) |
| 0 | 0x | 11100 | FCVTAS (vector) |
| 0 | 0x | 11101 | SCVTF (vector, integer) |

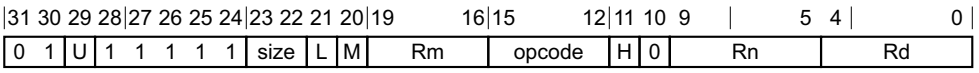

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 0 | 1x | 01100 | FCMGT (zero) |
| 0 | 1x | 01101 | FCMEQ (zero) |
| 0 | 1x | 01110 | FCMLT (zero) |
| 0 | 1x | 11010 | FCVTPS (vector) |
| 0 | 1x | 11011 | FCVTZS (vector, integer) |
| 0 | 1x | 11101 | FRECPE |
| 0 | 1x | 11111 | FRECPX |
| 1 | - | 00011 | USQADD |
| 1 | - | 00111 | SQNEG |
| 1 | - | 01000 | CMGE (zero) |
| 1 | - | 01001 | CMLE (zero) |
| 1 | - | 01010 | Unallocated. |
| 1 | - | 01011 | NEG (vector) |
| 1 | - | 10010 | SQXTUN, SQXTUN2 |
| 1 | - | 10100 | UQXTN, UQXTN2 |
| 1 | 0x | 10110 | FCVTXN, FCVTXN2 |
| 1 | 0x | 11010 | FCVTNU (vector) |
| 1 | 0x | 11011 | FCVTMU (vector) |
| 1 | 0x | 11100 | FCVTAU (vector) |
| 1 | 0x | 11101 | UCVTF (vector, integer) |
| 1 | 1x | 01100 | FCMGE (zero) |
| 1 | 1x | 01101 | FCMLE (zero) |
| 1 | 1x | 01110 | Unallocated. |
| 1 | 1x | 11010 | FCVTPU (vector) |
| 1 | 1x | 11011 | FCVTZU (vector, integer) |
| 1 | 1x | 11101 | FRSQRTE |
| 1 | 1x | 11111 | Unallocated. |

### C4.6.12 Advanced SIMD scalar x indexed element

This section describes the encoding of the Advanced SIMD scalar x indexed element instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

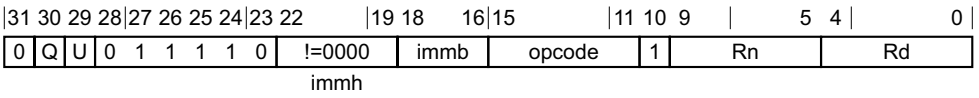



| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| - | - | 0000 | Unallocated. |
| - | - | 0010 | Unallocated. |
| - | - | 0100 | Unallocated. |
| - | - | 0110 | Unallocated. |
| - | - | 1000 | Unallocated. |
| - | - | 1010 | Unallocated. |
| - | - | 111x | Unallocated. |
| - | 0x | 0001 | Unallocated. |
| - | 0x | 0101 | Unallocated. |
| - | 0x | 1001 | Unallocated. |
| 0 | - | 0011 | SQDMLAL, SQDMLAL2 (by element) |
| 0 | - | 0111 | SQDMLSL, SQDMLSL2 (by element) |
| 0 | - | 1011 | SQDMULL, SQDMULL2 (by element) |
| 0 | - | 1100 | SQDMULH (by element) |
| 0 | - | 1101 | SQRDMULH (by element) |
| 0 | 1x | 0001 | FMLA (by element) |
| 0 | 1x | 0101 | FMLS (by element) |
| 0 | 1x | 1001 | FMUL (by element) |
| 1 | - | 0011 | Unallocated. |
| 1 | - | 0111 | Unallocated. |
| 1 | - | 1011 | Unallocated. |
| 1 | - | 110x | Unallocated. |
| 1 | 1x | 0001 | Unallocated. |
| 1 | 1x | 0101 | Unallocated. |
| 1 | 1x | 1001 | FMULX (by element) |

### C4.6.13 Advanced SIMD shift by immediate

This section describes the encoding of the Advanced SIMD shift by immediate instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



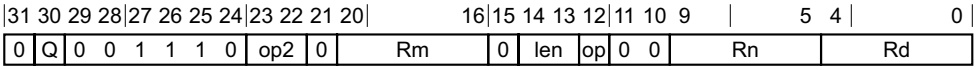

| Decode fields | | Instruction Page |
|---|---|---|
| U | opcode | |
| - | 00001 | Unallocated. |
| - | 00011 | Unallocated. |
| - | 00101 | Unallocated. |
| - | 00111 | Unallocated. |
| - | 01001 | Unallocated. |
| - | 01011 | Unallocated. |
| - | 01101 | Unallocated. |
| - | 01111 | Unallocated. |
| - | 10101 | Unallocated. |
| - | 1011x | Unallocated. |
| - | 110xx | Unallocated. |
| - | 11101 | Unallocated. |
| - | 11110 | Unallocated. |
| 0 | 00000 | SSHR |
| 0 | 00010 | SSRA |
| 0 | 00100 | SRSHR |
| 0 | 00110 | SRSRA |
| 0 | 01000 | Unallocated. |
| 0 | 01010 | SHL |
| 0 | 01100 | Unallocated. |
| 0 | 01110 | SQSHL (immediate) |
| 0 | 10000 | SHRN, SHRN2 |
| 0 | 10001 | RSHRN, RSHRN2 |
| 0 | 10010 | SQSHRN, SQSHRN2 |
| 0 | 10011 | SQRSHRN, SQRSHRN2 |
| 0 | 10100 | SSHLL, SSHLL2 |
| 0 | 11100 | SCVTF (vector, fixed-point) |
| 0 | 11111 | FCVTZS (vector, fixed-point) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | Instruction Page |
|---|---|---|
| U | opcode | |
| 1 | 00000 | USHR |
| 1 | 00010 | USRA |
| 1 | 00100 | URSHR |
| 1 | 00110 | URSRA |
| 1 | 01000 | SRI |
| 1 | 01010 | SLI |
| 1 | 01100 | SQSHLU |
| 1 | 01110 | UQSHL (immediate) |
| 1 | 10000 | SQSHRUN, SQSHRUN2 |
| 1 | 10001 | SQRSHRUN, SQRSHRUN2 |
| 1 | 10010 | UQSHRN, UQSHRN2 |
| 1 | 10011 | UQRSHRN, UQRSHRN2 |
| 1 | 10100 | USHLL, USHLL2 |
| 1 | 11100 | UCVTF (vector, fixed-point) |
| 1 | 11111 | FCVTZU (vector, fixed-point) |

### C4.6.14 Advanced SIMD table lookup

This section describes the encoding of the Advanced SIMD table lookup instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.



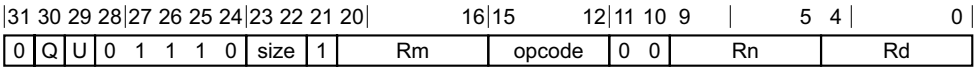

| Decode fields | | | Instruction Page |
|---|---|---|---|
| op2 | len | op | |
| x1 | - | - | Unallocated. |
| 00 | 00 | 0 | TBL - Single register table variant |
| 00 | 00 | 1 | TBX - Single register table variant |
| 00 | 01 | 0 | TBL - Two register table variant |
| 00 | 01 | 1 | TBX - Two register table variant |
| 00 | 10 | 0 | TBL - Three register table variant |
| 00 | 10 | 1 | TBX - Three register table variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| op2 | len | op | |
| 00 | 11 | 0 | TBL - Four register table variant |
| 00 | 11 | 1 | TBX - Four register table variant |
| 1x | - | - | Unallocated. |

### C4.6.15 Advanced SIMD three different

This section describes the encoding of the Advanced SIMD three different instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.



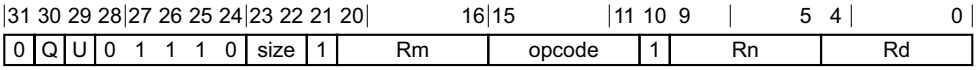

| Decode fields | | Instruction Page |
|---|---|---|
| U | opcode | |
| - | 1111 | Unallocated. |
| 0 | 0000 | SADDL, SADDL2 |
| 0 | 0001 | SADDW, SADDW2 |
| 0 | 0010 | SSUBL, SSUBL2 |
| 0 | 0011 | SSUBW, SSUBW2 |
| 0 | 0100 | ADDHN, ADDHN2 |
| 0 | 0101 | SABAL, SABAL2 |
| 0 | 0110 | SUBHN, SUBHN2 |
| 0 | 0111 | SABDL, SABDL2 |
| 0 | 1000 | SMLAL, SMLAL2 (vector) |
| 0 | 1001 | SQDMLAL, SQDMLAL2 (vector) |
| 0 | 1010 | SMLSL, SMLSL2 (vector) |
| 0 | 1011 | SQDMLSL, SQDMLSL2 (vector) |
| 0 | 1100 | SMULL, SMULL2 (vector) |
| 0 | 1101 | SQDMULL, SQDMULL2 (vector) |
| 0 | 1110 | PMULL, PMULL2 |
| 1 | 0000 | UADDL, UADDL2 |
| 1 | 0001 | UADDW, UADDW2 |
| 1 | 0010 | USUBL, USUBL2 |
| 1 | 0011 | USUBW, USUBW2 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | Instruction Page |
|---|---|---|
| U | opcode | |
| 1 | 0100 | RADDHN, RADDHN2 |
| 1 | 0101 | UABAL, UABAL2 |
| 1 | 0110 | RSUBHN, RSUBHN2 |
| 1 | 0111 | UABDL, UABDL2 |
| 1 | 1000 | UMLAL, UMLAL2 (vector) |
| 1 | 1001 | Unallocated. |
| 1 | 1010 | UMLSL, UMLSL2 (vector) |
| 1 | 1011 | Unallocated. |
| 1 | 1100 | UMULL, UMULL2 (vector) |
| 1 | 1101 | Unallocated. |
| 1 | 1110 | Unallocated. |

### C4.6.16 Advanced SIMD three same

This section describes the encoding of the Advanced SIMD three same instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 0 | - | 00000 | SHADD |
| 0 | - | 00001 | SQADD |
| 0 | - | 00010 | SRHADD |
| 0 | - | 00100 | SHSUB |
| 0 | - | 00101 | SQSUB |
| 0 | - | 00110 | CMGT (register) |
| 0 | - | 00111 | CMGE (register) |
| 0 | - | 01000 | SSHL |
| 0 | - | 01001 | SQSHL (register) |
| 0 | - | 01010 | SRSHL |
| 0 | - | 01011 | SQRSHL |
| 0 | - | 01100 | SMAX |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 0 | - | 01101 | SMIN |
| 0 | - | 01110 | SABD |
| 0 | - | 01111 | SABA |
| 0 | - | 10000 | ADD (vector) |
| 0 | - | 10001 | CMTST |
| 0 | - | 10010 | MLA (vector) |
| 0 | - | 10011 | MUL (vector) |
| 0 | - | 10100 | SMAXP |
| 0 | - | 10101 | SMINP |
| 0 | - | 10110 | SQDMULH (vector) |
| 0 | - | 10111 | ADDP (vector) |
| 0 | 0x | 11000 | FMAXNM (vector) |
| 0 | 0x | 11001 | FMLA (vector) |
| 0 | 0x | 11010 | FADD (vector) |
| 0 | 0x | 11011 | FMULX |
| 0 | 0x | 11100 | FCMEQ (register) |
| 0 | 0x | 11101 | Unallocated. |
| 0 | 0x | 11110 | FMAX (vector) |
| 0 | 0x | 11111 | FRECPS |
| 0 | 00 | 00011 | AND (vector) |
| 0 | 01 | 00011 | BIC (vector, register) |
| 0 | 1x | 11000 | FMINNM (vector) |
| 0 | 1x | 11001 | FMLS (vector) |
| 0 | 1x | 11010 | FSUB (vector) |
| 0 | 1x | 11011 | Unallocated. |
| 0 | 1x | 11100 | Unallocated. |
| 0 | 1x | 11101 | Unallocated. |
| 0 | 1x | 11110 | FMIN (vector) |
| 0 | 1x | 11111 | FRSQRTS |
| 0 | 10 | 00011 | ORR (vector, register) |
| 0 | 11 | 00011 | ORN (vector) |
| 1 | - | 00000 | UHADD |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

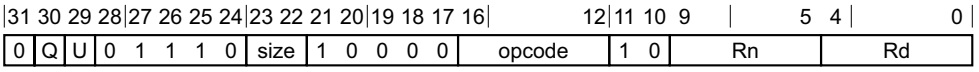
| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 1 | - | 00001 | UQADD |
| 1 | - | 00010 | URHADD |
| 1 | - | 00100 | UHSUB |
| 1 | - | 00101 | UQSUB |
| 1 | - | 00110 | CMHI (register) |
| 1 | - | 00111 | CMHS (register) |
| 1 | - | 01000 | USHL |
| 1 | - | 01001 | UQSHL (register) |
| 1 | - | 01010 | URSHL |
| 1 | - | 01011 | UQRSHL |
| 1 | - | 01100 | UMAX |
| 1 | - | 01101 | UMIN |
| 1 | - | 01110 | UABD |
| 1 | - | 01111 | UABA |
| 1 | - | 10000 | SUB (vector) |
| 1 | - | 10001 | CMEQ (register) |
| 1 | - | 10010 | MLS (vector) |
| 1 | - | 10011 | PMUL |
| 1 | - | 10100 | UMAXP |
| 1 | - | 10101 | UMINP |
| 1 | - | 10110 | SQRDMULH (vector) |
| 1 | - | 10111 | Unallocated. |
| 1 | 0x | 11000 | FMAXNMP (vector) |
| 1 | 0x | 11001 | Unallocated. |
| 1 | 0x | 11010 | FADDP (vector) |
| 1 | 0x | 11011 | FMUL (vector) |
| 1 | 0x | 11100 | FCMGE (register) |
| 1 | 0x | 11101 | FACGE |
| 1 | 0x | 11110 | FMAXP (vector) |
| 1 | 0x | 11111 | FDIV (vector) |
| 1 | 00 | 00011 | EOR (vector) |
| 1 | 01 | 00011 | BSL |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 1 | 1x | 11000 | FMINNMP (vector) |
| 1 | 1x | 11001 | Unallocated. |
| 1 | 1x | 11010 | FABD |
| 1 | 1x | 11011 | Unallocated. |
| 1 | 1x | 11100 | FCMGT (register) |
| 1 | 1x | 11101 | FACGT |
| 1 | 1x | 11110 | FMINP (vector) |
| 1 | 1x | 11111 | Unallocated. |
| 1 | 10 | 00011 | BIT |
| 1 | 11 | 00011 | BIF |

### C4.6.17 Advanced SIMD two-register miscellaneous

This section describes the encoding of the Advanced SIMD two-register miscellaneous instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| - | - | 1000x | Unallocated. |
| - | - | 10101 | Unallocated. |
| - | - | 11110 | Unallocated. |
| - | 0x | 011xx | Unallocated. |
| - | 0x | 11111 | Unallocated. |
| - | 1x | 10110 | Unallocated. |
| - | 1x | 10111 | Unallocated. |
| 0 | - | 00000 | REV64 |
| 0 | - | 00001 | REV16 (vector) |
| 0 | - | 00010 | SADDLP |
| 0 | - | 00011 | SUQADD |
| 0 | - | 00100 | CLS (vector) |
| 0 | - | 00101 | CNT |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 0 | - | 00110 | SADALP |
| 0 | - | 00111 | SQABS |
| 0 | - | 01000 | CMGT (zero) |
| 0 | - | 01001 | CMEQ (zero) |
| 0 | - | 01010 | CMLT (zero) |
| 0 | - | 01011 | ABS |
| 0 | - | 10010 | XTN, XTN2 |
| 0 | - | 10011 | Unallocated. |
| 0 | - | 10100 | SQXTN, SQXTN2 |
| 0 | 0x | 10110 | FCVTN, FCVTN2 |
| 0 | 0x | 10111 | FCVTL, FCVTL2 |
| 0 | 0x | 11000 | FRINTN (vector) |
| 0 | 0x | 11001 | FRINTM (vector) |
| 0 | 0x | 11010 | FCVTNS (vector) |
| 0 | 0x | 11011 | FCVTMS (vector) |
| 0 | 0x | 11100 | FCVTAS (vector) |
| 0 | 0x | 11101 | SCVTF (vector, integer) |
| 0 | 1x | 01100 | FCMGT (zero) |
| 0 | 1x | 01101 | FCMEQ (zero) |
| 0 | 1x | 01110 | FCMLT (zero) |
| 0 | 1x | 01111 | FABS (vector) |
| 0 | 1x | 11000 | FRINTP (vector) |
| 0 | 1x | 11001 | FRINTZ (vector) |
| 0 | 1x | 11010 | FCVTPS (vector) |
| 0 | 1x | 11011 | FCVTZS (vector, integer) |
| 0 | 1x | 11100 | URECPE |
| 0 | 1x | 11101 | FRECPE |
| 0 | 1x | 11111 | Unallocated. |
| 1 | - | 00000 | REV32 (vector) |
| 1 | - | 00001 | Unallocated. |
| 1 | - | 00010 | UADDLP |
| 1 | - | 00011 | USQADD |

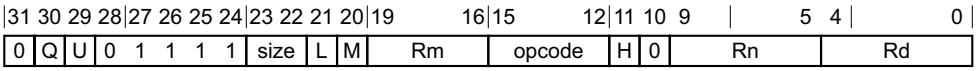

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 1 | - | 00100 | CLZ (vector) |
| 1 | - | 00110 | UADALP |
| 1 | - | 00111 | SQNEG |
| 1 | - | 01000 | CMGE (zero) |
| 1 | - | 01001 | CMLE (zero) |
| 1 | - | 01010 | Unallocated. |
| 1 | - | 01011 | NEG (vector) |
| 1 | - | 10010 | SQXTUN, SQXTUN2 |
| 1 | - | 10011 | SHLL, SHLL2 |
| 1 | - | 10100 | UQXTN, UQXTN2 |
| 1 | 0x | 10110 | FCVTXN, FCVTXN2 |
| 1 | 0x | 10111 | Unallocated. |
| 1 | 0x | 11000 | FRINTA (vector) |
| 1 | 0x | 11001 | FRINTX (vector) |
| 1 | 0x | 11010 | FCVTNU (vector) |
| 1 | 0x | 11011 | FCVTMU (vector) |
| 1 | 0x | 11100 | FCVTAU (vector) |
| 1 | 0x | 11101 | UCVTF (vector, integer) |
| 1 | 00 | 00101 | NOT |
| 1 | 01 | 00101 | RBIT (vector) |
| 1 | 1x | 00101 | Unallocated. |
| 1 | 1x | 01100 | FCMGE (zero) |
| 1 | 1x | 01101 | FCMLE (zero) |
| 1 | 1x | 01110 | Unallocated. |
| 1 | 1x | 01111 | FNEG (vector) |
| 1 | 1x | 11000 | Unallocated. |
| 1 | 1x | 11001 | FRINTI (vector) |
| 1 | 1x | 11010 | FCVTPU (vector) |
| 1 | 1x | 11011 | FCVTZU (vector, integer) |
| 1 | 1x | 11100 | URSQRTE |
| 1 | 1x | 11101 | FRSQRTE |
| 1 | 1x | 11111 | FSQRT (vector) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C4.6.18 Advanced SIMD vector x indexed element

This section describes the encoding of the Advanced SIMD vector x indexed element instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.



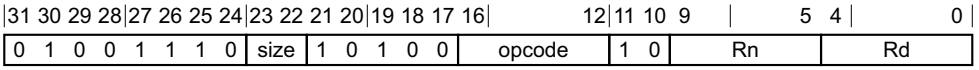

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| - | - | 111x | Unallocated. |
| - | 0x | 0001 | Unallocated. |
| - | 0x | 0101 | Unallocated. |
| - | 0x | 1001 | Unallocated. |
| 0 | - | 0000 | Unallocated. |
| 0 | - | 0010 | SMLAL, SMLAL2 (by element) |
| 0 | - | 0011 | SQDMLAL, SQDMLAL2 (by element) |
| 0 | - | 0100 | Unallocated. |
| 0 | - | 0110 | SMLSL, SMLSL2 (by element) |
| 0 | - | 0111 | SQDMLSL, SQDMLSL2 (by element) |
| 0 | - | 1000 | MUL (by element) |
| 0 | - | 1010 | SMULL, SMULL2 (by element) |
| 0 | - | 1011 | SQDMULL, SQDMULL2 (by element) |
| 0 | - | 1100 | SQDMULH (by element) |
| 0 | - | 1101 | SQRDMULH (by element) |
| 0 | 1x | 0001 | FMLA (by element) |
| 0 | 1x | 0101 | FMLS (by element) |
| 0 | 1x | 1001 | FMUL (by element) |
| 1 | - | 0000 | MLA (by element) |
| 1 | - | 0010 | UMLAL, UMLAL2 (by element) |
| 1 | - | 0011 | Unallocated. |
| 1 | - | 0100 | MLS (by element) |
| 1 | - | 0110 | UMLSL, UMLSL2 (by element) |
| 1 | - | 0111 | Unallocated. |
| 1 | - | 1000 | Unallocated. |
| 1 | - | 1010 | UMULL, UMULL2 (by element) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| U | size | opcode | |
| 1 | - | 1011 | Unallocated. |
| 1 | - | 110x | Unallocated. |
| 1 | 1x | 0001 | Unallocated. |
| 1 | 1x | 0101 | Unallocated. |
| 1 | 1x | 1001 | FMULX (by element) |

### C4.6.19 Cryptographic AES

This section describes the encoding of the Cryptographic AES instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

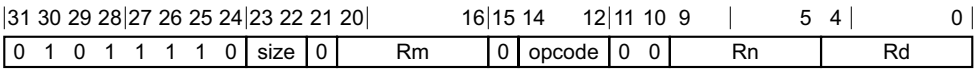



| Decode fields | | Instruction Page |
|---|---|---|
| size | opcode | |
| - | x1xxx | Unallocated. |
| - | 000xx | Unallocated. |
| - | 1xxxx | Unallocated. |
| x1 | - | Unallocated. |
| 00 | 00100 | AESE |
| 00 | 00101 | AESD |
| 00 | 00110 | AESMC |
| 00 | 00111 | AESIMC |
| 1x | - | Unallocated. |

### C4.6.20 Cryptographic three-register SHA

This section describes the encoding of the Cryptographic three-register SHA instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

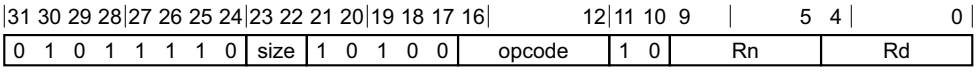

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | Instruction Page |
|---|---|---|
| size | opcode | |
| - | 111 | Unallocated. |
| x1 | - | Unallocated. |
| 00 | 000 | SHA1C |
| 00 | 001 | SHA1P |
| 00 | 010 | SHA1M |
| 00 | 011 | SHA1SU0 |
| 00 | 100 | SHA256H |
| 00 | 101 | SHA256H2 |
| 00 | 110 | SHA256SU1 |
| 1x | - | Unallocated. |

### C4.6.21 Cryptographic two-register SHA

This section describes the encoding of the Cryptographic two-register SHA instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

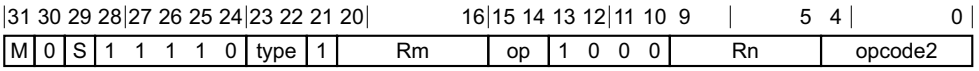



| Decode fields | | Instruction Page |
|---|---|---|
| size | opcode | |
| - | xx1xx | Unallocated. |
| - | x1xxx | Unallocated. |
| - | 1xxxx | Unallocated. |
| x1 | - | Unallocated. |
| 00 | 00000 | SHA1H |
| 00 | 00001 | SHA1SU1 |
| 00 | 00010 | SHA256SU0 |
| 00 | 00011 | Unallocated. |
| 1x | - | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C4.6.22 Floating-point compare

This section describes the encoding of the Floating-point compare instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

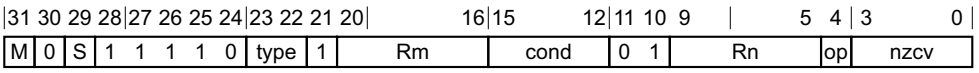



| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| M | S | type | op | opcode2 | |
| - | - | - | - | xxxx1 | Unallocated. |
| - | - | - | - | xxx1x | Unallocated. |
| - | - | - | - | xx1xx | Unallocated. |
| - | - | - | x1 | - | Unallocated. |
| - | - | - | 1x | - | Unallocated. |
| - | - | 1x | - | - | Unallocated. |
| - | 1 | - | - | - | Unallocated. |
| 0 | 0 | 00 | 00 | 00000 | FCMP |
| 0 | 0 | 00 | 00 | 01000 | FCMP |
| 0 | 0 | 00 | 00 | 10000 | FCMPE |
| 0 | 0 | 00 | 00 | 11000 | FCMPE |
| 0 | 0 | 01 | 00 | 00000 | FCMP |
| 0 | 0 | 01 | 00 | 01000 | FCMP |
| 0 | 0 | 01 | 00 | 10000 | FCMPE |
| 0 | 0 | 01 | 00 | 11000 | FCMPE |
| 1 | - | - | - | - | Unallocated. |

### C4.6.23 Floating-point conditional compare

This section describes the encoding of the Floating-point conditional compare instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

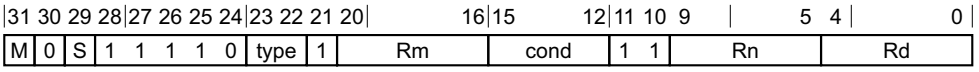

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| M | S | type | op | |
| - | - | 1x | - | Unallocated. |
| - | 1 | - | - | Unallocated. |
| 0 | 0 | 00 | 0 | FCCMP - Single-precision variant |
| 0 | 0 | 00 | 1 | FCCMPE - Single-precision variant |
| 0 | 0 | 01 | 0 | FCCMP - Double-precision variant |
| 0 | 0 | 01 | 1 | FCCMPE - Double-precision variant |
| 1 | - | - | - | Unallocated. |

### C4.6.24 Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

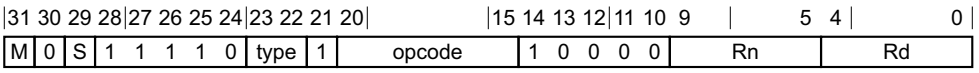



| Decode fields | | | Instruction Page |
|---|---|---|---|
| M | S | type | |
| - | - | 1x | Unallocated. |
| - | 1 | - | Unallocated. |
| 0 | 0 | 00 | FCSEL - Single-precision variant |
| 0 | 0 | 01 | FCSEL - Double-precision variant |
| 1 | - | - | Unallocated. |

### C4.6.25 Floating-point data-processing (1 source)

This section describes the encoding of the Floating-point data-processing (1 source) instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta



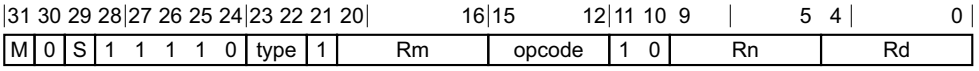

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| M | S | type | opcode | |
| - | - | - | x1xxxx | Unallocated. |
| - | - | - | 1xxxxx | Unallocated. |
| - | 1 | - | - | Unallocated. |
| 0 | 0 | 00 | 000000 | FMOV (register) - Single-precision variant |
| 0 | 0 | 00 | 000001 | FABS (scalar) - Single-precision variant |
| 0 | 0 | 00 | 000010 | FNEG (scalar) - Single-precision variant |
| 0 | 0 | 00 | 000011 | FSQRT (scalar) - Single-precision variant |
| 0 | 0 | 00 | 000100 | Unallocated. |
| 0 | 0 | 00 | 000101 | FCVT - Single-precision to double-precision variant |
| 0 | 0 | 00 | 000110 | Unallocated. |
| 0 | 0 | 00 | 000111 | FCVT - Single-precision to half-precision variant |
| 0 | 0 | 00 | 001000 | FRINTN (scalar) - Single-precision variant |
| 0 | 0 | 00 | 001001 | FRINTP (scalar) - Single-precision variant |
| 0 | 0 | 00 | 001010 | FRINTM (scalar) - Single-precision variant |
| 0 | 0 | 00 | 001011 | FRINTZ (scalar) - Single-precision variant |
| 0 | 0 | 00 | 001100 | FRINTA (scalar) - Single-precision variant |
| 0 | 0 | 00 | 001101 | Unallocated. |
| 0 | 0 | 00 | 001110 | FRINTX (scalar) - Single-precision variant |
| 0 | 0 | 00 | 001111 | FRINTI (scalar) - Single-precision variant |
| 0 | 0 | 01 | 000000 | FMOV (register) - Double-precision variant |
| 0 | 0 | 01 | 000001 | FABS (scalar) - Double-precision variant |
| 0 | 0 | 01 | 000010 | FNEG (scalar) - Double-precision variant |
| 0 | 0 | 01 | 000011 | FSQRT (scalar) - Double-precision variant |
| 0 | 0 | 01 | 000100 | FCVT - Double-precision to single-precision variant |
| 0 | 0 | 01 | 000101 | Unallocated. |
| 0 | 0 | 01 | 000110 | Unallocated. |
| 0 | 0 | 01 | 000111 | FCVT - Double-precision to half-precision variant |
| 0 | 0 | 01 | 001000 | FRINTN (scalar) - Double-precision variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| M | S | type | opcode | |
| 0 | 0 | 01 | 001001 | FRINTP (scalar) - Double-precision variant |
| 0 | 0 | 01 | 001010 | FRINTM (scalar) - Double-precision variant |
| 0 | 0 | 01 | 001011 | FRINTZ (scalar) - Double-precision variant |
| 0 | 0 | 01 | 001100 | FRINTA (scalar) - Double-precision variant |
| 0 | 0 | 01 | 001101 | Unallocated. |
| 0 | 0 | 01 | 001110 | FRINTX (scalar) - Double-precision variant |
| 0 | 0 | 01 | 001111 | FRINTI (scalar) - Double-precision variant |
| 0 | 0 | 10 | 00xxxx | Unallocated. |
| 0 | 0 | 11 | 0000xx | Unallocated. |
| 0 | 0 | 11 | 000100 | FCVT - Half-precision to single-precision variant |
| 0 | 0 | 11 | 000101 | FCVT - Half-precision to double-precision variant |
| 0 | 0 | 11 | 00011x | Unallocated. |
| 0 | 0 | 11 | 001xxx | Unallocated. |
| 1 | - | - | - | Unallocated. |

### C4.6.26 Floating-point data-processing (2 source)

This section describes the encoding of the Floating-point data-processing (2 source) instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.



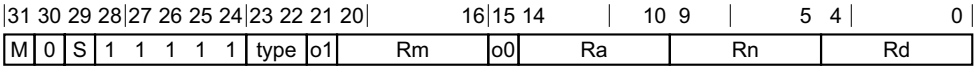

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| M | S | type | opcode | |
| - | - | - | 1xx1 | Unallocated. |
| - | - | - | 1x1x | Unallocated. |
| - | - | - | 11xx | Unallocated. |
| - | - | 1x | - | Unallocated. |
| - | 1 | - | - | Unallocated. |
| 0 | 0 | 00 | 0000 | FMUL (scalar) - Single-precision variant |
| 0 | 0 | 00 | 0001 | FDIV (scalar) - Single-precision variant |
| 0 | 0 | 00 | 0010 | FADD (scalar) - Single-precision variant |
| 0 | 0 | 00 | 0011 | FSUB (scalar) - Single-precision variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| M | S | type | opcode | |
| 0 | 0 | 00 | 0100 | FMAX (scalar) - Single-precision variant |
| 0 | 0 | 00 | 0101 | FMIN (scalar) - Single-precision variant |
| 0 | 0 | 00 | 0110 | FMAXNM (scalar) - Single-precision variant |
| 0 | 0 | 00 | 0111 | FMINNM (scalar) - Single-precision variant |
| 0 | 0 | 00 | 1000 | FNMUL - Single-precision variant |
| 0 | 0 | 01 | 0000 | FMUL (scalar) - Double-precision variant |
| 0 | 0 | 01 | 0001 | FDIV (scalar) - Double-precision variant |
| 0 | 0 | 01 | 0010 | FADD (scalar) - Double-precision variant |
| 0 | 0 | 01 | 0011 | FSUB (scalar) - Double-precision variant |
| 0 | 0 | 01 | 0100 | FMAX (scalar) - Double-precision variant |
| 0 | 0 | 01 | 0101 | FMIN (scalar) - Double-precision variant |
| 0 | 0 | 01 | 0110 | FMAXNM (scalar) - Double-precision variant |
| 0 | 0 | 01 | 0111 | FMINNM (scalar) - Double-precision variant |
| 0 | 0 | 01 | 1000 | FNMUL - Double-precision variant |
| 1 | - | - | - | Unallocated. |

### C4.6.27 Floating-point data-processing (3 source)

This section describes the encoding of the Floating-point data-processing (3 source) instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.



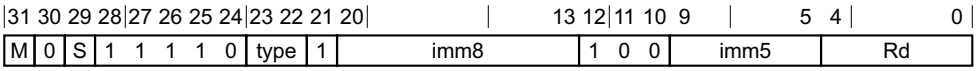

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| M | S | type | o1 | o0 | |
| - | - | 1x | - | - | Unallocated. |
| - | 1 | - | - | - | Unallocated. |
| 0 | 0 | 00 | 0 | 0 | FMADD - Single-precision variant |
| 0 | 0 | 00 | 0 | 1 | FMSUB - Single-precision variant |
| 0 | 0 | 00 | 1 | 0 | FNMADD - Single-precision variant |
| 0 | 0 | 00 | 1 | 1 | FNMSUB - Single-precision variant |
| 0 | 0 | 01 | 0 | 0 | FMADD - Double-precision variant |
| 0 | 0 | 01 | 0 | 1 | FMSUB - Double-precision variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| M | S | type | o1 | o0 | |
| 0 | 0 | 01 | 1 | 0 | FNMADD - Double-precision variant |
| 0 | 0 | 01 | 1 | 1 | FNMSUB - Double-precision variant |
| 1 | - | - | - | - | Unallocated. |

### C4.6.28 Floating-point immediate

This section describes the encoding of the Floating-point immediate instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.

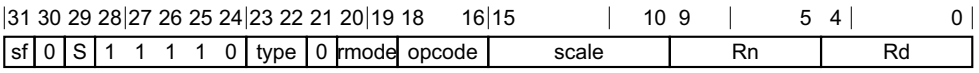



| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| M | S | type | imm5 | |
| - | - | - | xxxx1 | Unallocated. |
| - | - | - | xxx1x | Unallocated. |
| - | - | - | xx1xx | Unallocated. |
| - | - | - | x1xxx | Unallocated. |
| - | - | - | 1xxxx | Unallocated. |
| - | - | 1x | - | Unallocated. |
| - | 1 | - | - | Unallocated. |
| 0 | 0 | 00 | 00000 | FMOV (scalar, immediate) - Single-precision variant |
| 0 | 0 | 01 | 00000 | FMOV (scalar, immediate) - Double-precision variant |
| 1 | - | - | - | Unallocated. |

### C4.6.29 Conversion between floating-point and fixed-point

This section describes the encoding of the Conversion between floating-point and fixed-point instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

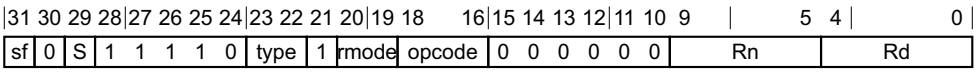



| Decode fields | | | | | | Instruction Page |
|---|---|---|---|---|---|---|
| sf | S | type | rmode | opcode | scale | |
| - | - | - | - | 1xx | - | Unallocated. |
| - | - | - | x0 | 00x | - | Unallocated. |
| - | - | - | x1 | 01x | - | Unallocated. |
| - | - | - | 0x | 00x | - | Unallocated. |
| - | - | - | 1x | 01x | - | Unallocated. |
| - | - | 1x | - | - | - | Unallocated. |
| - | 1 | - | - | - | - | Unallocated. |
| 0 | - | - | - | - | 0xxxx | Unallocated. |
| 0 | 0 | 00 | 00 | 010 | - | SCVTF (scalar, fixed-point) - 32-bit to single-precision variant |
| 0 | 0 | 00 | 00 | 011 | - | UCVTF (scalar, fixed-point) - 32-bit to single-precision variant |
| 0 | 0 | 00 | 11 | 000 | - | FCVTZS (scalar, fixed-point) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 11 | 001 | - | FCVTZU (scalar, fixed-point) - Single-precision to 32-bit variant |
| 0 | 0 | 01 | 00 | 010 | - | SCVTF (scalar, fixed-point) - 32-bit to double-precision variant |
| 0 | 0 | 01 | 00 | 011 | - | UCVTF (scalar, fixed-point) - 32-bit to double-precision variant |
| 0 | 0 | 01 | 11 | 000 | - | FCVTZS (scalar, fixed-point) - Double-precision to 32-bit variant |
| 0 | 0 | 01 | 11 | 001 | - | FCVTZU (scalar, fixed-point) - Double-precision to 32-bit variant |
| 1 | 0 | 00 | 00 | 010 | - | SCVTF (scalar, fixed-point) - 64-bit to single-precision variant |
| 1 | 0 | 00 | 00 | 011 | - | UCVTF (scalar, fixed-point) - 64-bit to single-precision variant |
| 1 | 0 | 00 | 11 | 000 | - | FCVTZS (scalar, fixed-point) - Single-precision to 64-bit variant |
| 1 | 0 | 00 | 11 | 001 | - | FCVTZU (scalar, fixed-point) - Single-precision to 64-bit variant |
| 1 | 0 | 01 | 00 | 010 | - | SCVTF (scalar, fixed-point) - 64-bit to double-precision variant |
| 1 | 0 | 01 | 00 | 011 | - | UCVTF (scalar, fixed-point) - 64-bit to double-precision variant |
| 1 | 0 | 01 | 11 | 000 | - | FCVTZS (scalar, fixed-point) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 11 | 001 | - | FCVTZU (scalar, fixed-point) - Double-precision to 64-bit variant |

### C4.6.30 Conversion between floating-point and integer

This section describes the encoding of the Conversion between floating-point and integer instruction class. This section is decoded from *Data processing - SIMD and floating point* on page C4-227.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| sf | S | type | rmode | opcode | |
| - | - | - | x1 | 01x | Unallocated. |
| - | - | - | x1 | 10x | Unallocated. |
| - | - | - | 1x | 01x | Unallocated. |
| - | - | - | 1x | 10x | Unallocated. |
| - | - | 11 | - | - | Unallocated. |
| - | 0 | 10 | - | 0xx | Unallocated. |
| - | 0 | 10 | - | 10x | Unallocated. |
| - | 1 | - | - | - | Unallocated. |
| 0 | 0 | 00 | x1 | 11x | Unallocated. |
| 0 | 0 | 00 | 00 | 000 | FCVTNS (scalar) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 00 | 001 | FCVTNU (scalar) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 00 | 010 | SCVTF (scalar, integer) - 32-bit to single-precision variant |
| 0 | 0 | 00 | 00 | 011 | UCVTF (scalar, integer) - 32-bit to single-precision variant |
| 0 | 0 | 00 | 00 | 100 | FCVTAS (scalar) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 00 | 101 | FCVTAU (scalar) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 00 | 110 | FMOV (general) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 00 | 111 | FMOV (general) - 32-bit to single-precision variant |
| 0 | 0 | 00 | 01 | 000 | FCVTPS (scalar) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 01 | 001 | FCVTPU (scalar) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 1x | 11x | Unallocated. |
| 0 | 0 | 00 | 10 | 000 | FCVTMS (scalar) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 10 | 001 | FCVTMU (scalar) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 11 | 000 | FCVTZS (scalar, integer) - Single-precision to 32-bit variant |
| 0 | 0 | 00 | 11 | 001 | FCVTZU (scalar, integer) - Single-precision to 32-bit variant |
| 0 | 0 | 01 | - | 11x | Unallocated. |
| 0 | 0 | 01 | 00 | 000 | FCVTNS (scalar) - Double-precision to 32-bit variant |
| 0 | 0 | 01 | 00 | 001 | FCVTNU (scalar) - Double-precision to 32-bit variant |
| 0 | 0 | 01 | 00 | 010 | SCVTF (scalar, integer) - 32-bit to double-precision variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| sf | S | type | rmode | opcode | |
| 0 | 0 | 01 | 00 | 011 | UCVTF (scalar, integer) - 32-bit to double-precision variant |
| 0 | 0 | 01 | 00 | 100 | FCVTAS (scalar) - Double-precision to 32-bit variant |
| 0 | 0 | 01 | 00 | 101 | FCVTAU (scalar) - Double-precision to 32-bit variant |
| 0 | 0 | 01 | 01 | 000 | FCVTPS (scalar) - Double-precision to 32-bit variant |
| 0 | 0 | 01 | 01 | 001 | FCVTPU (scalar) - Double-precision to 32-bit variant |
| 0 | 0 | 01 | 10 | 000 | FCVTMS (scalar) - Double-precision to 32-bit variant |
| 0 | 0 | 01 | 10 | 001 | FCVTMU (scalar) - Double-precision to 32-bit variant |
| 0 | 0 | 01 | 11 | 000 | FCVTZS (scalar, integer) - Double-precision to 32-bit variant |
| 0 | 0 | 01 | 11 | 001 | FCVTZU (scalar, integer) - Double-precision to 32-bit variant |
| 0 | 0 | 10 | - | 11x | Unallocated. |
| 1 | 0 | 00 | - | 11x | Unallocated. |
| 1 | 0 | 00 | 00 | 000 | FCVTNS (scalar) - Single-precision to 64-bit variant |
| 1 | 0 | 00 | 00 | 001 | FCVTNU (scalar) - Single-precision to 64-bit variant |
| 1 | 0 | 00 | 00 | 010 | SCVTF (scalar, integer) - 64-bit to single-precision variant |
| 1 | 0 | 00 | 00 | 011 | UCVTF (scalar, integer) - 64-bit to single-precision variant |
| 1 | 0 | 00 | 00 | 100 | FCVTAS (scalar) - Single-precision to 64-bit variant |
| 1 | 0 | 00 | 00 | 101 | FCVTAU (scalar) - Single-precision to 64-bit variant |
| 1 | 0 | 00 | 01 | 000 | FCVTPS (scalar) - Single-precision to 64-bit variant |
| 1 | 0 | 00 | 01 | 001 | FCVTPU (scalar) - Single-precision to 64-bit variant |
| 1 | 0 | 00 | 10 | 000 | FCVTMS (scalar) - Single-precision to 64-bit variant |
| 1 | 0 | 00 | 10 | 001 | FCVTMU (scalar) - Single-precision to 64-bit variant |
| 1 | 0 | 00 | 11 | 000 | FCVTZS (scalar, integer) - Single-precision to 64-bit variant |
| 1 | 0 | 00 | 11 | 001 | FCVTZU (scalar, integer) - Single-precision to 64-bit variant |
| 1 | 0 | 01 | x1 | 11x | Unallocated. |
| 1 | 0 | 01 | 00 | 000 | FCVTNS (scalar) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 00 | 001 | FCVTNU (scalar) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 00 | 010 | SCVTF (scalar, integer) - 64-bit to double-precision variant |
| 1 | 0 | 01 | 00 | 011 | UCVTF (scalar, integer) - 64-bit to double-precision variant |
| 1 | 0 | 01 | 00 | 100 | FCVTAS (scalar) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 00 | 101 | FCVTAU (scalar) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 00 | 110 | FMOV (general) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 00 | 111 | FMOV (general) - 64-bit to double-precision variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| sf | S | type | rmode | opcode | |
| 1 | 0 | 01 | 01 | 000 | FCVTPS (scalar) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 01 | 001 | FCVTPU (scalar) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 1x | 11x | Unallocated. |
| 1 | 0 | 01 | 10 | 000 | FCVTMS (scalar) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 10 | 001 | FCVTMU (scalar) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 11 | 000 | FCVTZS (scalar, integer) - Double-precision to 64-bit variant |
| 1 | 0 | 01 | 11 | 001 | FCVTZU (scalar, integer) - Double-precision to 64-bit variant |
| 1 | 0 | 10 | x0 | 11x | Unallocated. |
| 1 | 0 | 10 | 01 | 110 | FMOV (general) - Top half of 128-bit to 64-bit variant |
| 1 | 0 | 10 | 01 | 111 | FMOV (general) - 64-bit to top half of 128-bit variant |
| 1 | 0 | 10 | 1x | 11x | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter C5
# The A64 System Instruction Class

This chapter describes the A64 System instruction class, and the System instruction class encoding space, that is a subset of the System registers encoding space. It contains the following sections:

- *The System instruction class encoding space* on page C5-266.
- *Special-purpose registers* on page C5-289.
- *A64 system instructions for cache maintenance* on page C5-342.
- *A64 system instructions for address translation* on page C5-358.
- *A64 system instructions for TLB maintenance* on page C5-371.

See *Conventions used in AArch64 instruction and System register descriptions* on page C2-135 for information about entries used in the instruction encoding descriptions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C5.1 The System instruction class encoding space

Part of the A64 instruction encoding space is assigned to instructions that access the System register space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- Access to Special-purpose registers such as SPSR_ELx, ELR_ELx, and the equivalent fields of the Process State.
- The cache and TLB maintenance instructions and address translation instructions.
- Barriers and the `CLREX` instruction.
- Architectural hint instructions.

This section describes the general model for accessing this functionality.

**Note**

In AArch32 state this functionality is provided through conceptual coprocessors CP14 and CP15, and in part through CP10 and CP11. These are accessed through a generic coprocessor interface. In ARMv8:

- AArch32 state retains this conceptual coprocessor model, and adds register and operation aliases, to simplify access to this functionality.
- In the instruction encoding descriptions, AArch64 state retains the naming of the instruction arguments as `op1`, `CRn`, `CRm`, and `op2`. However, there is no functional distinction between the `op`*n* arguments and the `CR`*x* arguments.

*Principles of the System instruction class encoding* describes some general properties of these encodings. *System instruction class encoding overview* on page C5-267 then describes the top-level encoding of these instructions, and the following sections then describe the next level of the encoding hierarchy:

- *op0==0b00, architectural hints, barriers and CLREX, and PSTATE access* on page C5-268.
- *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-271.
- *op0==0b10, Moves to and from debug and trace System registers* on page C5-275.
- *op0==0b11, Moves to and from non-debug System registers and Special-purpose registers* on page C5-278.
- *Reserved encodings for IMPLEMENTATION DEFINED registers* on page C5-288.

### C5.1.1 Principles of the System instruction class encoding

In ARMv8, an encoding in the System instruction space is identified by a set of arguments, {`op0`, `op1`, `CRn`, `CRm`, `op2`}. These form an encoding hierarchy, where:

`op0` Defines the top-level division of the encoding space, see *System instruction class encoding overview* on page C5-267.

`op1` Identifies the lowest Exception level at which the encoding is accessible, as follows:

- **Accessible at EL0** `op1` has the value 3.
- **Accessible at EL1** `op1` has the value 0, 1, or 2. The value is the same as the `op1` value used to access the equivalent AArch32 register.
- **Accessible at EL2** `op1` has the value 4.
- **Accessible at EL3** `op1` has the value 6.

ARM strongly recommends that implementers adopt this use of `op1` when using the IMPLEMENTATION DEFINED regions of the encoding space described in *Reserved encodings for IMPLEMENTATION DEFINED registers* on page C5-288.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### System register width

In AArch64 state, each encoding in the System instruction space can provide access to a 64-bit register. An AArch64 System register is described as either a *32-bit register* or a *64-bit register*. For a 32-bit registers, the upper bits, bits[63:32], are RES0.

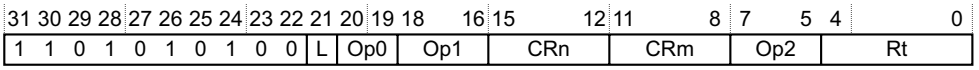
## C5.1.2 System instruction class encoding overview

The encoding of the System instruction class describes each instruction as being either:

- A transfer to a System register. This is a System instruction with the semantics of a write.
- A transfer from a System register. This is a System instruction with the semantics of a read.

A System instruction that initiates an operation operates as if it was making a transfer to a register.

In the AArch64 instruction set, the decode structure for the System instruction class is:




The value of L indicates the transfer direction:

**0** Transfer to System register.

**1** Transfer from System register.

The op0 field is the top level encoding of the System instruction type. Its possible values are:

0b00 These encodings provide:

- Instructions with an immediate field for accessing PSTATE, the current PE state.
- The architectural hint instructions.
- Barriers and the CLREX instruction.

For more information about these encodings, see *op0==0b00, architectural hints, barriers and CLREX, and PSTATE access* on page C5-268.

0b01 These encodings provide the cache maintenance, TLB maintenance, and address translation instructions.

**Note**

These are equivalent to operations in the AArch32 CP15 space.

For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-271.

0b10 These encodings provide moves to and from:

- Legacy AArch32 System registers for execution environments, to provide access to these registers from higher exception levels that are using AArch64.
- Debug and trace registers.

**Note**

These are equivalent to the registers in the AArch32 CP14 space.

For more information, see *op0==0b10, Moves to and from debug and trace System registers* on page C5-275.

0b11 These encodings provide:

- Moves to and from Non-debug System registers. The accessed registers provide system control, and system status information.

  **Note**

  The accessed registers are equivalent to the registers in the AArch32 CP15 space.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Access to Special-purpose registers.

For more information, see *Instructions for accessing Special-purpose registers* on page C5-287 and *Instructions for accessing non-debug System registers* on page C5-278.

### UNDEFINED behaviors

In the System register instruction encoding space, the following principles apply:

- All unallocated encodings are treated as UNDEFINED.
- All encodings with `L==1` and `op0==0b0x` are UNDEFINED, except for encodings in the area reserved for IMPLEMENTATION DEFINED use, see *Reserved encodings for IMPLEMENTATION DEFINED registers* on page C5-288.

For registers and operations that are accessible from a particular Exception level, any attempt to access those registers from a lower Exception level is UNDEFINED.

If a particular Exception level:

- Defines a register to be RO, then any attempt to write to that register, at that Exception level, is UNDEFINED. This means that any access to that register with `L==0` is UNDEFINED.
- Defines a register to be WO, then any attempt to read from that register, at that Exception level, is UNDEFINED. This means that any access to that register with `L==1` is UNDEFINED.

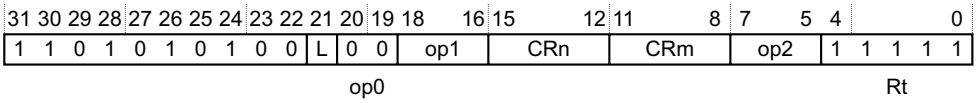
For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED, but see the recommendation in *Principles of the System instruction class encoding* on page C5-266.

## C5.1.3 `op0==0b00`, architectural hints, barriers and `CLREX`, and PSTATE access

The different groups of System register instructions with `op0==0b00`:

- Are identified by the value of `CRn`.
- Are always encoded with a value of `0b11111` in the `Rt` field.

The encoding of these instructions is:




The encoding of the `CRn` field is as follows:

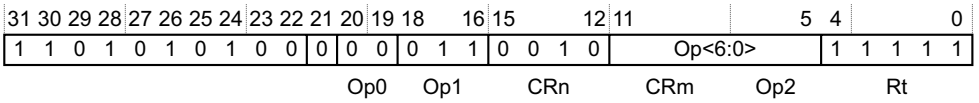
**0b0010** See *Architectural hint instructions*.

**0b0011** See *Barriers and CLREX* on page C5-269.

**0b0100** See *Instructions for accessing the PSTATE fields* on page C5-270.

### Architectural hint instructions

Within the `op0==0b00` encodings, the architectural hint instructions are identified by `CRn` having the value `0b0010`. The encoding of these instructions is:




The value of `op<6:0>`, formed by concatenating the `CRm` and `op2` fields, determines the hint instruction as follows:

`0b0000000` `NOP` instruction. This has no effect on architectural state other than to advance the PC.

`0b0000001` `YIELD` instruction.

`0b0000010` `WFE` instruction.

`0b0000011` `WFI` instruction.

`0b0000100` `SEV` instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0b0000101 SEVL instruction.

0b0000110-0b1111111
Unallocated values. These encodings behave as NOPs.

**Note**

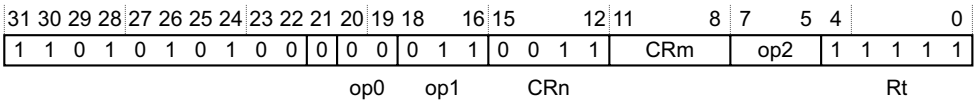
- Instruction encodings with bits[4:0] not set to 0b11111 are UNDEFINED.
- The operation of the A64 instructions for architectural hints are identical to the corresponding A32 and T32 instructions.

For more information about:

- The WFE, WFI, SEV, and SEVL instructions, see *Mechanisms for entering a low-power state* on page D1-1611.
- The YIELD instruction, see *Software control features and EL0* on page B1-64.

### Barriers and CLREX

Within the op0==0b00 encodings, the barriers and CLREX instructions are identified by CRn having the value 0b0011. The encoding of these instructions is:




The value of op2 determines the instruction, as follows. For the DSB and DMB instructions, CRm controls the instruction options.

0b010 CLREX instruction. The value of CRm is ignored.

0b100 DSB instruction. The value of CRm sets the option type, see Table C5-1.

0b101 DMB instruction. The value of CRm sets the option type, see Table C5-1.

0b110 ISB instruction. The value of CRm is ignored.

0b000, 0b001, 0b011, 0b111
UNDEFINED.

**Note**

Instruction encodings with bits[4:0] not set to 0b11111 are UNDEFINED.

Table C5-1 shows the CRm encodings for the data barrier option types.

**Table C5-1 CRm encoding for DMB and DSB instructions**

| CRm value | Option, for DMB and DSB | Meaning |
|---|---|---|
| 0001 | OSHLD | Outer Shareable, load |
| 0010 | OSHST | Outer Shareable, store |
| 0011 | OSH | Outer Shareable, all |
| 0101 | NSHLD | Non-shareable, load |
| 0110 | NSHST | Non-shareable, store |
| 0111 | NSH | Non-shareable, all |
| 1001 | ISHLD | Inner Shareable, load |
| 1010 | ISHST | Inner Shareable, store |
| 1011 | ISH | Inner Shareable, all |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C5-1 CRm encoding for DMB and DSB instructions (continued)**

| CRm value | Option, for DMB and DSB | Meaning |
|---|---|---|
| 1101 | LD | Full system, load |
| 1110 | ST | Full system, store |
| 0000, 0100, 1000, 1100 | #<imm>[a] | Full system, all |
| 1111 | SY | Full system, all |

a. #<imm> is a 4-bit unsigned immediate in the range 0-15, encoded in the CRm field.

**Note**

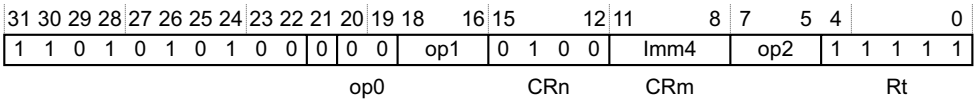
The operation of the A64 instructions for barriers and CLREX are identical to the corresponding A32 and T32 instructions.

For more information about:

- The barrier instructions, see *Memory barriers* on page B2-86.
- The CLREX instruction, see *Synchronization and semaphores* on page B2-107.

## Instructions for accessing the PSTATE fields

Within the op0==0b00 encodings, the instructions that can be used to modify PSTATE fields directly are identified by CRn having the value 0b0100. The encoding of these instructions is:




These instructions are:

```
MSR DAIFSet, #Imm4    ; Used to set any or all of DAIF to 1
MSR DAIFClr, #Imm4    ; Used to clear any or all of DAIF to 0
MSR SPSel, #Imm1      ; Used to select the Stack Pointer, between SP_EL0 and SP_ELx
```

The value of op2 selects the instruction form, which defines the constraints on the values of the op1 and Imm4 arguments, as follows:

op2==0b101 Selects the MSR SPSel instruction.

op1 must be 0b000.

This instruction is accessible at EL1 or higher.

Imm4<0> selects the accessed stack pointer, as follows:

**0** Selects SP_EL0.

**1** Selects *SP_ELx* on page K13-5712, where *x* is the number of the current Exception level, 1, 2, or 3.

Imm4<3:1> are SBZ.

op2==0b110 Selects the MSR DAIFSet instruction, that sets the specified PSTATE.{D, A, I, F} bits to 1.

op1 must be 0b011.

This instruction is accessible at EL1 or higher, and when the value of the SCTLR_EL1.UMA bit is 1 it is also accessible at EL0.

Imm4 determines which of the PSTATE.{D, A, I, F} bits are set to 1, as follows:

Imm4<3> If this bit is set to 1 then the D bit is set to 1, otherwise the D bit is not changed.

Imm4<2> If this bit is set to 1 then the A bit is set to 1, otherwise the A bit is not changed.

Imm4<1> If this bit is set to 1 then the I bit is set to 1, otherwise the I bit is not changed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`Imm4<0>` If this bit is set to 1 then the F bit is set to 1, otherwise the F bit is not changed.

`op2==0b111` Selects the `MSR DAIFClr` instruction, that clears the specified PSTATE.{D, A, I, F} bits to 0.

`op1` must be `0b011`.

This instruction is accessible at EL1 or higher, and when the value of the SCTLR_EL1.UMA bit is 1 it is also accessible at EL0.

`Imm4` determines which of the PSTATE.{D, A, I, F} bits is cleared to 0, as follows:

`Imm4<3>` If this bit is set to 1 then the D bit is cleared to 0, otherwise the D bit is not changed.

`Imm4<2>` If this bit is set to 1 then the A bit is cleared to 0, otherwise the A bit is not changed.

`Imm4<1>` If this bit is set to 1 then the I bit is cleared to 0, otherwise the I bit is not changed.

`Imm4<0>` If this bit is set to 1 then the F bit is cleared to 0, otherwise the F bit is not changed.

All other combinations of `op1` and `op2` are reserved, and the corresponding instructions are UNDEFINED.

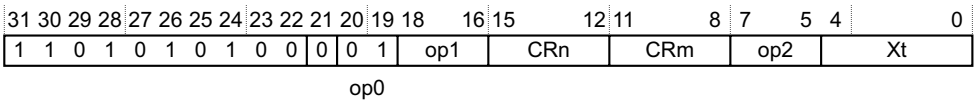
**Note**

For PSTATE updates, instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.

Writes to PSTATE.{D, A, I, F} occur in program order without the need for additional synchronization. Changing PSTATE.SPSel to use EL0 synchronizes any updates to SP_EL0 that have been written by an `MSR` to SP_EL0, without the need for additional synchronization.

### C5.1.4 `op0==0b01`, cache maintenance, TLB maintenance, and address translation instructions

The System instructions are encoded with `op0==0b01`. The different groups of System instructions are identified by the values of `CRn` and `CRm`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:




The grouping of these instructions depending on the `CRn` and `CRm` fields is as follows:

`CRn==7` The instruction group is determined by the value of `CRm`, as follows:

- `CRm=={1, 5}` Instruction cache maintenance instructions.
- `CRm==4` Data cache zero operation.
- `CRm=={6, 10, 11, 14}` Data cache maintenance instructions.

  See *Cache maintenance instructions, and data cache zero* on page C5-272.
- `CRm==8` See *Address translation instructions* on page C5-273.

`CRn==8` See *TLB maintenance instructions* on page C5-274.

`CRn=={11, 15}` See *Reserved encoding space for IMPLEMENTATION DEFINED instructions* on page C5-275.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Cache maintenance instructions, and data cache zero

Table C5-2 lists the Cache maintenance instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as `0b11111`.

**Table C5-2 Cache maintenance instructions**

| Instruction | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|
| | op0 | op1 | CRn | CRm | op2 | |
| **Instruction cache maintenance instructions** | | | | | | |
| IC IALLUIS | 1 | 0 | 7 | 1 | 0 | Accessible from EL1 or higher. |
| IC IALLU | | | | 5 | 0 | |
| IC IVAU, `Xt` | | 3 | 7 | 5 | 1 | When SCTLR_EL1.UCI == 1, accessible from EL0 or higher. Otherwise, accessible from EL1 or higher. |
| **Data cache maintenance instructions** | | | | | | |
| DC IVAC, `Xt` | 1 | 0 | 7 | 6 | 1 | Accessible from EL1 or higher. |
| DC ISW, `Xt` | | | | | 2 | |
| DC CSW, `Xt` | | | | 10 | 2 | |
| DC CISW, `Xt` | | | | 14 | 2 | |
| DC CVAC, `Xt` | | 3 | 7 | 10 | 1 | When SCTLR_EL1.UCI == 1, accessible from EL0 or higher. Otherwise, accessible from EL1 or higher. |
| DC CVAU, `Xt` | | | | 11 | 1 | |
| DC CIVAC, `Xt` | | | | 14 | 1 | |
| **Data cache zero operation** | | | | | | |
| DC ZVA, `Xt` | 1 | 3 | 7 | 4 | 1 | When SCTLR_EL1.UCI == 1, accessible from EL0 or higher. Otherwise, accessible from EL1 or higher. |

For more information about these instructions, see *Overview of the cache maintenance instructions* on page D3-1712 and *Cache maintenance instructions* on page D3-1716.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Address translation instructions

Table C5-3 lists the Address translation instructions and their encodings. The syntax of the instructions includes `Xt`, that provides the address to be translated.

**Table C5-3 Address translation instructions**

| Instruction | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|
| | op0 | op1 | CRn | CRm | op2 | |
| AT S1E1R, `Xt` | 1 | 0 | 7 | 8 | 0 | Accessible from EL1 or higher. |
| AT S1E1W, `Xt` | | | | | 1 | |
| AT S1E0R, `Xt` | | | | | 2 | |
| AT S1E0W, `Xt` | | | | | 3 | |
| AT S1E2R, `Xt` | | 4 | 7 | 8 | 0 | Accessible from EL2 or higher. |
| AT S1E2W, `Xt` | | | | | 1 | |
| AT S12E1R, `Xt` | | | | | 4 | |
| AT S12E1W, `Xt` | | | | | 5 | |
| AT S12E0R, `Xt` | | | | | 6 | |
| AT S12E0W, `Xt` | | | | | 7 | |
| AT S1E3R, `Xt` | | 6 | 7 | 8 | 0 | Accessible only from EL3. |
| AT S1E3W, `Xt` | | | | | 1 | |

For more information about these instructions, see *Address translation instructions* on page D4-1795.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### TLB maintenance instructions

Table C5-4 lists the TLB maintenance instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as `0b11111`.

**Table C5-4 TLB maintenance instructions**

| Instruction | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|
| | op0 | op1 | CRn | CRm | op2 | |
| TLBI VMALLE1IS | 1 | 0 | 8 | 3 | 0 | Accessible from EL1 or higher. |
| TLBI VAE1IS, Xt | | | | | 1 | |
| TLBI ASIDE1IS, Xt | | | | | 2 | |
| TLBI VAAE1IS, Xt | | | | | 3 | |
| TLBI VALE1IS, Xt | | | | | 5 | |
| TLBI VAALE1IS, Xt | | | | | 7 | |
| TLBI VMALLE1 | | | | 7 | 0 | Accessible from EL1 or higher. |
| TLBI VAE1, Xt | | | | | 1 | |
| TLBI ASIDE1, Xt | | | | | 2 | |
| TLBI VAAE1, Xt | | | | | 3 | |
| TLBI VALE1, Xt | | | | | 5 | |
| TLBI VAALE1, Xt | | | | | 7 | |
| TLBI IPAS2E1IS, Xt | 1 | 4 | 8 | 0 | 1 | Accessible from EL2 or higher. |
| TLBI IPAS2LE1IS, Xt | | | | | 5 | |
| TLBI ALLE2IS | | | | 3 | 0 | |
| TLBI VAE2IS, Xt | | | | | 1 | |
| TLBI ALLE1IS | | | | | 4 | |
| TLBI VALE2IS, Xt | | | | | 5 | |
| TLBI VMALLS12E1IS | | | | | 6 | |
| TLBI IPAS2E1, Xt | | | | 4 | 1 | |
| TLBI IPAS2LE1, Xt | | | | | 5 | |
| TLBI ALLE2 | | | | 7 | 0 | |
| TLBI VAE2, Xt | | | | | 1 | |
| TLBI ALLE1 | | | | | 4 | |
| TLBI VALE2, Xt | | | | | 5 | |
| TLBI VMALLS12E1 | | | | | 6 | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

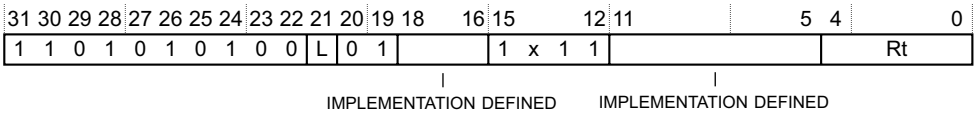
**Table C5-4 TLB maintenance instructions (continued)**

| Instruction | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|
| | op0 | op1 | CRn | CRm | op2 | |
| TLBI ALLE3IS | 1 | 6 | 8 | 3 | 0 | Accessible only from EL3. |
| TLBI VAE3IS, Xt | | | | | 1 | |
| TLBI VALE3IS, Xt | | | | | 5 | |
| TLBI ALLE3 | | | | 7 | 0 | |
| TLBI VAE3, Xt | | | | | 1 | |
| TLBI VALE3, Xt | | | | | 5 | |

For more information about these instructions, see *TLB maintenance instructions* on page D4-1839.

### Reserved encoding space for IMPLEMENTATION DEFINED instructions

The A64 instruction set reserves the following encoding space for IMPLEMENTATION DEFINED instructions:




The value of L defines the use of Rt as follows:

**0** Rt is an argument supplied to the instruction.

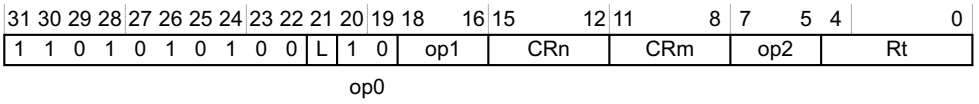
**1** Rt is a result returned by the instruction.

IMPLEMENTATION DEFINED instructions in this encoding space are accessed using the SYS and SYSL instructions, see *SYS* on page C6-807 and *SYSL* on page C6-809.

See also *Reserved encodings for IMPLEMENTATION DEFINED registers* on page C5-288.

## C5.1.5 op0==0b10, Moves to and from debug and trace System registers

The instructions that move data to and from the debug, Execution environment, and trace System registers are encoded with op0==0b10. This means the encoding of these instructions is:




**Note**

These encodings access the registers that are equivalent to the AArch32 CP14 registers.

The value of op1 provides the next level of decode of these instructions, as follows:

**op1 == {0, 3, 4}**

Debug. See *Instructions for accessing debug System registers* on page C5-276

**Note**

The standard encoding of debug registers is op0==0b10, op1=={0, 3, 4}. The following sections describe registers in the op0==0b10 encoding space that are classified as debug registers, see *Instructions for accessing non-debug System registers* on page C5-278:

- *MDCR_EL2, Monitor Debug Configuration Register (EL2)* on page D7-2208.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *MDCR_EL3, Monitor Debug Configuration Register (EL3)* on page D7-2212.
- *SDER32_EL3, AArch32 Secure Debug Enable Register* on page D7-2234.

`op1` == **1** Trace. See the appropriate trace architecture specification.

### Instructions for accessing debug System registers

The instructions for accessing debug System registers are:

```
MSR <System register>, Xt    ; Write to System register
MRS Xt, <System register>    ; Read from System register
```

Where `<System_register>` is the register name, for example MDCCSR_EL0.

This section includes only the System register access encodings for which both:

- `op0` is `0b10`.
- The value of `op1` is one of {0, 3, 4}.

**Note**

These encodings access the registers that are equivalent to the AArch32 CP14 registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table C5-5 shows the mapping of the System register encodings for debug System register access.

**Table C5-5 System instruction encodings for debug System register access**

| Register | Access instruction encoding | | | | | Permitted accesses |
|---|---|---|---|---|---|---|
| | op0 | op1 | CRn | CRm | op2 | |
| OSDTRRX_EL1 | 2 | 0 | 0 | 0 | 2 | RW |
| MDCCINT_EL1 | | | | 2 | 0 | RW |
| MDSCR_EL1 | | | | | 2 | RW |
| OSDTRTX_EL1 | | | | 3 | 2 | RW |
| OSECCR_EL1 | | | | 6 | 2 | RW |
| DBGBVR<n>_EL1 | | | | 0-15[a] | 4 | RW |
| DBGBCR<n>_EL1 | | | | 0-15[a] | 5 | RW |
| DBGWVR<n>_EL1 | | | | 0-15[a] | 6 | RW |
| DBGWCR<n>_EL1 | | | | 0-15[a] | 7 | RW |
| MDRAR_EL1 | | | 1 | 0 | 0 | RO |
| OSLAR_EL1 | | | | | 4 | WO |
| OSLSR_EL1 | | | | 1 | 4 | RO |
| OSDLR_EL1 | | | | 3 | 4 | RW |
| DBGPRCR_EL1 | | | | 4 | 4 | RW |
| DBGCLAIMSET_EL1 | | | 7 | 8 | 6 | RW |
| DBGCLAIMCLR_EL1 | | | | 9 | 6 | RW |
| DBGAUTHSTATUS_EL1 | | | | 14 | 6 | RO |
| MDCCSR_EL0 | | 3 | 0 | 1 | 0 | RO |
| DBGDTR_EL0 | | | | 4 | 0 | RW |
| DBGDTRRX_EL0 | | | | 5 | 0 | RO |
| DBGDTRTX_EL0 | | | | | | WO |
| DBGVCR32_EL2 | | 4 | 0 | 7 | 0 | RW |

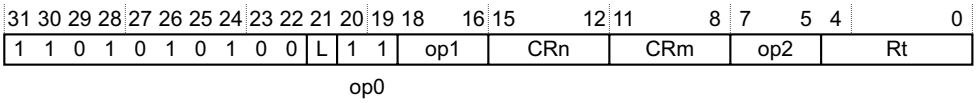
a. Unimplemented breakpoint and watchpoint register access instructions are unallocated. CRm encodes <*n*>, the breakpoint or watchpoint number.

For more information see *Mapping of the System registers between the Execution states* on page D1-1622.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.1.6 op0==0b11, Moves to and from non-debug System registers and Special-purpose registers

The instructions that move data to and from non-debug System registers are encoded with op0==0b11, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:




The value of CRn provides the next level of decode of these instructions, as follows:

**CRn==0b0000, 0b0001, 0b0010, 0b0011, 0b0101, 0b0110, 0b0111, 0b1001, 0b1010, 0b1100, 0b1101, 0b1110**

See *Instructions for accessing non-debug System registers*.

**CRn==0b0100** See *Instructions for accessing Special-purpose registers* on page C5-287.

**CRn==0b1011, 0b1111** See *Reserved encodings for IMPLEMENTATION DEFINED registers* on page C5-288.

#### Instructions for accessing non-debug System registers

The A64 instructions for accessing System registers are:

```
MSR <System register>, Xt    ; Write to System register
MRS Xt, <System register>    ; Read from System register
```

Where <System_register> is the register name, for example MIDR_EL1.

This section includes only the System register access encodings for which both:

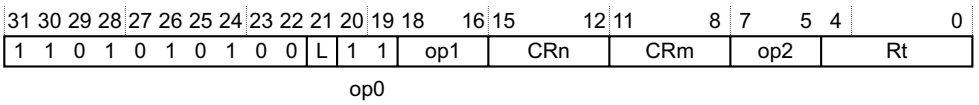
- op0 is 0b11.
- The value of CRn is one of {0, 1, 2, 3, 5, 6, 7, 9, 10, 12, 13, 14}.

**Note**

- These encodings access the registers that are equivalent to the AArch32 CP15 registers.
- While this group is described as accessing the non-debug System registers, its correct characterization is by the {op0, CRn} values given in this subsection, and the group includes the debug registers described in the following sections:
  - *MDCR_EL2, Monitor Debug Configuration Register (EL2)* on page D7-2208.
  - *MDCR_EL3, Monitor Debug Configuration Register (EL3)* on page D7-2212.
  - *SDER32_EL3, AArch32 Secure Debug Enable Register* on page D7-2234.

  These registers are exceptions to the standard encoding of debug registers, that has op0==0b10, see *Instructions for accessing debug System registers* on page C5-276.

The instruction encoding for these accesses is:




See text for permitted values of CRn

Table C5-6 on page C5-279 shows the encodings of the register access instructions. In the *Notes* on page C5-279 column of the table:

**Config-RO** Means it is configurable whether read accesses are permitted. Write accesses are UNDEFINED.

**Config-WO** Means it is configurable whether write accesses are permitted. Read accesses are UNDEFINED.

**Config-RW** Means it is configurable whether accesses are permitted. Either read and write accesses are permitted, or read and write accesses are UNDEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See the register descriptions for information about the control that determines whether these accesses are permitted.

**Table C5-6 System instruction encodings for non-Debug System register accesses**

| Register accessed | Width (bits) | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|---|
| | | op0 | op1 | CRn | CRm | op2 | |
| MIDR_EL1 | 32 | 3 | 0 | 0 | 0 | 0 | RO. |
| MPIDR_EL1 | 64 | | | | | 5 | RO. |
| REVIDR_EL1 | 32 | | | | | 6 | RO. |
| ID_PFR0_EL1 | 32 | | | | 1 | 0 | RO, but UNKNOWN if AArch32 is not implemented. |
| ID_PFR1_EL1 | 32 | | | | | 1 | |
| ID_DFR0_EL1 | 32 | | | | | 2 | |
| ID_AFR0_EL1 | 32 | | | | | 3 | |
| ID_MMFR0_EL1 | 32 | | | | | 4 | |
| ID_MMFR1_EL1 | 32 | | | | | 5 | |
| ID_MMFR2_EL1 | 32 | | | | | 6 | |
| ID_MMFR3_EL1 | 32 | | | | | 7 | |
| ID_ISAR0_EL1 | 32 | | | | 2 | 0 | RO, but UNKNOWN if AArch32 is not implemented. |
| ID_ISAR1_EL1 | 32 | | | | | 1 | |
| ID_ISAR2_EL1 | 32 | | | | | 2 | |
| ID_ISAR3_EL1 | 32 | | | | | 3 | |
| ID_ISAR4_EL1 | 32 | | | | | 4 | |
| ID_ISAR5_EL1 | 32 | | | | | 5 | |
| ID_MMFR4_EL1 | 32 | | | | | 6 | |
| Reserved, RAZ | - | | | | | 7 | RO. |
| MVFR0_EL1 | 32 | | | | 3 | 0 | RO, but UNKNOWN if AArch32 is not implemented. |
| MVFR1_EL1 | 32 | | | | | 1 | |
| MVFR2_EL1 | 32 | | | | | 2 | |
| Reserved, RAZ | - | | | | | *n* | RO, for *n*=3-7. |
| ID_AA64PFR0_EL1 | 64 | | | | 4 | 0 | RO. |
| ID_AA64PFR1_EL1 | 64 | | | | | 1 | RO. |
| Reserved, RAZ | - | | | | | *n* | RO, for *n*=2-7. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C5-6 System instruction encodings for non-Debug System register accesses (continued)**

| Register accessed | Width (bits) | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|---|
| | | op0 | op1 | CRn | CRm | op2 | |
| ID_AA64DFR0_EL1 | 64 | 3 | 0 | 0 | 5 | 0 | RO. |
| ID_AA64DFR1_EL1 | 64 | | | | | 1 | RO. |
| ID_AA64AFR0_EL1 | 64 | | | | | 4 | RO. |
| ID_AA64AFR1_EL1 | 64 | | | | | 5 | RO. |
| Reserved, RAZ | - | | | | | *n* | RO, for *n*={2, 3, 6, 7}. |
| ID_AA64ISAR0_EL1 | 64 | | | | 6 | 0 | RO. |
| ID_AA64ISAR1_EL1 | 64 | | | | | 1 | RO. |
| Reserved, RAZ | - | | | | | *n* | RO, for *n*=2-7. |
| ID_AA64MMFR0_EL1 | 64 | | | | 7 | 0 | RO. |
| ID_AA64MMFR1_EL1 | 64 | | | | | 1 | RO. |
| Reserved, RAZ | - | | | | | *n* | RO, for *n*=2-7. |
| SCTLR_EL1 | 32 | | | 1 | 0 | 0 | RW. |
| ACTLR_EL1 | 64 | | | | | 1 | RW, contents IMPLEMENTATION DEFINED. |
| CPACR_EL1 | 32 | | | | | 2 | RW. |
| TTBR0_EL1 | 64 | | | 2 | 0 | 0 | RW. |
| TTBR1_EL1 | 64 | | | | | 1 | RW. |
| TCR_EL1 | 64 | | | | | 2 | RW. |
| ICC_PMR_EL1<br>ICV_PMR_EL1 | 32 | | | 4 | 6 | 0 | RW.[a] |
| AFSR0_EL1 | 32 | | | 5 | 1 | 0 | RW, contents IMPLEMENTATION DEFINED. |
| AFSR1_EL1 | 32 | | | | | 1 | |
| ESR_EL1 | 32 | | | | 2 | 0 | RW. |
| FAR_EL1 | 64 | | | 6 | 0 | 0 | RW. |
| PAR_EL1 | 64 | | | 7 | 4 | 0 | RW. |
| PMINTENSET_EL1 | 32 | | | 9 | 14 | 1 | RW.[b] |
| PMINTENCLR_EL1 | 32 | | | | | 2 | RW.[b] |
| MAIR_EL1 | 64 | | | 10 | 2 | 0 | RW. |
| AMAIR_EL1 | 64 | | | | 3 | 0 | RW, contents IMPLEMENTATION DEFINED. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C5-6 System instruction encodings for non-Debug System register accesses (continued)**

| Register accessed | Width (bits) | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|---|
| | | op0 | op1 | CRn | CRm | op2 | |
| VBAR_EL1 | 64 | 3 | 0 | 12 | 0 | 0 | RW. |
| RVBAR_EL1 | 64 | | | | | 1 | RO. Implemented only if EL2 and EL3 are not implemented. |
| RMR_EL1 | 64 | | | | | 2 | RW. Implemented only when both:<br>• EL1 can use AArch32 and AArch64<br>• EL2 and EL3 are not implemented. |
| ISR_EL1 | 32 | | | | 1 | 0 | RO. |
| ICC_IAR0_EL1<br>ICV_IAR0_EL1 | 32 | | | | 8 | 0 | RO.[a] |
| ICC_EOIR0_EL1<br>ICV_EOIR0_EL1 | 32 | | | | | 1 | WO.[a] |
| ICC_HPPIR0_EL1<br>ICV_HPPIR0_EL1 | 32 | | | | | 2 | RO.[a] |
| ICC_BPR0_EL1<br>ICV_BPR0_EL1 | 32 | | | | | 3 | RW.[a] |
| ICC_AP0R<n>_EL1<br>ICV_AP0R<n>_EL1 | 32 | | | | | 4-7 | RW, <n> = `op2-4`.[a] |
| ICC_AP1R<n>_EL1<br>ICV_AP1R<n>_EL1 | 32 | | | | 9 | 0-3 | RW, <n> = `op2`.[a] |
| ICC_DIR_EL1<br>ICV_DIR_EL1 | 32 | | | | 11 | 1 | WO.[a] |
| ICC_RPR_EL1<br>ICV_RPR_EL1 | 32 | | | | | 3 | RO.[a] |
| ICC_SGI1R_EL1 | 64 | | | | | 5 | WO.[a] |
| ICC_ASGI1R_EL1 | 64 | | | | | 6 | WO.[a] |
| ICC_SGI0R_EL1 | 64 | | | | | 7 | WO.[a] |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C5-6 System instruction encodings for non-Debug System register accesses (continued)**

| Register accessed | Width (bits) | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|---|
| | | op0 | op1 | CRn | CRm | op2 | |
| ICC_IAR1_EL1<br>ICV_IAR1_EL1 | 32 | 3 | 0 | 12 | 12 | 0 | RO.[a] |
| ICC_EOIR1_EL1<br>ICV_EOIR1_EL1 | 32 | | | | | 1 | WO.[a] |
| ICC_HPPIR1_EL1<br>ICV_HPPIR1_EL1 | 32 | | | | | 2 | RO.[a] |
| ICC_BPR1_EL1<br>ICV_BPR1_EL1 | 32 | | | | | 3 | RW.[a] |
| ICC_CTLR_EL1<br>ICV_CTLR_EL1 | 32 | | | | | 4 | RW.[a] |
| ICC_SRE_EL1 | 32 | | | | | 5 | RW.[a] |
| ICC_IGRPEN0_EL1 | 32 | | | | | 6 | RW.[a] |
| ICC_IGRPEN1_EL1 | 32 | | | | | 7 | RW.[a] |
| ICC_IGRPEN0_EL1<br>ICV_IGRPEN0_EL1 | 32 | | | | | 6 | RW.[a] |
| ICC_IGRPEN1_EL1<br>ICV_IGRPEN1_EL1 | 32 | | | | | 7 | RW.[a] |
| CONTEXTIDR_EL1 | 32 | | | 13 | 0 | 1 | RW. |
| TPIDR_EL1 | 64 | | | | | 4 | RW. |
| CNTKCTL_EL1 | 32 | | | 14 | 1 | 0 | RW.[c] |
| CCSIDR_EL1 | 32 | | 1 | 0 | 0 | 0 | RO. |
| CLIDR_EL1 | 32 | | | | | 1 | RO. |
| AIDR_EL1 | 32 | | | | | 7 | RO. |
| CSSELR_EL1 | 32 | | 2 | 0 | 0 | 0 | RW. |
| CTR_EL0 | 32 | | 3 | 0 | 0 | 1 | Config-RO at EL0, otherwise RO. |
| DCZID_EL0 | 32 | | | | | 7 | RO. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C5-6 System instruction encodings for non-Debug System register accesses (continued)**

| Register accessed | Width (bits) | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|---|
| | | op0 | op1 | CRn | CRm | op2 | |
| PMCR_EL0 | 32 | 3 | 3 | 9 | 12 | 0 | Config-RW at EL0, otherwise RW.[b] |
| PMCNTENSET_EL0 | 32 | | | | | 1 | |
| PMCNTENCLR_EL0 | 32 | | | | | 2 | |
| PMOVSCLR_EL0 | 32 | | | | | 3 | |
| PMSWINC_EL0 | 32 | | | | | 4 | Config-WO at EL0, otherwise WO.[b] |
| PMSELR_EL0 | 32 | | | | | 5 | Config-RW at EL0, otherwise RW.[b] |
| PMCEID0_EL0 | 32 | | | | | 6 | Config-RO at EL0, otherwise RO.[b] |
| PMCEID1_EL0 | 32 | | | | | 7 | |
| PMCCNTR_EL0 | 64 | | | | 13 | 0 | Config-RW at EL0, otherwise RW.[b] |
| PMXEVTYPER_EL0 | 32 | | | | | 1 | |
| PMXEVCNTR_EL0 | 32 | | | | | 2 | |
| PMUSERENR_EL0 | 32 | | | | 14 | 0 | RO at EL0, RW at other Exception levels.[b] |
| PMOVSSET_EL0 | 32 | | | | | 3 | Config-RW at EL0, otherwise RW.[b] |
| TPIDR_EL0 | 64 | | | 13 | 0 | 2 | RW. |
| TPIDRRO_EL0 | 64 | | | | | 3 | RW. |
| CNTFRQ_EL0 | 32 | | | 14 | 0 | 0 | RO at EL1, RW at the highest Exception level implemented. Config-RO at EL0.[c] |
| CNTPCT_EL0 | 64 | | | | | 1 | Config-RO at EL0, otherwise RO.[c] |
| CNTVCT_EL0 | 64 | | | | | 2 | |
| CNTP_TVAL_EL0 | 32 | | | | 2 | 0 | Config-RW at EL0 and Non-secure EL1, otherwise RW.[c] |
| CNTP_CTL_EL0 | 32 | | | | | 1 | |
| CNTP_CVAL_EL0 | 64 | | | | | 2 | |
| CNTV_TVAL_EL0 | 32 | | 3 | 14 | 3 | 0 | Config-RW at EL0, otherwise RW.[c] |
| CNTV_CTL_EL0 | 32 | | | | | 1 | |
| CNTV_CVAL_EL0 | 64 | | | | | 2 | |
| PMEVCNTR<n>_EL0 | 32 | | | 14 | {8-10} | {0-7} | RW. `CRm` and `op2` encode <n>, the counter number[b]:<br>• For `CRm`=={8, 12}, <n>=`op2`.<br>• For `CRm`=={9, 13}, <n>=`op2`+8.<br>• For `CRm`=={10, 14}, <n>=`op2`+16.<br>• For `CRm`=={11, 15}, <n>=`op2`+24.<br>Config-RW at EL0, otherwise RW. |
| | | | | | 11 | {0-6} | |
| PMEVTYPER<n>_EL0 | 32 | | | | {12-14} | {0-7} | |
| | | | | | 15 | {0-6} | |
| PMCCFILTR_EL0 | 32 | | | | | 7 | Config-RW at EL0, otherwise RW.[b] |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C5-6 System instruction encodings for non-Debug System register accesses (continued)**

| Register accessed | Width (bits) | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|---|
| | | op0 | op1 | CRn | CRm | op2 | |
| VPIDR_EL2 | 32 | 3 | 4 | 0 | 0 | 0 | RW. |
| VMPIDR_EL2 | 64 | | | | | 5 | RW. |
| SCTLR_EL2 | 32 | | | 1 | 0 | 0 | RW. |
| ACTLR_EL2 | 64 | | | | | 1 | RW, contents IMPLEMENTATION DEFINED. |
| HCR_EL2 | 64 | | | | 1 | 0 | RW. |
| MDCR_EL2 | 32 | | | | | 1 | RW.[d] |
| CPTR_EL2 | 32 | | | | | 2 | RW. |
| HSTR_EL2 | 32 | | | | | 3 | RW. |
| HACR_EL2 | 32 | | | | | 7 | RW, contents IMPLEMENTATION DEFINED. |
| TTBR0_EL2 | 64 | | | 2 | 0 | 0 | RW. |
| TCR_EL2 | 32 | | | | | 2 | RW. |
| VTTBR_EL2 | 64 | | | | 1 | 0 | RW. |
| VTCR_EL2 | 32 | | | | | 2 | RW. |
| DACR32_EL2 | 32 | | | 3 | 0 | 0 | RW. If EL1 cannot use AArch32, the register is UNDEFINED.[e] |
| IFSR32_EL2 | 32 | | | 5 | 0 | 1 | RW. If EL1 cannot use AArch32, the register is UNDEFINED.[e] |
| AFSR0_EL2 | 32 | | | | 1 | 0 | RW, contents IMPLEMENTATION DEFINED. |
| AFSR1_EL2 | 32 | | | | | 1 | RW, contents IMPLEMENTATION DEFINED. |
| ESR_EL2 | 32 | | | | 2 | 0 | RW. |
| FPEXC32_EL2 | 32 | | | | 3 | 0 | RW. If EL1 cannot use AArch32, the register is UNDEFINED.[e] |
| FAR_EL2 | 64 | | | 6 | 0 | 0 | RW. |
| HPFAR_EL2 | 64 | | | | | 4 | RW. |
| MAIR_EL2 | 64 | | | 10 | 2 | 0 | RW. |
| AMAIR_EL2 | 64 | | | | 3 | 0 | RW, contents IMPLEMENTATION DEFINED. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C5-6 System instruction encodings for non-Debug System register accesses (continued)**

| Register accessed | Width (bits) | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|---|
| | | op0 | op1 | CRn | CRm | op2 | |
| VBAR_EL2 | 64 | 3 | 4 | 12 | 0 | 0 | RW. |
| RVBAR_EL2 | 64 | | | | | 1 | RO. Implemented only if EL3 is not implemented. |
| RMR_EL2 | 64 | | | | | 2 | RW. Implemented only if both:<br>• EL2 can use AArch32 and AArch64<br>• EL3 is not implemented. |
| ICH_AP0R<n>_EL2 | 32 | | | | 8 | 0-3 | RW, <n>=op2.[a] |
| ICH_AP1R<n>_EL2 | 32 | | | | 9 | 0-3 | RW, <n>=op2.[a] |
| ICC_SRE_EL2 | 32 | | | | | 5 | RW.[a] |
| ICH_HCR_EL2 | 32 | | | | 11 | 0 | RW.[a] |
| ICH_VTR_EL2 | 32 | | | | | 1 | RO.[a] |
| ICH_MISR_EL2 | 32 | | | | | 2 | RO.[a] |
| ICH_EISR_EL2 | 32 | | | | | 3 | RO.[a] |
| ICH_ELRSR_EL2 | 32 | | | | | 5 | RO.[a] |
| ICH_VMCR_EL2 | 32 | | | | | 7 | RW.[a] |
| ICH_LR<n>_EL2 | 64 | | | | 12, 13 | 0-7 | RW[a]:<br>• For CRm==12, <n>=op2.<br>• For CRm==13, <n>=op2+8. |
| TPIDR_EL2 | 64 | | | 13 | 0 | 2 | RW. |
| CNTVOFF_EL2 | 64 | | | 14 | 0 | 3 | RW.[c] |
| CNTHCTL_EL2 | 32 | | | | 1 | 0 | RW.[c] |
| CNTHP_TVAL_EL2 | 32 | | | | 2 | 0 | RW.[c] |
| CNTHP_CTL_EL2 | 32 | | | | | 1 | RW.[c] |
| CNTHP_CVAL_EL2 | 64 | | | | | 2 | RW.[c] |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table C5-6 System instruction encodings for non-Debug System register accesses (continued)**

| Register accessed | Width (bits) | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|---|
| | | op0 | op1 | CRn | CRm | op2 | |
| SCTLR_EL3 | 32 | 3 | 6 | 1 | 0 | 0 | RW. |
| ACTLR_EL3[63:0] | 64 | | | | | 1 | RW, contents IMPLEMENTATION DEFINED. |
| SCR_EL3 | 32 | | | | 1 | 0 | RW. |
| SDER32_EL3 | 32 | | | | 1 | 1 | RW. If EL1 cannot use AArch32, the register is UNDEFINED. [d, e] |
| CPTR_EL3 | 32 | | | | 1 | 2 | RW. |
| MDCR_EL3 | 32 | | | | 3 | 1 | RW. [d] |
| TTBR0_EL3 | 64 | | | 2 | 0 | 0 | RW. |
| TCR_EL3 | 32 | | | | | 2 | RW. |
| AFSR0_EL3 | 32 | | | 5 | 1 | 0 | RW, contents IMPLEMENTATION DEFINED. |
| AFSR1_EL3 | 32 | | | | | 1 | RW, contents IMPLEMENTATION DEFINED. |
| ESR_EL3 | 32 | | | | 2 | 0 | RW. |
| FAR_EL3 | 64 | | | 6 | 0 | 0 | RW. |
| MAIR_EL3 | 64 | | | 10 | 2 | 0 | RW. |
| AMAIR_EL3 | 64 | | | | 3 | 0 | RW, contents IMPLEMENTATION DEFINED. |
| VBAR_EL3 | 64 | | | 12 | 0 | 0 | RW. |
| RVBAR_EL3 | 64 | | | | | 1 | RO. |
| RMR_EL3 | 64 | | | | | 2 | RW. Implemented only if EL3 can use both AArch32 and AArch64. |
| ICC_CTLR_EL3 | 32 | | | | 12 | 4 | RW. [a] |
| ICC_SRE_EL3 | 32 | | | | | 5 | RW. [a] |
| ICC_IGRPEN1_EL3 | 32 | | | | | 7 | RW. [a] |
| TPIDR_EL3 | 64 | | | 13 | 0 | 2 | RW. |
| CNTPS_TVAL_EL1 | 32 | | 7 | 14 | 2 | 0 | RW at EL3, Config-RW at Secure EL1. [c] |
| CNTPS_CTL_EL1 | 32 | | | | | 1 | |
| CNTPS_CVAL_EL1 | 64 | | | | | 2 | |

a. GIC System register, see *About the GIC System registers* on page C5-287 As that subsection describes, each ICV_* register uses the same encoding as the corresponding ICC_* register.
b. Performance Monitors Extension System register, see *Performance Monitors registers* on page D7-2236.
c. Generic Timer System register, see *Generic Timer registers* on page D7-2276.
d. Debug register in the `op0`==3 encoding space, see *Debug registers* on page D7-2165.
e. Defined to allow access from AArch64 state to registers that are only used in AArch32 state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *About the GIC System registers*

From version 3.0 of the GIC architecture specification, the specification defines three groups of System registers, identified by the prefix of the register name:

**ICC_** GIC physical CPU interface System registers.

**ICH_** GIC virtual interface control System registers.

**ICV_** GIC Virtual CPU interface System registers.

**Note**

These registers are in addition to the GIC memory-mapped register groups GICC_, GICD_, GICH_, GICR_, GICV_, and GITS_.

When implemented, the GIC System registers form part of an ARM processor implementation, and therefore these registers are included in the register summaries. However, the registers are defined only in the GIC Architecture Specification.

As Table C5-6 on page C5-279 shows, the ICV_* registers have the same {`op0`, `op1`, `CRn`, `CRm`, `op2`} encodings as the corresponding ICC_* registers. For these encodings, GIC register configuration fields determine which register is accessed.

For more information see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).

## Instructions for accessing Special-purpose registers

The A64 instructions for accessing Special-purpose registers are:

```
MSR <Special-purpose register>, Xt    ; Write to Special-purpose register
MRS Xt, <Special-purpose register>    ; Read from Special-purpose register
```

For these accesses, `CRn` has the value 4. The encoding for Special-purpose register accesses is:

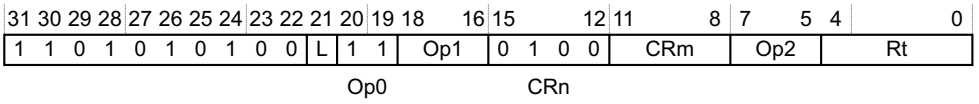



Table C5-7 lists the encodings for `op1`, `CRm`, and `op2` fields for accesses to the Special-purpose registers in AArch64.

**Table C5-7 Special-purpose register accesses**

| Register | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|
| | op0 | op1 | CRn | CRm | op2 | |
| SPSR_EL1 | 3 | 0 | 4 | 0 | 0 | Accessible from EL1 or higher. |
| ELR_EL1 | | | | | 1 | |
| SP_EL0 | | | | 1 | 0 | Accessible from EL1 or higher. If SP_EL0 is the current stack pointer then the access is UNDEFINED. |
| SPSel | | | | 2 | 0 | Accessible from EL1 or higher. |
| CurrentEL | | | | | 2 | RO. Accessible from EL1 or higher. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

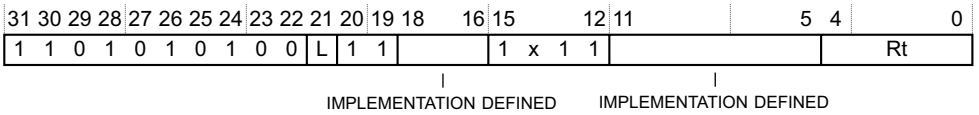
**Table C5-7 Special-purpose register accesses (continued)**

| Register | Access instruction encoding | | | | | Notes |
|---|---|---|---|---|---|---|
| | op0 | op1 | CRn | CRm | op2 | |
| DAIF | 3 | 3 | 4 | 2 | 1 | Configurable whether accesses at EL0 are permitted. |
| NZCV | | | | | 0 | Accessible from EL0 or higher. |
| FPCR | | | | 4 | 0 | Accessible from EL0 or higher. |
| FPSR | | | | | 1 | |
| DSPSR_EL0 | | | | 5 | 0 | Accessible only in Debug state, from EL0 or higher. |
| DLR_EL0 | | | | | 1 | |
| SPSR_EL2 | 3 | 4 | 4 | 0 | 0 | Accessible from EL2 or higher. |
| ELR_EL2 | | | | | 1 | |
| SP_EL1 | | | | 1 | 0 | |
| SPSR_irq | | | | 3 | 0 | |
| SPSR_abt | | | | | 1 | |
| SPSR_und | | | | | 2 | |
| SPSR_fiq | | | | | 3 | |
| SPSR_EL3 | 3 | 6 | 4 | 0 | 0 | Accessible from EL3 or higher. |
| ELR_EL3 | | | | | 1 | |
| SP_EL2 | | | | 1 | 0 | |

All direct and indirect reads and writes to Special-purpose registers appear to occur in program order relative to other instructions.

### Reserved encodings for IMPLEMENTATION DEFINED registers

The System register encoding space with `op0==0b11` reserves the following encodings for IMPLEMENTATION DEFINED registers:




The value of `L` defines the access type and the use of `Rt` as follows:

**0** Write the value in `Rt` to the IMPLEMENTATION DEFINED register.

**1** Read the value of the IMPLEMENTATION DEFINED register to `Rt`.

For more information about these encodings see *S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED registers* on page D7-2107. As that section describes, any IMPLEMENTATION DEFINED registers are accessed in a similar way to architecturally-defined System registers, using `MRS` and `MSR` instructions, see:

- *MRS* on page C6-660.
- *MSR (immediate)* on page C6-661.
- *MSR (register)* on page C6-663.

See also *Reserved encoding space for IMPLEMENTATION DEFINED instructions* on page C5-275.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C5.2 Special-purpose registers

The Special-purpose registers are:

- CurrentEL, DAIF, and NZCV, that return PSTATE information.
- The ELRs, DLR_EL0 and ELR_ELx, that hold the return address for the return from Debug state, or for the exception return.
- FPCR and FPSR, that provide floating-point status and control.
- The stack pointers, SP_ELx, and stack pointer selector, SPSel.
- The SPSRs, DSPSR_EL0 and SPSR_ELx, that hold the PE state from immediately before entering Debug state or taking an exception. This means they hold the state required for the return from Debug state, or for the exception return.
- SPSR_abt, SPSR_fiq, SPSR_irq, and SPSR_und, that map to the corresponding AArch32 registers.

**Note**

The AArch32 SPSRs SPSR_hyp, SPSR_mon, and SPSR_svc are mapped to the AArch64 SPSR_ELx registers.

The characteristic of a Special-purpose register is that all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

This section describes the following registers:

- CurrentEL, that software can read to determine the current Exception level.
- DAIF, that specifies the current interrupt mask bits.
- DLR_EL0, that holds the address to return to for a return from Debug state.
- DSPSR_EL0, that holds process state on entry to Debug state.
- ELR_EL1, that holds the address to return to for an exception return from EL1.
- ELR_EL2, that holds the address to return to for an exception return from EL2.
- ELR_EL3, that holds the address to return to for an exception return from EL3.
- FPCR, that provides control of floating-point operation.
- FPSR, that provides floating-point status information.
- NZCV, that holds the condition flags.
- SP_EL0, that holds the stack pointer for EL0.
- SP_EL1, that holds the stack pointer for EL1.
- SP_EL2, that holds the stack pointer for EL2.
- SP_EL3, that holds the stack pointer for EL3.
- SPSel, that at EL1 or higher selects between the SP for the current Exception level and SP_EL0.
- SPSR_abt, that holds process state on taking an exception to AArch32 Abort mode.
- SPSR_EL1, that holds process state on taking an exception to AArch64 EL1.
- SPSR_EL2, that holds process state on taking an exception to AArch64 EL2.
- SPSR_EL3, that holds process state on taking an exception to AArch64 EL3.
- SPSR_fiq, that holds process state on taking an exception to AArch32 FIQ mode.
- SPSR_irq, that holds process state on taking an exception to AArch32 IRQ mode.
- SPSR_und, that holds process state on taking an exception to AArch32 Undefined mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.1 CurrentEL, Current Exception Level

The CurrentEL characteristics are:

**Purpose**

Holds the current Exception level.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

A write to the CurrentEL register is UNDEFINED.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

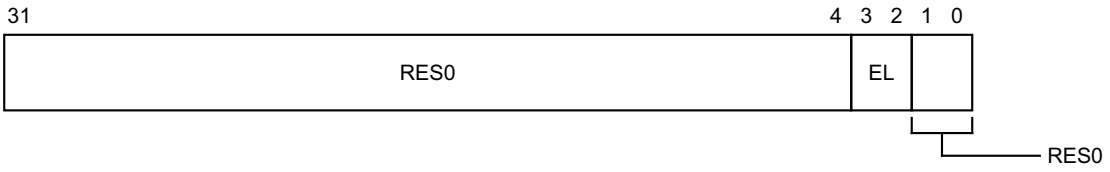
There are no configuration notes.

**Attributes**

CurrentEL is a 32-bit register.

#### Field descriptions

The CurrentEL bit assignments are:




**Bits [31:4]**

Reserved, RES0.

**EL, bits [3:2]**

Current Exception level. Possible values of this field are:

`00` EL0

`01` EL1

`10` EL2

`11` EL3

When this register has an architecturally-defined reset value, this field resets to an architecturally UNKNOWN value.

**Bits [1:0]**

Reserved, RES0.

#### Accessing the CurrentEL:

To access the CurrentEL:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRS <Xt>, CurrentEL ; Read CurrentEL into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0100 | 0010 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.2 DAIF, Interrupt Mask Bits

The DAIF characteristics are:

**Purpose**

Allows access to the interrupt mask bits.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If SCTLR_EL1.UMA==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

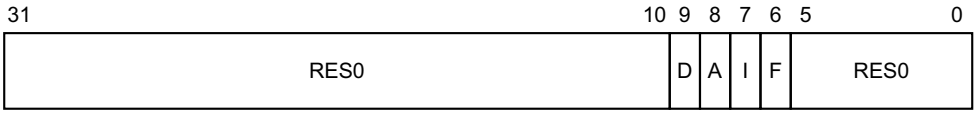
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch64. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

DAIF is a 32-bit register.

#### Field descriptions

The DAIF bit assignments are:




**Bits [31:10]**

Reserved, RES0.

**D, bit [9]**

Process state D mask. The possible values of this bit are:

`0` Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current Exception level are not masked.

`1` Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

When this register has an architecturally-defined reset value, this field resets to `1`.

**A, bit [8]**

SError (System Error) mask bit. The possible values of this bit are:

`0` Exception not masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 Exception masked.

When this register has an architecturally-defined reset value, this field resets to 1.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

When this register has an architecturally-defined reset value, this field resets to 1.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

When this register has an architecturally-defined reset value, this field resets to 1.

**Bits [5:0]**

Reserved, RES0.

### Accessing the DAIF:

To access the DAIF:

```
MRS <Xt>, DAIF ; Read DAIF into Xt
MSR DAIF, <Xt> ; Write Xt to DAIF
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 0100 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.3 DLR_EL0, Debug Link Register

The DLR_EL0 characteristics are:

**Purpose**

In Debug state, holds the address to restart from.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | RW | RW | RW | RW | RW |

Access to this register is from Debug state only. During normal execution this register is unallocated.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register DLR_EL0[31:0] is architecturally mapped to AArch32 System register DLR.

**Attributes**

DLR_EL0 is a 64-bit register.

#### Field descriptions

DLR_EL0 is a member of multiple register groups and is defined elsewhere. For the full definition, see DLR_EL0.

#### Accessing the DLR_EL0:

To access the DLR_EL0:

```
MRS <Xt>, DLR_EL0 ; Read DLR_EL0 into Xt
MSR DLR_EL0, <Xt> ; Write Xt to DLR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 0100 | 0101 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.4 DSPSR_EL0, Debug Saved Program Status Register

The DSPSR_EL0 characteristics are:

**Purpose**

Holds the saved process state on entry to Debug state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | RW | RW | RW | RW | RW |

Access to this register is from Debug state only. During normal execution this register is unallocated.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register DSPSR_EL0 is architecturally mapped to AArch32 System register DSPSR.

**Attributes**

DSPSR_EL0 is a 32-bit register.

#### Field descriptions

DSPSR_EL0 is a member of multiple register groups and is defined elsewhere. For the full definition, see DSPSR_EL0.

#### Accessing the DSPSR_EL0:

To access the DSPSR_EL0:

```
MRS <Xt>, DSPSR_EL0 ; Read DSPSR_EL0 into Xt
MSR DSPSR_EL0, <Xt> ; Write Xt to DSPSR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 0100 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.5 ELR_EL1, Exception Link Register (EL1)

The ELR_EL1 characteristics are:

**Purpose**

When taking an exception to EL1, holds the address to return to.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

An exception return from EL1 using AArch64 makes ELR_EL1 become UNKNOWN.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

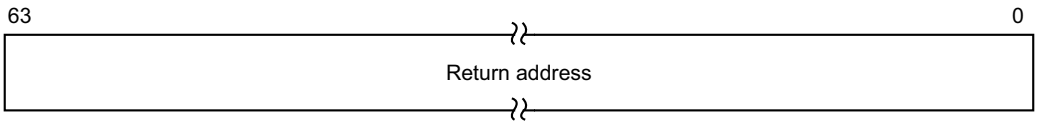
There are no configuration notes.

**Attributes**

ELR_EL1 is a 64-bit register.

#### Field descriptions

The ELR_EL1 bit assignments are:




**Bits [63:0]**

Return address.

#### Accessing the ELR_EL1:

To access the ELR_EL1:

```
MRS <Xt>, ELR_EL1 ; Read ELR_EL1 into Xt
MSR ELR_EL1, <Xt> ; Write Xt to ELR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0100 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.6 ELR_EL2, Exception Link Register (EL2)

The ELR_EL2 characteristics are:

**Purpose**

When taking an exception to EL2, holds the address to return to.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

An exception return from EL2 using AArch64 makes ELR_EL2 become UNKNOWN.

When EL2 is in AArch32 Execution state and an exception is taken from EL0, EL1, or EL2 to EL3 and AArch64 execution, the upper 32-bits of ELR_EL2 are either set to 0 or hold the same value that they did before AArch32 execution. Which option is adopted is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

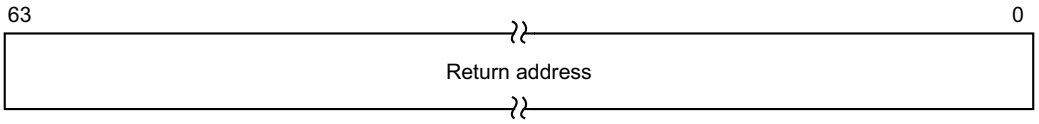
AArch64 System register ELR_EL2 is architecturally mapped to AArch32 System register ELR_hyp.

**Attributes**

ELR_EL2 is a 64-bit register.

#### Field descriptions

The ELR_EL2 bit assignments are:




**Bits [63:0]**

Return address.

#### Accessing the ELR_EL2:

To access the ELR_EL2:

```
MRS <Xt>, ELR_EL2 ; Read ELR_EL2 into Xt
MSR ELR_EL2, <Xt> ; Write Xt to ELR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0100 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.7 ELR_EL3, Exception Link Register (EL3)

The ELR_EL3 characteristics are:

**Purpose**

When taking an exception to EL3, holds the address to return to.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

An exception return from EL3 using AArch64 makes ELR_EL3 become UNKNOWN.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

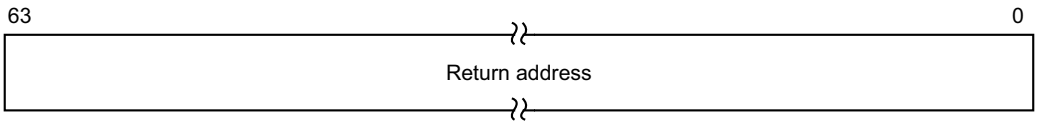
There are no configuration notes.

**Attributes**

ELR_EL3 is a 64-bit register.

#### Field descriptions

The ELR_EL3 bit assignments are:




**Bits [63:0]**

Return address.

#### Accessing the ELR_EL3:

To access the ELR_EL3:

```
MRS <Xt>, ELR_EL3 ; Read ELR_EL3 into Xt
MSR ELR_EL3, <Xt> ; Write Xt to ELR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0100 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.8 FPCR, Floating-point Control Register

The FPCR characteristics are:

**Purpose**

Controls floating-point behavior.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CPACR_EL1.FPEN==00, EL0 and EL1 accesses to this register are trapped to EL1.

If CPACR_EL1.FPEN==01, EL0 accesses to this register are trapped to EL1.

If CPACR_EL1.FPEN==10, EL0 and EL1 accesses to this register are trapped to EL1.

If CPTR_EL2.TFP==1, Non-secure EL0, EL1, and EL2 accesses to this register are trapped to EL2.

If CPTR_EL3.TFP==1, EL0, EL1, EL2, and EL3 accesses to this register are trapped to EL3.

**Configurations**

The named fields in this register map to the equivalent fields in the AArch32 FPSCR.

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to non-zero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

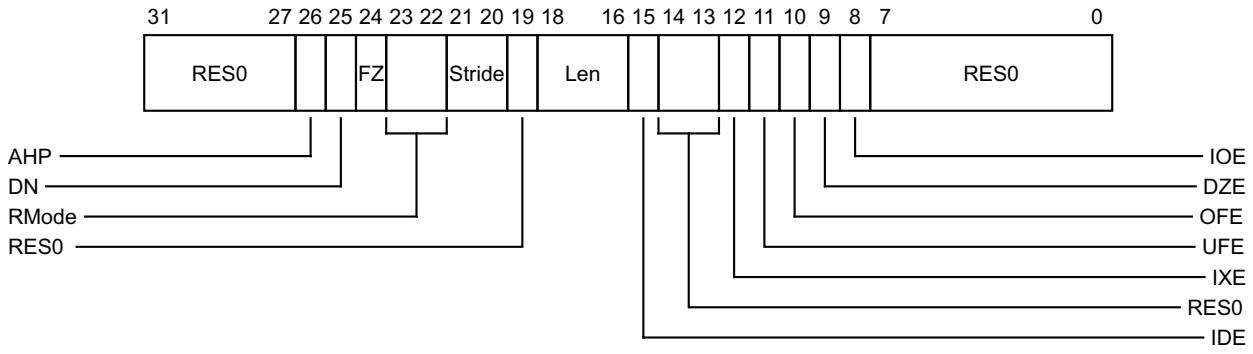
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

FPCR is a 32-bit register.

#### Field descriptions

The FPCR bit assignments are:




**Bits [31:27]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**AHP, bit [26]**

Alternative half-precision control bit:

0 IEEE half-precision format selected.

1 Alternative half-precision format selected.

**DN, bit [25]**

Default NaN mode control bit:

0 NaN operands propagate through to the output of a floating-point operation.

1 Any operation involving one or more NaNs returns the Default NaN.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

**FZ, bit [24]**

Flush-to-zero mode control bit:

0 Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.

1 Flush-to-zero mode enabled.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

**RMode, bits [23:22]**

Rounding Mode control field. The encoding of this field is:

00 Round to Nearest (RN) mode

01 Round towards Plus Infinity (RP) mode

10 Round towards Minus Infinity (RM) mode

11 Round towards Zero (RZ) mode.

The specified rounding mode is used by both scalar and Advanced SIMD floating-point instructions.

**Stride, bits [21:20]**

This field has no function in AArch64, and non-zero values are ignored during AArch64 execution. It is included only for context saving and restoration of AArch32 FPSCR.Stride.

**Bit [19]**

Reserved, RES0.

**Len, bits [18:16]**

This field has no function in AArch64, and non-zero values are ignored during AArch64 execution. It is included only for context saving and restoration of AArch32 FPSCR.Len.

**IDE, bit [15]**

Input Denormal exception trap enable. Possible values are:

0 Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.IDC bit is set to 1.

1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IDC bit. The trap handling software can decide whether to set the FPSR.IDC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RES0.

**Bits [14:13]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IXE, bit [12]**

Inexact exception trap enable. Possible values are:

`0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.IXC bit is set to 1.

`1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IXC bit. The trap handling software can decide whether to set the FPSR.IXC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RES0.

**UFE, bit [11]**

Underflow exception trap enable. Possible values are:

`0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.UFC bit is set to 1.

`1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.UFC bit. The trap handling software can decide whether to set the FPSR.UFC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RES0.

**OFE, bit [10]**

Overflow exception trap enable. Possible values are:

`0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.OFC bit is set to 1.

`1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.OFC bit. The trap handling software can decide whether to set the FPSR.OFC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RES0.

**DZE, bit [9]**

Division by Zero exception trap enable. Possible values are:

`0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.DZC bit is set to 1.

`1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.DZC bit. The trap handling software can decide whether to set the FPSR.DZC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RES0.

**IOE, bit [8]**

Invalid Operation exception trap enable. Possible values are:

`0` Untrapped exception handling selected. If the floating-point exception occurs then the FPSR.IOC bit is set to 1.

`1` Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IOC bit. The trap handling software can decide whether to set the FPSR.IOC bit to 1.

The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.

If the implementation does not support this exception, this bit is RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [7:0]**

Reserved, RES0.

### Accessing the FPCR:

To access the FPCR:

```
MRS <Xt>, FPCR ; Read FPCR into Xt
MSR FPCR, <Xt> ; Write Xt to FPCR
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 0100 | 0100 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.9 FPSR, Floating-point Status Register

The FPSR characteristics are:

**Purpose**

Provides floating-point system status information.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CPACR_EL1.FPEN==00, EL0 and EL1 accesses to this register are trapped to EL1.

If CPACR_EL1.FPEN==01, EL0 accesses to this register are trapped to EL1.

If CPACR_EL1.FPEN==10, EL0 and EL1 accesses to this register are trapped to EL1.

If CPTR_EL2.TFP==1, Non-secure EL0, EL1, and EL2 accesses to this register are trapped to EL2.

If CPTR_EL3.TFP==1, EL0, EL1, EL2, and EL3 accesses to this register are trapped to EL3.

**Configurations**

The named fields in this register map to the equivalent fields in the AArch32 FPSCR.

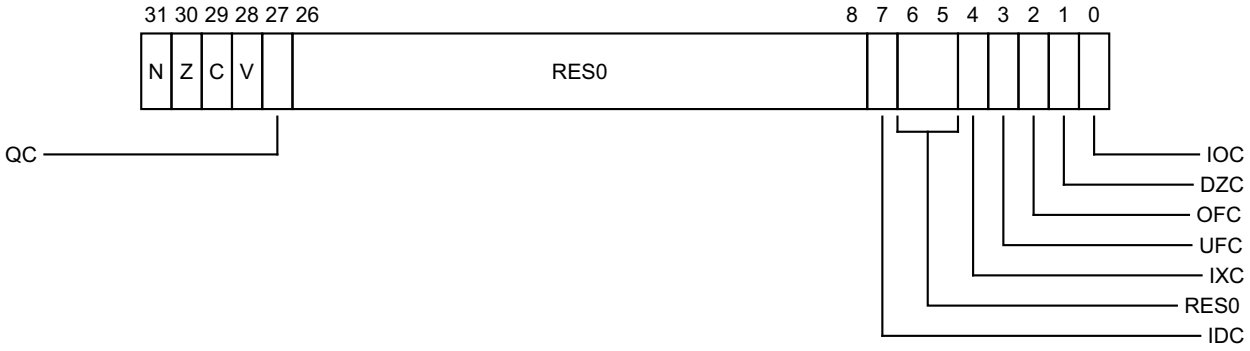
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

FPSR is a 32-bit register.

#### Field descriptions

The FPSR bit assignments are:




**N, bit [31]**

Negative condition flag for AArch32 floating-point comparison operations. AArch64 floating-point comparisons set the PSTATE.N flag instead.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Z, bit [30]**

Zero condition flag for AArch32 floating-point comparison operations. AArch64 floating-point comparisons set the PSTATE.Z flag instead.

**C, bit [29]**

Carry condition flag for AArch32 floating-point comparison operations. AArch64 floating-point comparisons set the PSTATE.C flag instead.

**V, bit [28]**

Overflow condition flag for AArch32 floating-point comparison operations. AArch64 floating-point comparisons set the PSTATE.V flag instead.

**QC, bit [27]**

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.

**Bits [26:8]**

Reserved, RES0.

**IDC, bit [7]**

Input Denormal cumulative exception bit. This bit is set to 1 to indicate that the Input Denormal exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IDE bit. This bit is only set to 1 to indicate an exception if FPCR.IDE is 0, or if trapping software sets it.

**Bits [6:5]**

Reserved, RES0.

**IXC, bit [4]**

Inexact cumulative exception bit. This bit is set to 1 to indicate that the Inexact exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IXE bit. This bit is only set to 1 to indicate an exception if FPCR.IXE is 0, or if trapping software sets it.

**UFC, bit [3]**

Underflow cumulative exception bit. This bit is set to 1 to indicate that the Underflow exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.UFE bit. This bit is only set to 1 to indicate an exception if FPCR.UFE is 0, or if trapping software sets it.

**OFC, bit [2]**

Overflow cumulative exception bit. This bit is set to 1 to indicate that the Overflow exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.OFE bit. This bit is only set to 1 to indicate an exception if FPCR.OFE is 0, or if trapping software sets it.

**DZC, bit [1]**

Division by Zero cumulative exception bit. This bit is set to 1 to indicate that the Division by Zero exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.DZE bit. This bit is only set to 1 to indicate an exception if FPCR.DZE is 0, or if trapping software sets it.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IOC, bit [0]**

Invalid Operation cumulative exception bit. This bit is set to 1 to indicate that the Invalid Operation exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IOE bit. This bit is only set to 1 to indicate an exception if FPCR.IOE is 0, or if trapping software sets it.

### Accessing the FPSR:

To access the FPSR:

```
MRS <Xt>, FPSR ; Read FPSR into Xt
MSR FPSR, <Xt> ; Write Xt to FPSR
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 0100 | 0100 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.10 NZCV, Condition Flags

The NZCV characteristics are:

**Purpose**

Allows access to the condition flags.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | RW | RW | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

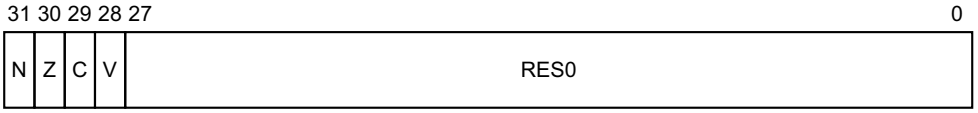
There are no configuration notes.

**Attributes**

NZCV is a 32-bit register.

#### Field descriptions

The NZCV bit assignments are:




**N, bit [31]**

Negative condition flag. Set to 1 if the result of the last flag-setting instruction was negative.

**Z, bit [30]**

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

**C, bit [29]**

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

**V, bit [28]**

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

**Bits [27:0]**

Reserved, RES0.

#### Accessing the NZCV:

To access the NZCV:

```
MRS <Xt>, NZCV ; Read NZCV into Xt
MSR NZCV, <Xt> ; Write Xt to NZCV
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 0100 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.11 SP_EL0, Stack Pointer (EL0)

The SP_EL0 characteristics are:

**Purpose**

Holds the stack pointer associated with EL0. At higher Exception levels, this is used as the current stack pointer when the value of SPSel.SP is 0.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

This accessibility information only applies when the value of SPSel.SP is 1, and only for accesses using the MRS or MSR instructions. In addition, this register is accessible at EL0 as the current stack pointer.

When the value of SPSel.SP is 0:

- Any access to SP_EL0 using the MRS or MSR instructions is UNDEFINED.
- This register is accessible at all Exception levels as the current stack pointer.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

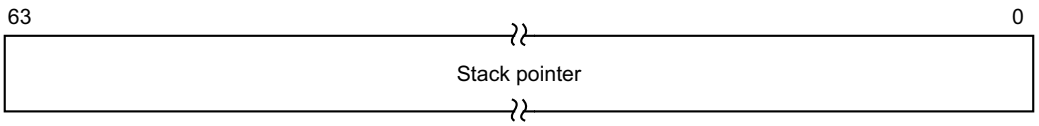
There are no configuration notes.

**Attributes**

SP_EL0 is a 64-bit register.

#### Field descriptions

The SP_EL0 bit assignments are:




**Bits [63:0]**

Stack pointer.

#### Accessing the SP_EL0:

To access the SP_EL0:

```
MRS <Xt>, SP_EL0 ; Read SP_EL0 into Xt
MSR SP_EL0, <Xt> ; Write Xt to SP_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0100 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.12 SP_EL1, Stack Pointer (EL1)

The SP_EL1 characteristics are:

**Purpose**

Holds the stack pointer associated with EL1. When executing at EL1, the value of SPSel.SP determines the current stack pointer:

| SPSel.SP | current stack pointer |
|---|---|
| 0 | SP_EL0 |
| 1 | SP_EL1 |

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL1 as the current stack pointer.

**Note**

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

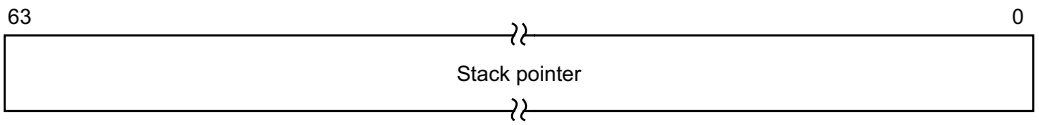
There are no configuration notes.

**Attributes**

SP_EL1 is a 64-bit register.

#### Field descriptions

The SP_EL1 bit assignments are:




**Bits [63:0]**

Stack pointer.

#### Accessing the SP_EL1:

To access the SP_EL1:

```
MRS <Xt>, SP_EL1 ; Read SP_EL1 into Xt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MSR SP_EL1, <Xt> ; Write Xt to SP_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0100 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.13 SP_EL2, Stack Pointer (EL2)

The SP_EL2 characteristics are:

**Purpose**

Holds the stack pointer associated with EL2. When executing at EL2, the value of SPSel.SP determines the current stack pointer:

| SPSel.SP | current stack pointer |
|---|---|
| 0 | SP_EL0 |
| 1 | SP_EL2 |

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL2 as the current stack pointer.

**Note**

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

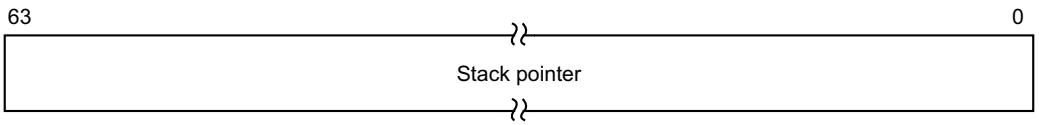
There are no configuration notes.

**Attributes**

SP_EL2 is a 64-bit register.

#### Field descriptions

The SP_EL2 bit assignments are:




**Bits [63:0]**

Stack pointer.

#### Accessing the SP_EL2:

To access the SP_EL2:

```
MRS <Xt>, SP_EL2 ; Read SP_EL2 into Xt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MSR SP_EL2, <Xt> ; Write Xt to SP_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0100 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.14 SP_EL3, Stack Pointer (EL3)

The SP_EL3 characteristics are:

**Purpose**

Holds the stack pointer associated with EL3. When executing at EL3, the value of SPSel.SP determines the current stack pointer:

| SPSel.SP | current stack pointer |
|---|---|
| 0 | SP_EL0 |
| 1 | SP_EL3 |

**Usage constraints**

This register is not accessible using MRS and MSR instructions.

When the value of SPSel.SP is 1, this register is accessible at EL3 as the current stack pointer.

**Note**

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

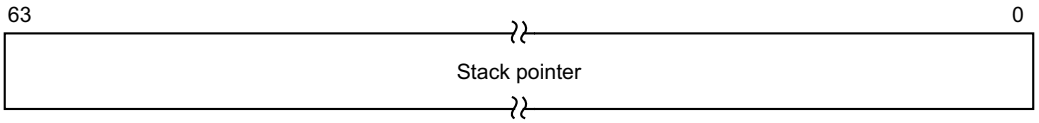
There are no configuration notes.

**Attributes**

SP_EL3 is a 64-bit register.

#### Field descriptions

The SP_EL3 bit assignments are:




**Bits [63:0]**

Stack pointer.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.15 SPSel, Stack Pointer Select

The SPSel characteristics are:

**Purpose**

Allows the Stack Pointer to be selected between SP_EL0 and SP_ELx.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

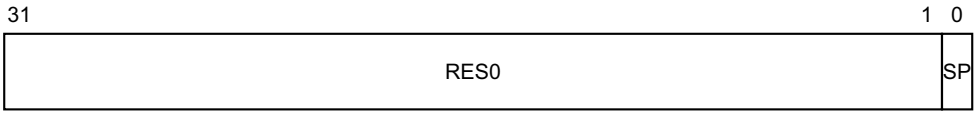
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch64. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

SPSel is a 32-bit register.

#### Field descriptions

The SPSel bit assignments are:




**Bits [31:1]**

Reserved, RES0.

**SP, bit [0]**

Stack pointer to use. Possible values of this bit are:

0 Use SP_EL0 at all Exception levels.

1 Use SP_ELx for Exception level ELx.

When this register has an architecturally-defined reset value, this field resets to 1.

#### Accessing the SPSel:

To access the SPSel:

```
MRS <Xt>, SPSel ; Read SPSel into Xt
MSR SPSel, <Xt> ; Write Xt to SPSel
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0100 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.16 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Abort mode.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register SPSR_abt is architecturally mapped to AArch32 System register SPSR_abt.

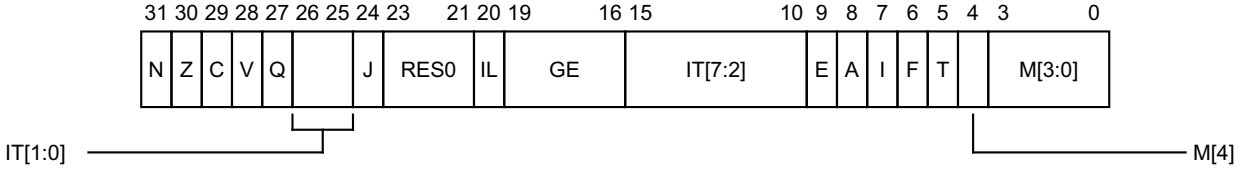
If EL1 does not support execution in AArch32, this register is RES0.

**Attributes**

SPSR_abt is a 32-bit register.

## Field descriptions

The SPSR_abt bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to Abort mode, and copied to CPSR.N on executing an exception return operation in Abort mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to Abort mode, and copied to CPSR.Z on executing an exception return operation in Abort mode.

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to Abort mode, and copied to CPSR.C on executing an exception return operation in Abort mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to Abort mode, and copied to CPSR.V on executing an exception return operation in Abort mode.

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor |
| 0b0111 | Abort |
| 0b1010 | Hyp |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved.

### Accessing the SPSR_abt:

To access the SPSR_abt:

```
MRS <Xt>, SPSR_abt ; Read SPSR_abt into Xt
MSR SPSR_abt, <Xt> ; Write Xt to SPSR_abt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0100 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.17 SPSR_EL1, Saved Program Status Register (EL1)

The SPSR_EL1 characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register SPSR_EL1 is architecturally mapped to AArch32 System register SPSR_svc.

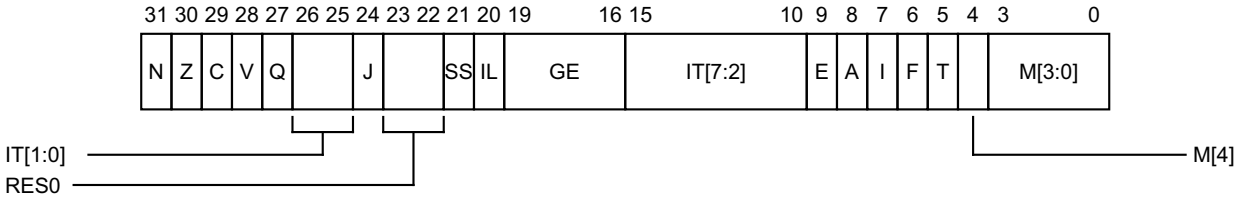
**Attributes**

SPSR_EL1 is a 32-bit register.

## Field descriptions

The SPSR_EL1 bit assignments are:

***When exception taken from AArch32:***




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to Supervisor mode, and copied to CPSR.N on executing an exception return operation in Supervisor mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to Supervisor mode, and copied to CPSR.Z on executing an exception return operation in Supervisor mode.

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to Supervisor mode, and copied to CPSR.C on executing an exception return operation in Supervisor mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to Supervisor mode, and copied to CPSR.V on executing an exception return operation in Supervisor mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:22]**

Reserved, RES0.

**SS, bit [21]**

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

`0` Taken from A32 state.

`1` Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

`1` Exception taken from AArch32.

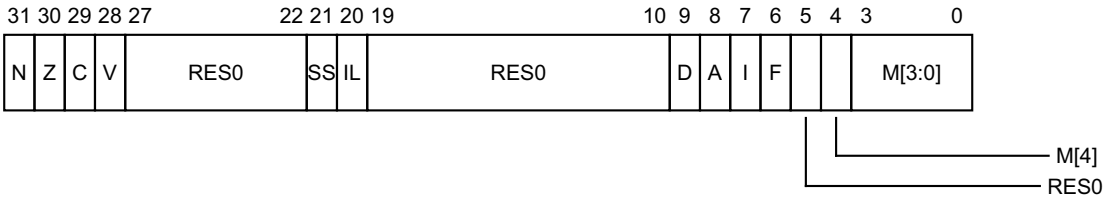
**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| `0b0000` | User |
| `0b0001` | FIQ |
| `0b0010` | IRQ |
| `0b0011` | Supervisor |
| `0b0111` | Abort |
| `0b1011` | Undefined |
| `0b1111` | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

***When exception taken from AArch64:***





*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**N, bit [31]**

Set to the value of the N condition flag on taking an exception to EL1, and copied to the N condition flag on executing an exception return operation in EL1.

**Z, bit [30]**

Set to the value of the Z condition flag on taking an exception to EL1, and copied to the Z condition flag on executing an exception return operation in EL1.

**C, bit [29]**

Set to the value of the C condition flag on taking an exception to EL1, and copied to the C condition flag on executing an exception return operation in EL1.

**V, bit [28]**

Set to the value of the V condition flag on taking an exception to EL1, and copied to the V condition flag on executing an exception return operation in EL1.

**Bits [27:22]**

Reserved, RES0.

**SS, bit [21]**

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**Bits [19:10]**

Reserved, RES0.

**D, bit [9]**

Process state D mask. The possible values of this bit are:

`0` Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current Exception level are not masked.

`1` Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

**A, bit [8]**

SError (System Error) mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**Bit [5]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

`0` Exception taken from AArch64.

**M[3:0], bits [3:0]**

AArch64 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| `0b0000` | EL0t |
| `0b0100` | EL1t |
| `0b0101` | EL1h |

Other values are reserved, and returning to an Exception level that is using AArch64 with a reserved value in this field is treated as an illegal exception return.

The bits in this field are interpreted as follows:

- M[3:2] holds the Exception Level.
- M[1] is unused and is RES0 for all non-reserved values.
- M[0] is used to select the SP:
  - — 0 means the SP is always SP0.
  - — 1 means the exception SP is determined by the EL.

### Accessing the SPSR_EL1:

To access the SPSR_EL1:

```
MRS <Xt>, SPSR_EL1 ; Read SPSR_EL1 into Xt
MSR SPSR_EL1, <Xt> ; Write Xt to SPSR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0100 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.18 SPSR_EL2, Saved Program Status Register (EL2)

The SPSR_EL2 characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register SPSR_EL2 is architecturally mapped to AArch32 System register SPSR_hyp.

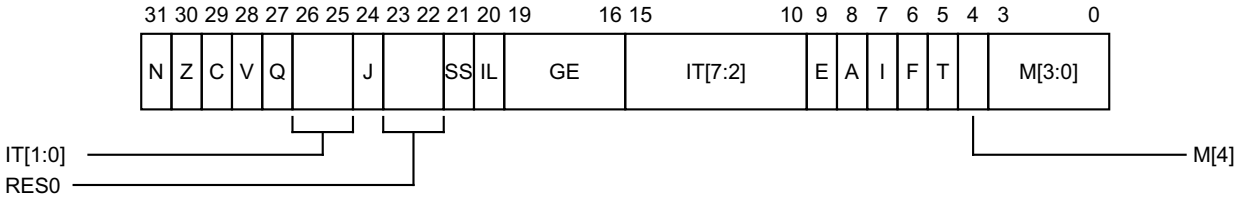
**Attributes**

SPSR_EL2 is a 32-bit register.

## Field descriptions

The SPSR_EL2 bit assignments are:

***When exception taken from AArch32:***




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to Hyp mode, and copied to CPSR.N on executing an exception return operation in Hyp mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to Hyp mode, and copied to CPSR.Z on executing an exception return operation in Hyp mode.

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to Hyp mode, and copied to CPSR.C on executing an exception return operation in Hyp mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to Hyp mode, and copied to CPSR.V on executing an exception return operation in Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:22]**

Reserved, RES0.

**SS, bit [21]**

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

`0` Taken from A32 state.

`1` Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

`1` Exception taken from AArch32.

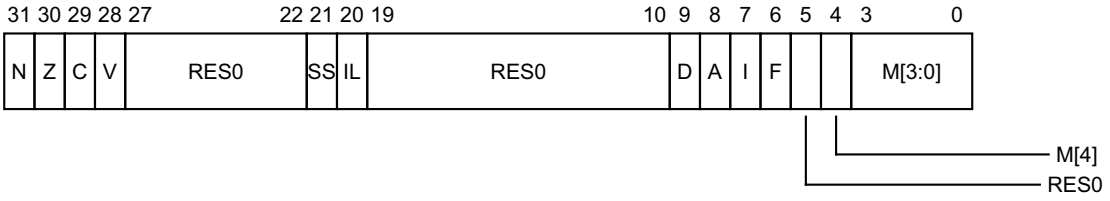
**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| `0b0000` | User |
| `0b0001` | FIQ |
| `0b0010` | IRQ |
| `0b0011` | Supervisor |
| `0b0111` | Abort |
| `0b1010` | Hyp |
| `0b1011` | Undefined |
| `0b1111` | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

***When exception taken from AArch64:***





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**N, bit [31]**

Set to the value of the N condition flag on taking an exception to EL2, and copied to the N condition flag on executing an exception return operation in EL2.

**Z, bit [30]**

Set to the value of the Z condition flag on taking an exception to EL2, and copied to the Z condition flag on executing an exception return operation in EL2.

**C, bit [29]**

Set to the value of the C condition flag on taking an exception to EL2, and copied to the C condition flag on executing an exception return operation in EL2.

**V, bit [28]**

Set to the value of the V condition flag on taking an exception to EL2, and copied to the V condition flag on executing an exception return operation in EL2.

**Bits [27:22]**

Reserved, RES0.

**SS, bit [21]**

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**Bits [19:10]**

Reserved, RES0.

**D, bit [9]**

Process state D mask. The possible values of this bit are:

`0` Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current Exception level are not masked.

`1` Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

**A, bit [8]**

SError (System Error) mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**Bit [5]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

`0` Exception taken from AArch64.

**M[3:0], bits [3:0]**

AArch64 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| `0b0000` | EL0t |
| `0b0100` | EL1t |
| `0b0101` | EL1h |
| `0b1000` | EL2t |
| `0b1001` | EL2h |

Other values are reserved, and returning to an Exception level that is using AArch64 with a reserved value in this field is treated as an illegal exception return.

The bits in this field are interpreted as follows:

- M[3:2] holds the Exception Level.
- M[1] is unused and is RES0 for all non-reserved values.
- M[0] is used to select the SP:
  - — 0 means the SP is always SP0.
  - — 1 means the exception SP is determined by the EL.

### Accessing the SPSR_EL2:

To access the SPSR_EL2:

```
MRS <Xt>, SPSR_EL2 ; Read SPSR_EL2 into Xt
MSR SPSR_EL2, <Xt> ; Write Xt to SPSR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0100 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.19 SPSR_EL3, Saved Program Status Register (EL3)

The SPSR_EL3 characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

An exception return from EL3 using AArch64 makes SPSR_EL3 become UNKNOWN.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register SPSR_EL3 can be mapped to AArch32 System register SPSR_mon, but this is not architecturally mandated.

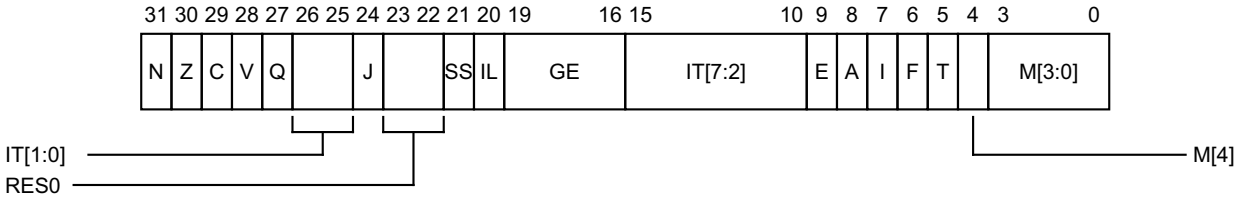
**Attributes**

SPSR_EL3 is a 32-bit register.

#### Field descriptions

The SPSR_EL3 bit assignments are:

***When exception taken from AArch32:***




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to Monitor mode, and copied to CPSR.N on executing an exception return operation in Monitor mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to Monitor mode, and copied to CPSR.Z on executing an exception return operation in Monitor mode.

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to Monitor mode, and copied to CPSR.C on executing an exception return operation in Monitor mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to Monitor mode, and copied to CPSR.V on executing an exception return operation in Monitor mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:22]**

Reserved, RES0.

**SS, bit [21]**

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

`0` Taken from A32 state.

`1` Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

`1` Exception taken from AArch32.

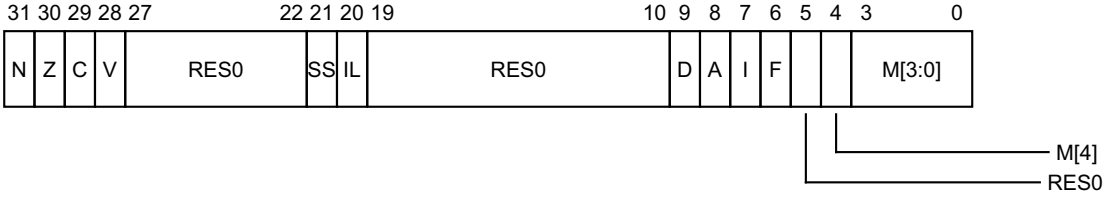
**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| `0b0000` | User |
| `0b0001` | FIQ |
| `0b0010` | IRQ |
| `0b0011` | Supervisor |
| `0b0110` | Monitor |
| `0b0111` | Abort |
| `0b1010` | Hyp |
| `0b1011` | Undefined |
| `0b1111` | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

***When exception taken from AArch64:***





*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**N, bit [31]**

Set to the value of the N condition flag on taking an exception to EL3, and copied to the N condition flag on executing an exception return operation in EL3.

**Z, bit [30]**

Set to the value of the Z condition flag on taking an exception to EL3, and copied to the Z condition flag on executing an exception return operation in EL3.

**C, bit [29]**

Set to the value of the C condition flag on taking an exception to EL3, and copied to the C condition flag on executing an exception return operation in EL3.

**V, bit [28]**

Set to the value of the V condition flag on taking an exception to EL3, and copied to the V condition flag on executing an exception return operation in EL3.

**Bits [27:22]**

Reserved, RES0.

**SS, bit [21]**

Software step. Shows the value of PSTATE.SS immediately before the exception was taken.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**Bits [19:10]**

Reserved, RES0.

**D, bit [9]**

Process state D mask. The possible values of this bit are:

`0` Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current Exception level are not masked.

`1` Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

**A, bit [8]**

SError (System Error) mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**Bit [5]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

`0` Exception taken from AArch64.

**M[3:0], bits [3:0]**

AArch64 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| `0b0000` | EL0t |
| `0b0100` | EL1t |
| `0b0101` | EL1h |
| `0b1000` | EL2t |
| `0b1001` | EL2h |
| `0b1100` | EL3t |
| `0b1101` | EL3h |

Other values are reserved, and returning to an Exception level that is using AArch64 with a reserved value in this field is treated as an illegal exception return.

The bits in this field are interpreted as follows:

- M[3:2] holds the Exception Level.
- M[1] is unused and is RES0 for all non-reserved values.
- M[0] is used to select the SP:
  - — 0 means the SP is always SP0.
  - — 1 means the exception SP is determined by the EL.

### Accessing the SPSR_EL3:

To access the SPSR_EL3:

```
MRS <Xt>, SPSR_EL3 ; Read SPSR_EL3 into Xt
MSR SPSR_EL3, <Xt> ; Write Xt to SPSR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0100 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.20 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to FIQ mode.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register SPSR_fiq is architecturally mapped to AArch32 System register SPSR_fiq.

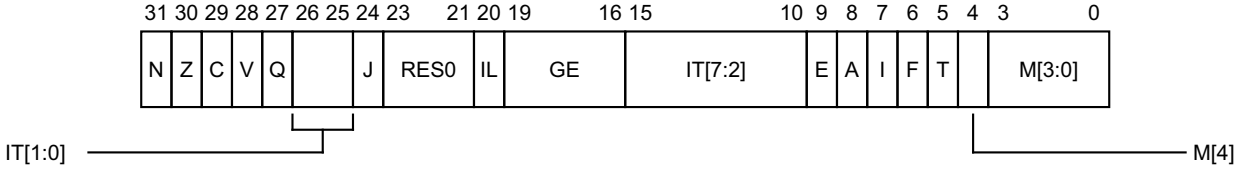
If EL1 does not support execution in AArch32, this register is RES0.

**Attributes**

SPSR_fiq is a 32-bit register.

#### Field descriptions

The SPSR_fiq bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to FIQ mode, and copied to CPSR.N on executing an exception return operation in FIQ mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to FIQ mode, and copied to CPSR.Z on executing an exception return operation in FIQ mode.

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to FIQ mode, and copied to CPSR.C on executing an exception return operation in FIQ mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to FIQ mode, and copied to CPSR.V on executing an exception return operation in FIQ mode.

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor |
| 0b0111 | Abort |
| 0b1010 | Hyp |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

## Accessing the SPSR_fiq:

To access the SPSR_fiq:

```
MRS <Xt>, SPSR_fiq ; Read SPSR_fiq into Xt
MSR SPSR_fiq, <Xt> ; Write Xt to SPSR_fiq
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0100 | 0011 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.2.21 SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to IRQ mode.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register SPSR_irq is architecturally mapped to AArch32 System register SPSR_irq.

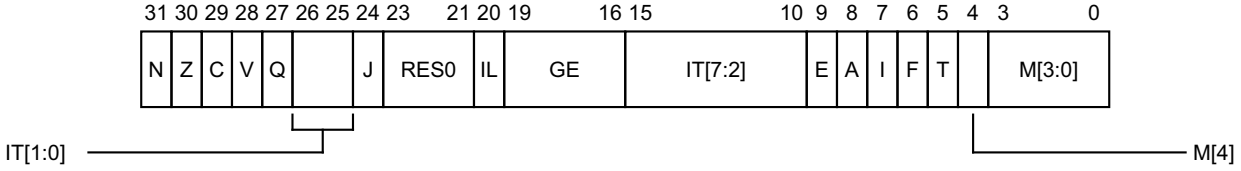
If EL1 does not support execution in AArch32, this register is RES0.

**Attributes**

SPSR_irq is a 32-bit register.

#### Field descriptions

The SPSR_irq bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to IRQ mode, and copied to CPSR.N on executing an exception return operation in IRQ mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to IRQ mode, and copied to CPSR.Z on executing an exception return operation in IRQ mode.

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to IRQ mode, and copied to CPSR.C on executing an exception return operation in IRQ mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to IRQ mode, and copied to CPSR.V on executing an exception return operation in IRQ mode.

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

`0` Taken from A32 state.

`1` Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

`1` Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor |
| 0b0111 | Abort |
| 0b1010 | Hyp |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

## Accessing the SPSR_irq:

To access the SPSR_irq:

```
MRS <Xt>, SPSR_irq ; Read SPSR_irq into Xt
MSR SPSR_irq, <Xt> ; Write Xt to SPSR_irq
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0100 | 0011 | 000 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### C5.2.22 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Undefined mode.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register SPSR_und is architecturally mapped to AArch32 System register SPSR_und.

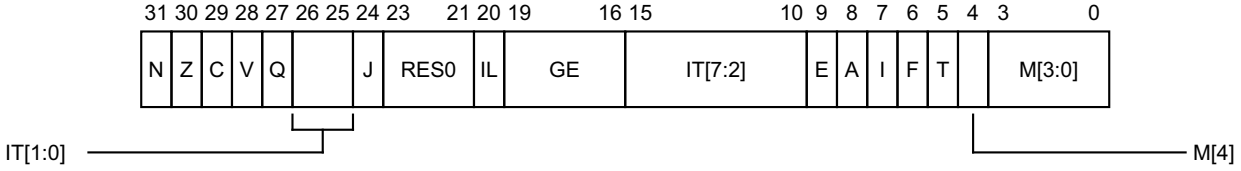
If EL1 does not support execution in AArch32, this register is RES0.

**Attributes**

SPSR_und is a 32-bit register.

## Field descriptions

The SPSR_und bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to Undefined mode, and copied to CPSR.N on executing an exception return operation in Undefined mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to Undefined mode, and copied to CPSR.Z on executing an exception return operation in Undefined mode.

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to Undefined mode, and copied to CPSR.C on executing an exception return operation in Undefined mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to Undefined mode, and copied to CPSR.V on executing an exception return operation in Undefined mode.

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor |
| 0b0111 | Abort |
| 0b1010 | Hyp |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

## Accessing the SPSR_und:

To access the SPSR_und:

```
MRS <Xt>, SPSR_und ; Read SPSR_und into Xt
MSR SPSR_und, <Xt> ; Write Xt to SPSR_und
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0100 | 0011 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C5.3 A64 system instructions for cache maintenance

The following sections define the cache maintenance system instructions in A64:


- *DC CISW, Data or unified Cache line Clean and Invalidate by Set/Way* on page C5-343.
- *DC CIVAC, Data or unified Cache line Clean and Invalidate by VA to PoC* on page C5-345.
- *DC CSW, Data or unified Cache line Clean by Set/Way* on page C5-346.
- *DC CVAC, Data or unified Cache line Clean by VA to PoC* on page C5-348.
- *DC CVAU, Data or unified Cache line Clean by VA to PoU* on page C5-349.
- *DC ISW, Data or unified Cache line Invalidate by Set/Way* on page C5-350.
- *DC IVAC, Data or unified Cache line Invalidate by VA to PoC* on page C5-352.
- *DC ZVA, Data Cache Zero by VA* on page C5-353.
- *IC IALLU, Instruction Cache Invalidate All to PoU* on page C5-355.
- *IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable* on page C5-356.
- *IC IVAU, Instruction Cache line Invalidate by VA to PoU* on page C5-357.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.1 DC CISW, Data or unified Cache line Clean and Invalidate by Set/Way

The DC CISW characteristics are:

**Purpose**

Clean and Invalidate data cache by set/way.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - — No cache lines.
  - — A single arbitrary cache line.
  - — Multiple arbitrary cache lines.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TSW==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

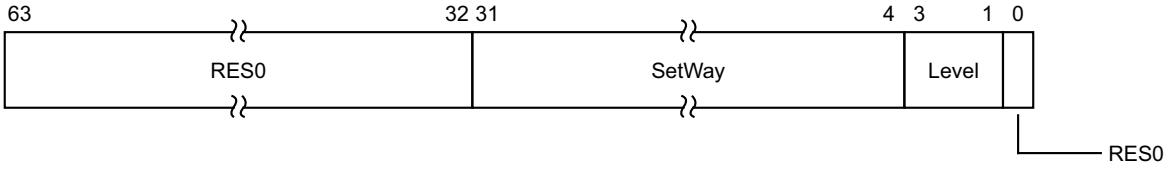
AArch64 System operation DC CISW performs the same function as AArch32 System operation DCCISW.

**Attributes**

DC CISW is a 64-bit system operation.

#### Field descriptions

The DC CISW input value bit assignments are:




**Bits [63:32]**

Reserved, RES0.

**SetWay, bits [31:4]**

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = $Log_2$(ASSOCIATIVITY), L = $Log_2$(LINELEN), B = (L + S), S = $Log_2$(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**

Reserved, RES0.

### Performing the DC CISW operation:

To perform the DC CISW operation:

```
DC CISW, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 0111 | 1110 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.2 DC CIVAC, Data or unified Cache line Clean and Invalidate by VA to PoC

The DC CIVAC characteristics are:

**Purpose**

Clean and Invalidate data cache by address to Point of Coherency.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-WO | WO | WO | WO | WO | WO |

If EL0 access is enabled, this instruction is available at EL0 when the VA has read access permission, otherwise it causes a Permission Fault.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TPC==1, Non-secure EL0 execution of this instruction is trapped to EL2.

If SCTLR_EL1.UCI==0, EL0 execution of this instruction is trapped to EL1.

**Configurations**

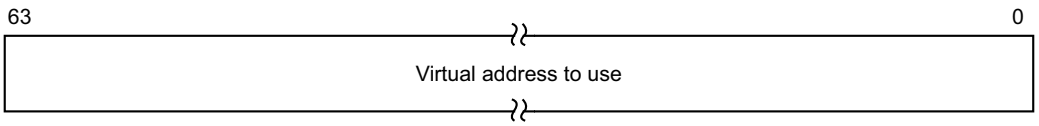
AArch64 System operation DC CIVAC performs the same function as AArch32 System operation DCCIMVAC.

**Attributes**

DC CIVAC is a 64-bit system operation.

#### Field descriptions

The DC CIVAC input value bit assignments are:




**Bits [63:0]**

Virtual address to use.

#### Performing the DC CIVAC operation:

To perform the DC CIVAC operation:

```
DC CIVAC, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 011 | 0111 | 1110 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.3 DC CSW, Data or unified Cache line Clean by Set/Way

The DC CSW characteristics are:

**Purpose**

Clean data cache by set/way.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TSW==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

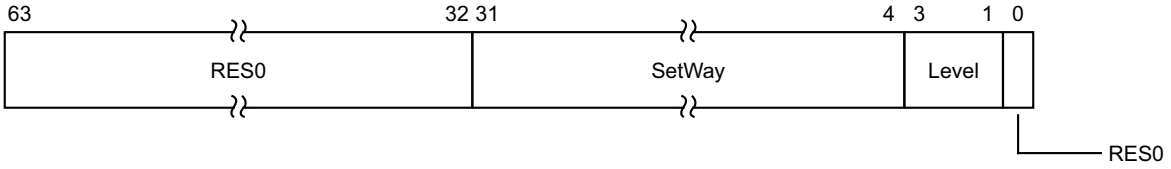
AArch64 System operation DC CSW performs the same function as AArch32 System operation DCCSW.

**Attributes**

DC CSW is a 64-bit system operation.

#### Field descriptions

The DC CSW input value bit assignments are:




**Bits [63:32]**

Reserved, RES0.

**SetWay, bits [31:4]**

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = $Log_2$(ASSOCIATIVITY), L = $Log_2$(LINELEN), B = (L + S), S = $Log_2$(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**

Reserved, RES0.

### Performing the DC CSW operation:

To perform the DC CSW operation:

```
DC CSW, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 0111 | 1010 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.4 DC CVAC, Data or unified Cache line Clean by VA to PoC

The DC CVAC characteristics are:

**Purpose**

Clean data cache by address to Point of Coherency.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-WO | WO | WO | WO | WO | WO |

If EL0 access is enabled, this instruction is available at EL0 when the VA has read access permission, otherwise it causes a Permission Fault.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TPC==1, Non-secure EL0 execution of this instruction is trapped to EL2.

If SCTLR_EL1.UCI==0, EL0 execution of this instruction is trapped to EL1.

**Configurations**

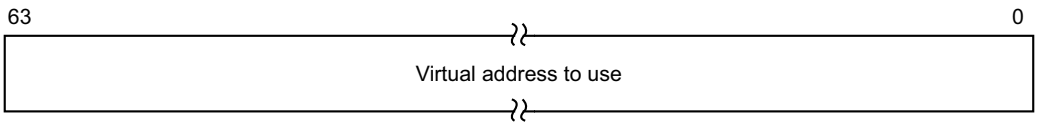
AArch64 System operation DC CVAC performs the same function as AArch32 System operation DCCMVAC.

**Attributes**

DC CVAC is a 64-bit system operation.

#### Field descriptions

The DC CVAC input value bit assignments are:




**Bits [63:0]**

Virtual address to use.

#### Performing the DC CVAC operation:

To perform the DC CVAC operation:

```
DC CVAC, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 011 | 0111 | 1010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.5 DC CVAU, Data or unified Cache line Clean by VA to PoU

The DC CVAU characteristics are:

**Purpose**

Clean data cache by address to Point of Unification.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-WO | WO | WO | WO | WO | WO |

If EL0 access is enabled, this instruction is available at EL0 when the VA has read access permission, otherwise it causes a Permission Fault.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TPU==1, Non-secure EL0 and EL1 execution of this instruction is trapped to EL2.

If SCTLR_EL1.UCI==0, EL0 execution of this instruction is trapped to EL1.

**Configurations**

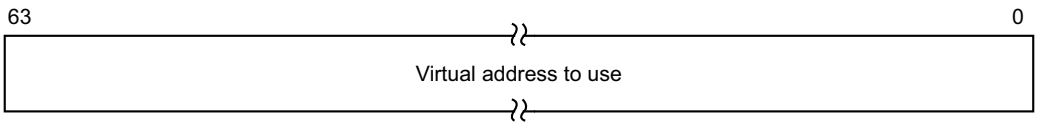
AArch64 System operation DC CVAU performs the same function as AArch32 System operation DCCMVAU.

**Attributes**

DC CVAU is a 64-bit system operation.

#### Field descriptions

The DC CVAU input value bit assignments are:




**Bits [63:0]**

Virtual address to use.

#### Performing the DC CVAU operation:

To perform the DC CVAU operation:

```
DC CVAU, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 011 | 0111 | 1011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.6 DC ISW, Data or unified Cache line Invalidate by Set/Way

The DC ISW characteristics are:

**Purpose**

Invalidate data cache by set/way.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

At EL1, this operation must operate as DC CISW if all of the following apply:

- EL2 is implemented.
- The value of HCR_EL2.SWIO is 1 or the value of HCR_EL2.VM is 1.
- The value of SCR_EL3.NS is 1 or EL3 is not implemented.

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TSW==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

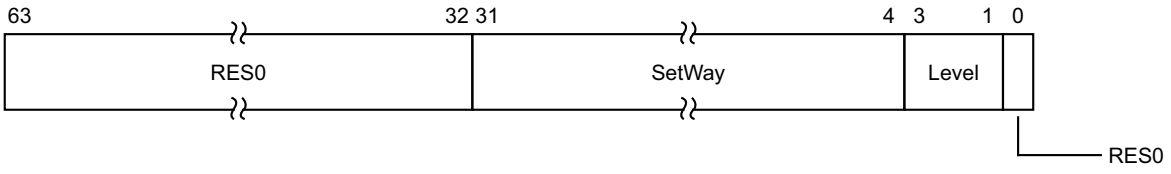
AArch64 System operation DC ISW performs the same function as AArch32 System operation DCISW.

**Attributes**

DC ISW is a 64-bit system operation.

#### Field descriptions

The DC ISW input value bit assignments are:




**Bits [63:32]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SetWay, bits [31:4]**

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = $Log_2$(ASSOCIATIVITY), L = $Log_2$(LINELEN), B = (L + S), S = $Log_2$(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**

Reserved, RES0.

### Performing the DC ISW operation:

To perform the DC ISW operation:

```
DC ISW, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 0111 | 0110 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.7 DC IVAC, Data or unified Cache line Invalidate by VA to PoC

The DC IVAC characteristics are:

**Purpose**

Invalidate data cache by address to Point of Coherency.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

This instruction requires write access permission to the VA, otherwise it causes a Permission Fault.

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

At EL1, this instruction must be performed as DC CIVAC if all of the following apply:

- EL2 is implemented.
- HCR_EL2.VM is set to 1.
- SCR_EL3.NS is set to 1 or EL3 is not implemented.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

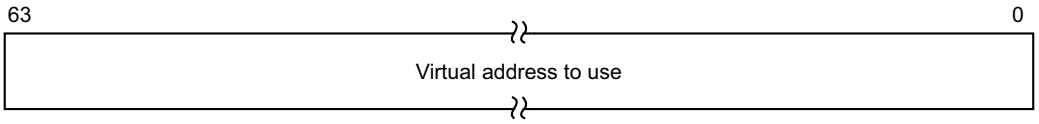
AArch64 System operation DC IVAC performs the same function as AArch32 System operation DCIMVAC.

**Attributes**

DC IVAC is a 64-bit system operation.

#### Field descriptions

The DC IVAC input value bit assignments are:




**Bits [63:0]**

Virtual address to use.

#### Performing the DC IVAC operation:

To perform the DC IVAC operation:

```
DC IVAC, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 0111 | 0110 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.8 DC ZVA, Data Cache Zero by VA

The DC ZVA characteristics are:

**Purpose**

Zero data cache by address. Zeroes a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-WO | WO | WO | WO | WO | WO |

When UNDEFINED, this instruction always takes the EL1 UNDEFINED exception.

When this instruction is executed, it can generate memory faults or watchpoints which are prioritised in the same way as other memory related faults or watchpoints. If a synchronous data abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is not set.

If the memory region being zeroed is any type of Device memory, this instruction can give an alignment fault which is prioritised in the same way as other alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte within the block being accessed, and so it:

- Will cause a Permission Fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TDZ==1, Non-secure EL0 and EL1 execution of this instruction is trapped to EL2.

If SCTLR_EL1.DZE==0, EL0 execution of this instruction is trapped to EL1.

**Configurations**

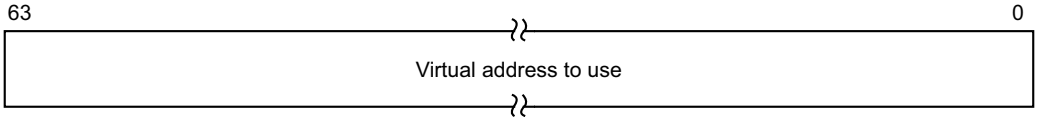
There are no configuration notes.

**Attributes**

DC ZVA is a 64-bit system operation.

#### Field descriptions

The DC ZVA input value bit assignments are:




**Bits [63:0]**

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the DC ZVA operation:

To perform the DC ZVA operation:

```
DC ZVA, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 011 | 0111 | 0100 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.9 IC IALLU, Instruction Cache Invalidate All to PoU

The IC IALLU characteristics are:

**Purpose**

Invalidate all instruction caches to Point of Unification.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TPU==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

AArch64 System operation IC IALLU performs the same function as AArch32 System operation ICIALLU.

**Attributes**

IC IALLU is a 64-bit system operation.

#### Field descriptions

The IC IALLU operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the IC IALLU operation:

To perform the IC IALLU operation:

```
IC IALLU
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 0111 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.10 IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The IC IALLUIS characteristics are:

**Purpose**

Invalidate all instruction caches in Inner Shareable domain to Point of Unification.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TPU==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

AArch64 System operation IC IALLUIS performs the same function as AArch32 System operation ICIALLUIS.

**Attributes**

IC IALLUIS is a 64-bit system operation.

#### Field descriptions

The IC IALLUIS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the IC IALLUIS operation:

To perform the IC IALLUIS operation:

```
IC IALLUIS
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 0111 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.3.11 IC IVAU, Instruction Cache line Invalidate by VA to PoU

The IC IVAU characteristics are:

**Purpose**

Invalidate instruction cache by address to Point of Unification.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-WO | WO | WO | WO | WO | WO |

If EL0 access is enabled, this instruction is available at EL0 when the VA has read access permission. It is IMPLEMENTATION DEFINED whether executing this instruction at EL0 causes a Permission Fault if the VA does not have read access permission at EL0.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TPU==1, Non-secure EL0 and EL1 execution of this instruction is trapped to EL2.

If SCTLR_EL1.UCI==0, EL0 execution of this instruction is trapped to EL1.

**Configurations**

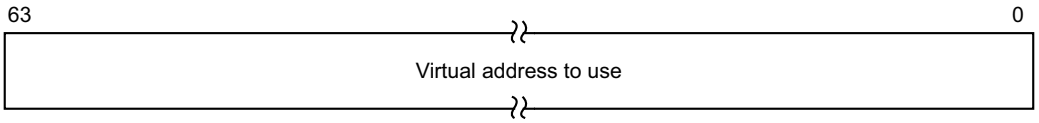
AArch64 System operation IC IVAU performs the same function as AArch32 System operation ICIMVAU.

**Attributes**

IC IVAU is a 64-bit system operation.

#### Field descriptions

The IC IVAU input value bit assignments are:




**Bits [63:0]**

Virtual address to use.

#### Performing the IC IVAU operation:

To perform the IC IVAU operation:

```
IC IVAU, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 011 | 0111 | 0101 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C5.4 A64 system instructions for address translation

The following sections define the address translation system instructions in A64:


- *AT S12E0R, Address Translate Stages 1 and 2 EL0 Read* on page C5-359.
- *AT S12E0W, Address Translate Stages 1 and 2 EL0 Write* on page C5-360.
- *AT S12E1R, Address Translate Stages 1 and 2 EL1 Read* on page C5-361.
- *AT S12E1W, Address Translate Stages 1 and 2 EL1 Write* on page C5-362.
- *AT S1E0R, Address Translate Stage 1 EL0 Read* on page C5-363.
- *AT S1E0W, Address Translate Stage 1 EL0 Write* on page C5-364.
- *AT S1E1R, Address Translate Stage 1 EL1 Read* on page C5-365.
- *AT S1E1W, Address Translate Stage 1 EL1 Write* on page C5-366.
- *AT S1E2R, Address Translate Stage 1 EL2 Read* on page C5-367.
- *AT S1E2W, Address Translate Stage 1 EL2 Write* on page C5-368.
- *AT S1E3R, Address Translate Stage 1 EL3 Read* on page C5-369.
- *AT S1E3W, Address Translate Stage 1 EL3 Write* on page C5-370.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.1 AT S12E0R, Address Translate Stages 1 and 2 EL0 Read

The AT S12E0R characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for EL0, with permissions as if reading from the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL2 does not exist, or stage 2 translation is disabled, or the instruction is executed at EL3 when the value of SCR_EL3.NS is 0, this operation executes as AT S1E0R.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

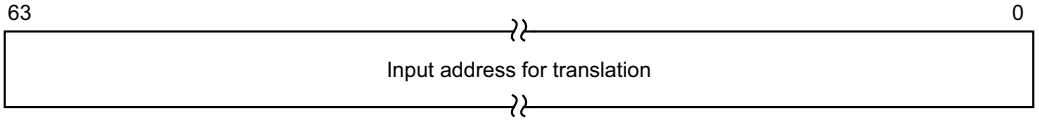
There are no configuration notes.

**Attributes**

AT S12E0R is a 64-bit system operation.

#### Field descriptions

The AT S12E0R input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S12E0R operation:

To perform the AT S12E0R operation:

```
AT S12E0R, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 0111 | 1000 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.2 AT S12E0W, Address Translate Stages 1 and 2 EL0 Write

The AT S12E0W characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for EL0, with permissions as if writing to the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL2 does not exist, or stage 2 translation is disabled, or the instruction is executed at EL3 when the value of SCR_EL3.NS is 0, this operation executes as AT S1E0W.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

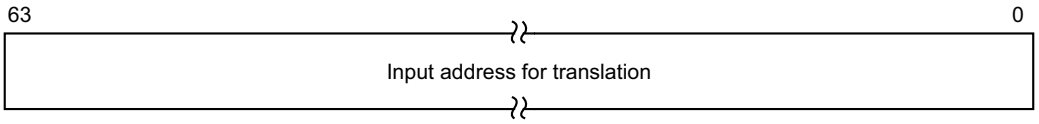
There are no configuration notes.

**Attributes**

AT S12E0W is a 64-bit system operation.

#### Field descriptions

The AT S12E0W input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S12E0W operation:

To perform the AT S12E0W operation:

```
AT S12E0W, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 0111 | 1000 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.3 AT S12E1R, Address Translate Stages 1 and 2 EL1 Read

The AT S12E1R characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for EL1, with permissions as if reading from the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL2 does not exist, or stage 2 translation is disabled, or the instruction is executed at EL3 when the value of SCR_EL3.NS is 0, this operation executes as AT S1E1R.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

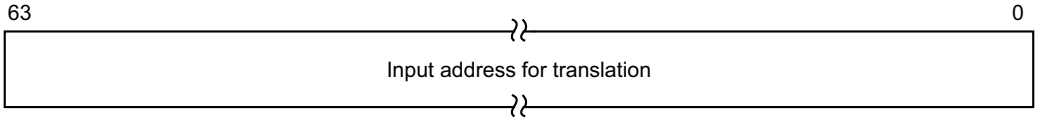
There are no configuration notes.

**Attributes**

AT S12E1R is a 64-bit system operation.

#### Field descriptions

The AT S12E1R input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S12E1R operation:

To perform the AT S12E1R operation:

```
AT S12E1R, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 0111 | 1000 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.4 AT S12E1W, Address Translate Stages 1 and 2 EL1 Write

The AT S12E1W characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for EL1, with permissions as if writing to the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL2 does not exist, or stage 2 translation is disabled, or the instruction is executed at EL3 when the value of SCR_EL3.NS is 0, this operation executes as AT S1E1W.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

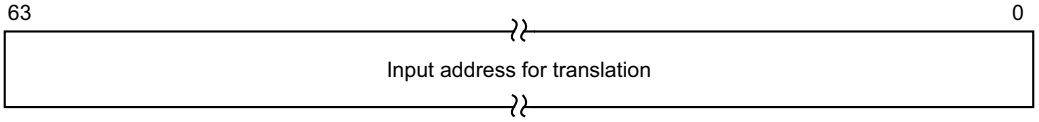
There are no configuration notes.

**Attributes**

AT S12E1W is a 64-bit system operation.

#### Field descriptions

The AT S12E1W input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S12E1W operation:

To perform the AT S12E1W operation:

```
AT S12E1W, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 0111 | 1000 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.5 AT S1E0R, Address Translate Stage 1 EL0 Read

The AT S1E0R characteristics are:

**Purpose**

Performs stage 1 address translation as defined for EL0, with permissions as if reading from the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

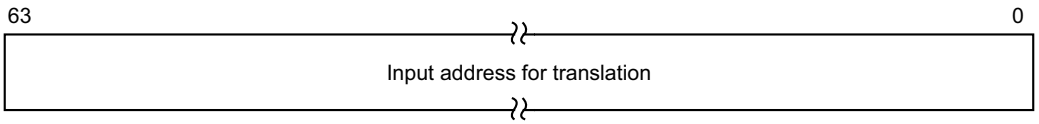
There are no configuration notes.

**Attributes**

AT S1E0R is a 64-bit system operation.

#### Field descriptions

The AT S1E0R input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S1E0R operation:

To perform the AT S1E0R operation:

```
AT S1E0R, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 0111 | 1000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.6 AT S1E0W, Address Translate Stage 1 EL0 Write

The AT S1E0W characteristics are:

**Purpose**

Performs stage 1 address translation as defined for EL0, with permissions as if writing to the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

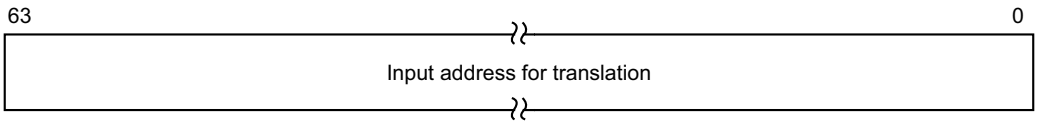
There are no configuration notes.

**Attributes**

AT S1E0W is a 64-bit system operation.

#### Field descriptions

The AT S1E0W input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S1E0W operation:

To perform the AT S1E0W operation:

```
AT S1E0W, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 0111 | 1000 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.7 AT S1E1R, Address Translate Stage 1 EL1 Read

The AT S1E1R characteristics are:

**Purpose**

Performs stage 1 address translation as defined for EL1, with permissions as if reading from the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

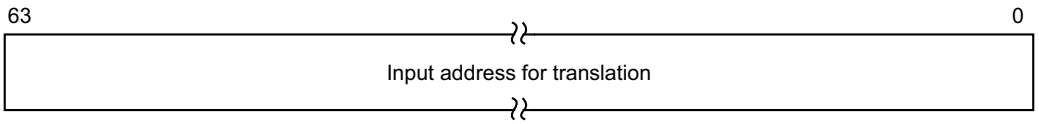
There are no configuration notes.

**Attributes**

AT S1E1R is a 64-bit system operation.

#### Field descriptions

The AT S1E1R input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S1E1R operation:

To perform the AT S1E1R operation:

```
AT S1E1R, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 0111 | 1000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.8 AT S1E1W, Address Translate Stage 1 EL1 Write

The AT S1E1W characteristics are:

**Purpose**

Performs stage 1 address translation as defined for EL1, with permissions as if writing to the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

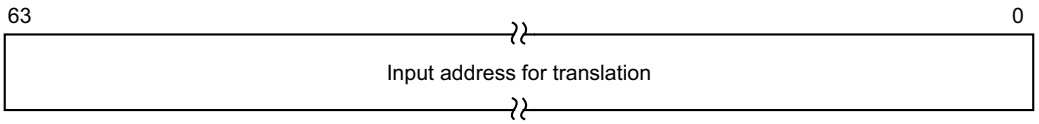
There are no configuration notes.

**Attributes**

AT S1E1W is a 64-bit system operation.

#### Field descriptions

The AT S1E1W input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S1E1W operation:

To perform the AT S1E1W operation:

```
AT S1E1W, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 0111 | 1000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.9 AT S1E2R, Address Translate Stage 1 EL2 Read

The AT S1E2R characteristics are:

**Purpose**

Performs stage 1 address translation as defined for EL2, with permissions as if reading from the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | - |

Performing this operation from EL3 is UNDEFINED if EL2 does not exist.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

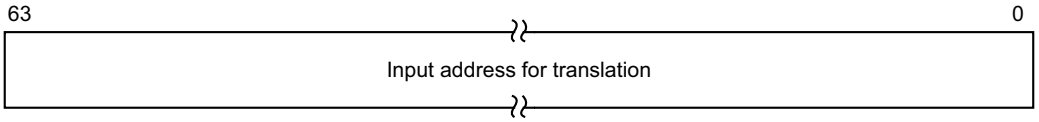
There are no configuration notes.

**Attributes**

AT S1E2R is a 64-bit system operation.

#### Field descriptions

The AT S1E2R input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S1E2R operation:

To perform the AT S1E2R operation:

```
AT S1E2R, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 0111 | 1000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.10 AT S1E2W, Address Translate Stage 1 EL2 Write

The AT S1E2W characteristics are:

**Purpose**

Performs stage 1 address translation as defined for EL2, with permissions as if writing to the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | - |

Performing this operation from EL3 is UNDEFINED if EL2 does not exist.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

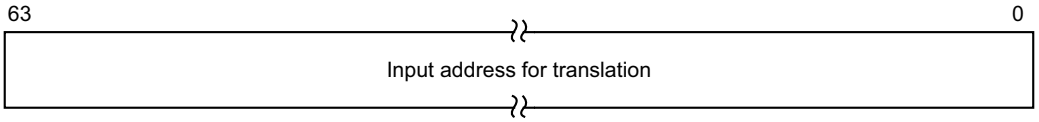
There are no configuration notes.

**Attributes**

AT S1E2W is a 64-bit system operation.

#### Field descriptions

The AT S1E2W input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S1E2W operation:

To perform the AT S1E2W operation:

```
AT S1E2W, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 0111 | 1000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.11 AT S1E3R, Address Translate Stage 1 EL3 Read

The AT S1E3R characteristics are:

**Purpose**

Performs stage 1 address translation as defined for EL3, with permissions as if reading from the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | WO | WO |

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

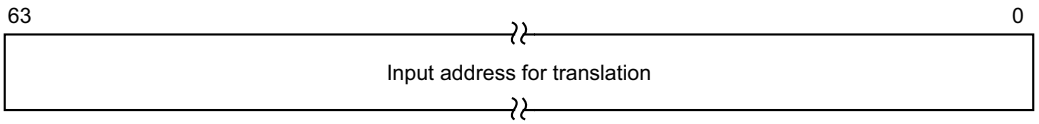
There are no configuration notes.

**Attributes**

AT S1E3R is a 64-bit system operation.

#### Field descriptions

The AT S1E3R input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S1E3R operation:

To perform the AT S1E3R operation:

```
AT S1E3R, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 110 | 0111 | 1000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.4.12 AT S1E3W, Address Translate Stage 1 EL3 Write

The AT S1E3W characteristics are:

**Purpose**

Performs stage 1 address translation as defined for EL3, with permissions as if writing to the given virtual address.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | WO | WO |

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

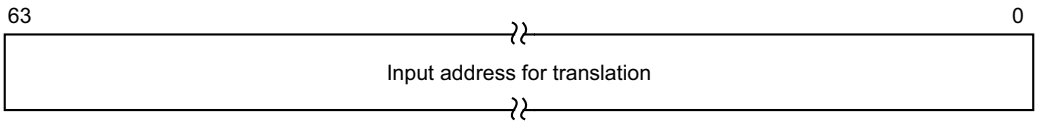
There are no configuration notes.

**Attributes**

AT S1E3W is a 64-bit system operation.

#### Field descriptions

The AT S1E3W input value bit assignments are:




**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

#### Performing the AT S1E3W operation:

To perform the AT S1E3W operation:

```
AT S1E3W, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 110 | 0111 | 1000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C5.5 A64 system instructions for TLB maintenance

The following sections define the TLB maintenance system instructions in A64:

- *TLBI ALLE1, TLB Invalidate All, EL1* on page C5-372.
- *TLBI ALLE1IS, TLB Invalidate All, EL1, Inner Shareable* on page C5-373.
- *TLBI ALLE2, TLB Invalidate All, EL2* on page C5-374.
- *TLBI ALLE2IS, TLB Invalidate All, EL2, Inner Shareable* on page C5-375.
- *TLBI ALLE3, TLB Invalidate All, EL3* on page C5-376.
- *TLBI ALLE3IS, TLB Invalidate All, EL3, Inner Shareable* on page C5-377.
- *TLBI ASIDE1, TLB Invalidate by ASID, EL1* on page C5-378.
- *TLBI ASIDE1IS, TLB Invalidate by ASID, EL1, Inner Shareable* on page C5-380.
- *TLBI IPAS2E1, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1* on page C5-382.
- *TLBI IPAS2E1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable* on page C5-383.
- *TLBI IPAS2LE1, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1* on page C5-385.
- *TLBI IPAS2LE1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable* on page C5-386.
- *TLBI VAAE1, TLB Invalidate by VA, All ASID, EL1* on page C5-388.
- *TLBI VAAE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable* on page C5-390.
- *TLBI VAALE1, TLB Invalidate by VA, All ASID, Last level, EL1* on page C5-392.
- *TLBI VAALE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable* on page C5-394.
- *TLBI VAE1, TLB Invalidate by VA, EL1* on page C5-396.
- *TLBI VAE1IS, TLB Invalidate by VA, EL1, Inner Shareable* on page C5-398.
- *TLBI VAE2, TLB Invalidate by VA, EL2* on page C5-400.
- *TLBI VAE2IS, TLB Invalidate by VA, EL2, Inner Shareable* on page C5-402.
- *TLBI VAE3, TLB Invalidate by VA, EL3* on page C5-404.
- *TLBI VAE3IS, TLB Invalidate by VA, EL3, Inner Shareable* on page C5-406.
- *TLBI VALE1, TLB Invalidate by VA, Last level, EL1* on page C5-408.
- *TLBI VALE1IS, TLB Invalidate by VA, Last level, EL1, Inner Shareable* on page C5-410.
- *TLBI VALE2, TLB Invalidate by VA, Last level, EL2* on page C5-412.
- *TLBI VALE2IS, TLB Invalidate by VA, Last level, EL2, Inner Shareable* on page C5-414.
- *TLBI VALE3, TLB Invalidate by VA, Last level, EL3* on page C5-416.
- *TLBI VALE3IS, TLB Invalidate by VA, Last level, EL3, Inner Shareable* on page C5-418.
- *TLBI VMALLE1, TLB Invalidate by VMID, All at stage 1, EL1* on page C5-420.
- *TLBI VMALLE1IS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable* on page C5-421.
- *TLBI VMALLS12E1, TLB Invalidate by VMID, All at Stage 1 and 2, EL1* on page C5-422.
- *TLBI VMALLS12E1IS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable* on page C5-423.

For more information about these instructions see *TLB maintenance instructions* on page D4-1839. In particular, for the full description of the scope of each instruction see *Scope of the A64 TLB maintenance instructions* on page D4-1842.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.1 TLBI ALLE1, TLB Invalidate All, EL1

The TLBI ALLE1 characteristics are:

**Purpose**

Invalidate all EL1&0 regime stage 1 and 2 TLB entries.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see ALL.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE1 is a 64-bit system operation.

#### Field descriptions

The TLBI ALLE1 operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBI ALLE1 operation:

To perform the TLBI ALLE1 operation:

```
TLBI ALLE1
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0111 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.2 TLBI ALLE1IS, TLB Invalidate All, EL1, Inner Shareable

The TLBI ALLE1IS characteristics are:

**Purpose**

Invalidate all EL1&0 regime stage 1 and 2 TLB entries on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see ALL.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE1IS is a 64-bit system operation.

#### Field descriptions

The TLBI ALLE1IS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBI ALLE1IS operation:

To perform the TLBI ALLE1IS operation:

```
TLBI ALLE1IS
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0011 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.3 TLBI ALLE2, TLB Invalidate All, EL2

The TLBI ALLE2 characteristics are:

**Purpose**

Invalidate all EL2 regime stage 1 TLB entries.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | - |

Performing this operation from EL3 is UNDEFINED if EL2 does not exist.

For details of the scope of this instruction see ALL.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE2 is a 64-bit system operation.

#### Field descriptions

The TLBI ALLE2 operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBI ALLE2 operation:

To perform the TLBI ALLE2 operation:

```
TLBI ALLE2
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0111 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.4 TLBI ALLE2IS, TLB Invalidate All, EL2, Inner Shareable

The TLBI ALLE2IS characteristics are:

**Purpose**

Invalidate all EL2 regime stage 1 TLB entries on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | - |

Performing this operation from EL3 is UNDEFINED if EL2 does not exist.

For details of the scope of this instruction see ALL.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE2IS is a 64-bit system operation.

#### Field descriptions

The TLBI ALLE2IS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBI ALLE2IS operation:

To perform the TLBI ALLE2IS operation:

```
TLBI ALLE2IS
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.5 TLBI ALLE3, TLB Invalidate All, EL3

The TLBI ALLE3 characteristics are:

**Purpose**

Invalidate all EL3 regime stage 1 TLB entries.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | WO | WO |

For details of the scope of this instruction see ALL.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE3 is a 64-bit system operation.

#### Field descriptions

The TLBI ALLE3 operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBI ALLE3 operation:

To perform the TLBI ALLE3 operation:

```
TLBI ALLE3
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 110 | 1000 | 0111 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.6 TLBI ALLE3IS, TLB Invalidate All, EL3, Inner Shareable

The TLBI ALLE3IS characteristics are:

**Purpose**

Invalidate all EL3 regime stage 1 TLB entries on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | WO | WO |

For details of the scope of this instruction see ALL.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE3IS is a 64-bit system operation.

#### Field descriptions

The TLBI ALLE3IS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBI ALLE3IS operation:

To perform the TLBI ALLE3IS operation:

```
TLBI ALLE3IS
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 110 | 1000 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.7 TLBI ASIDE1, TLB Invalidate by ASID, EL1

The TLBI ASIDE1 characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 1 TLB entries for the given ASID and the current VMID.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see ASID.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

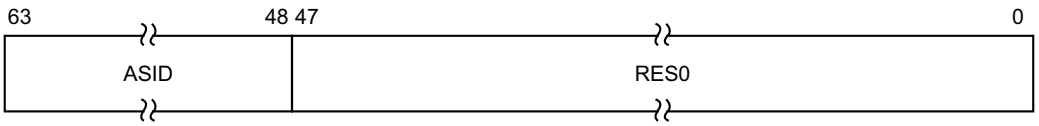
There are no configuration notes.

**Attributes**

TLBI ASIDE1 is a 64-bit system operation.

#### Field descriptions

The TLBI ASIDE1 input value bit assignments are:




**ASID, bits [63:48]**

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this operation.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

**Bits [47:0]**

Reserved, RES0.

#### Performing the TLBI ASIDE1 operation:

To perform the TLBI ASIDE1 operation:

```
TLBI ASIDE1, <Xt>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0111 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.8 TLBI ASIDE1IS, TLB Invalidate by ASID, EL1, Inner Shareable

The TLBI ASIDE1IS characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 1 TLB entries for the given ASID and the current VMID on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see ASID.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

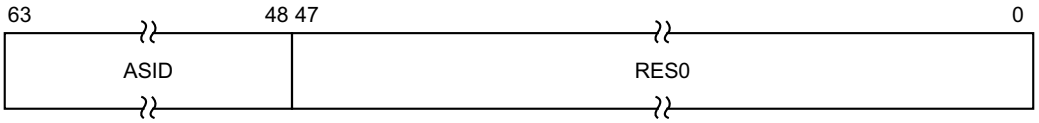
There are no configuration notes.

**Attributes**

TLBI ASIDE1IS is a 64-bit system operation.

#### Field descriptions

The TLBI ASIDE1IS input value bit assignments are:




**ASID, bits [63:48]**

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this operation.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

**Bits [47:0]**

Reserved, RES0.

#### Performing the TLBI ASIDE1IS operation:

To perform the TLBI ASIDE1IS operation:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`TLBI ASIDE1IS, <Xt>`

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0011 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.9 TLBI IPAS2E1, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI IPAS2E1 characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 2 TLB entries for the given IPA and the current VMID.

For details of the scope of this instruction see IPAS2.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If SCR_EL3.NS==0, or EL2 is not implemented, this instruction is a NOP.

This instruction must apply to structures that contain only stage 2 translation information, but does not need to apply to structures that contain combined stage 1 and stage 2 translation information.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

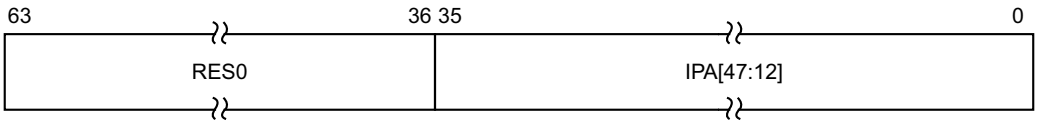
There are no configuration notes.

**Attributes**

TLBI IPAS2E1 is a 64-bit system operation.

#### Field descriptions

The TLBI IPAS2E1 input value bit assignments are:




**Bits [63:36]**

Reserved, RES0.

**IPA[47:12], bits [35:0]**

Bits[47:12] of the intermediate physical address to match.

#### Performing the TLBI IPAS2E1 operation:

To perform the TLBI IPAS2E1 operation:

```
TLBI IPAS2E1, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0100 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.10 TLBI IPAS2E1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBI IPAS2E1IS characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 2 TLB entries for the given IPA and the current VMID on all PEs in the same Inner Shareable domain.

For details of the scope of this instruction see IPAS2.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If SCR_EL3.NS==0, or EL2 is not implemented, this instruction is a NOP.

This instruction must apply to structures that contain only stage 2 translation information, but does not need to apply to structures that contain combined stage 1 and stage 2 translation information.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

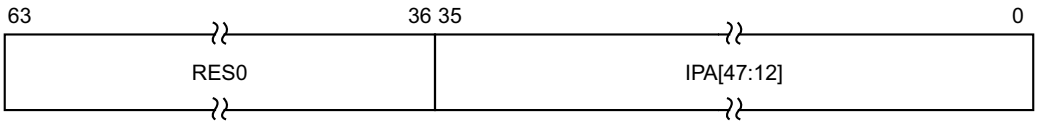
There are no configuration notes.

**Attributes**

TLBI IPAS2E1IS is a 64-bit system operation.

#### Field descriptions

The TLBI IPAS2E1IS input value bit assignments are:




**Bits [63:36]**

Reserved, RES0.

**IPA[47:12], bits [35:0]**

Bits[47:12] of the intermediate physical address to match.

#### Performing the TLBI IPAS2E1IS operation:

To perform the TLBI IPAS2E1IS operation:

```
TLBI IPAS2E1IS, <Xt>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.11 TLBI IPAS2LE1, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI IPAS2LE1 characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 2 TLB entries for the last level of translation, the given IPA, and the current VMID.

For details of the scope of this instruction see IPAS2L.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If SCR_EL3.NS==0, or EL2 is not implemented, this instruction is a NOP.

This instruction must apply to structures that contain only stage 2 translation information, but does not need to apply to structures that contain combined stage 1 and stage 2 translation information.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

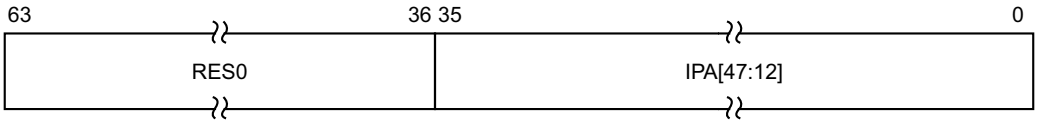
There are no configuration notes.

**Attributes**

TLBI IPAS2LE1 is a 64-bit system operation.

#### Field descriptions

The TLBI IPAS2LE1 input value bit assignments are:




**Bits [63:36]**

Reserved, RES0.

**IPA[47:12], bits [35:0]**

Bits[47:12] of the intermediate physical address to match.

#### Performing the TLBI IPAS2LE1 operation:

To perform the TLBI IPAS2LE1 operation:

```
TLBI IPAS2LE1, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0100 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.12 TLBI IPAS2LE1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI IPAS2LE1IS characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 2 TLB entries for the last level of translation, the given IPA, and the current VMID, on all PEs in the same Inner Shareable domain.

For details of the scope of this instruction see IPAS2L.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If SCR_EL3.NS==0, or EL2 is not implemented, this instruction is a NOP.

This instruction must apply to structures that contain only stage 2 translation information, but does not need to apply to structures that contain combined stage 1 and stage 2 translation information.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

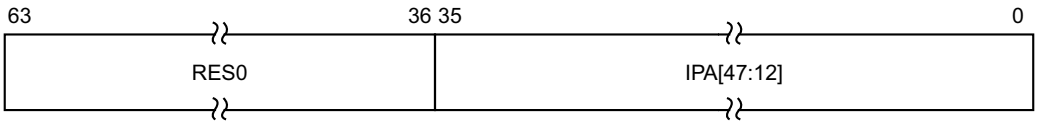
There are no configuration notes.

**Attributes**

TLBI IPAS2LE1IS is a 64-bit system operation.

#### Field descriptions

The TLBI IPAS2LE1IS input value bit assignments are:




**Bits [63:36]**

Reserved, RES0.

**IPA[47:12], bits [35:0]**

Bits[47:12] of the intermediate physical address to match.

#### Performing the TLBI IPAS2LE1IS operation:

To perform the TLBI IPAS2LE1IS operation:

```
TLBI IPAS2LE1IS, <Xt>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0000 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.13 TLBI VAAE1, TLB Invalidate by VA, All ASID, EL1

The TLBI VAAE1 characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 1 TLB entries for the given VA and the current VMID.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see VAA.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

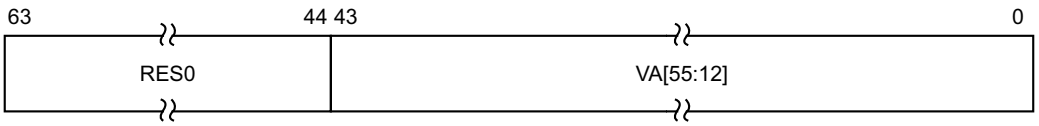
There are no configuration notes.

**Attributes**

TLBI VAAE1 is a 64-bit system operation.

#### Field descriptions

The TLBI VAAE1 input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this operation, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

### Performing the TLBI VAAE1 operation:

To perform the TLBI VAAE1 operation:

```
TLBI VAAE1, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0111 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.14 TLBI VAAE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI VAAE1IS characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 1 TLB entries for the given VA and the current VMID on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see VAA.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

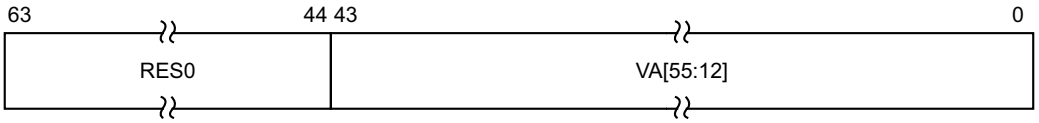
There are no configuration notes.

**Attributes**

TLBI VAAE1IS is a 64-bit system operation.

#### Field descriptions

The TLBI VAAE1IS input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this operation, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

### Performing the TLBI VAAE1IS operation:

To perform the TLBI VAAE1IS operation:

```
TLBI VAAE1IS, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0011 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.15 TLBI VAALE1, TLB Invalidate by VA, All ASID, Last level, EL1

The TLBI VAALE1 characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 1 TLB entries for the last level of translation table walk, the given VA, and the current VMID.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see VAAL.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

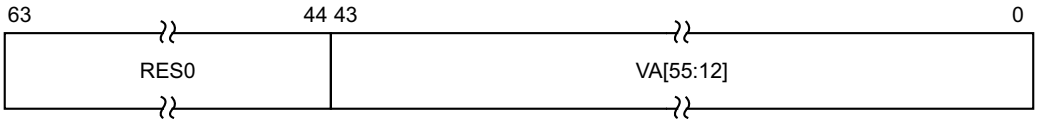
There are no configuration notes.

**Attributes**

TLBI VAALE1 is a 64-bit system operation.

#### Field descriptions

The TLBI VAALE1 input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this operation, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

### Performing the TLBI VAALE1 operation:

To perform the TLBI VAALE1 operation:

```
TLBI VAALE1, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0111 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.16 TLBI VAALE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI VAALE1IS characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 1 TLB entries for the last level of translation table walk, the given VA, and the current VMID, on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see VAAL.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

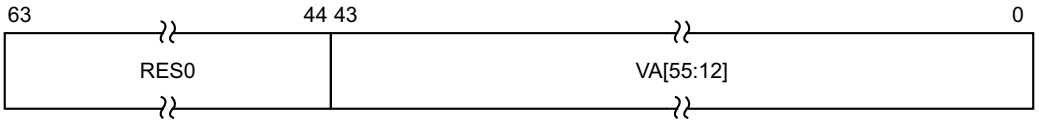
There are no configuration notes.

**Attributes**

TLBI VAALE1IS is a 64-bit system operation.

#### Field descriptions

The TLBI VAALE1IS input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this operation, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

### Performing the TLBI VAALE1IS operation:

To perform the TLBI VAALE1IS operation:

```
TLBI VAALE1IS, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0011 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.17 TLBI VAE1, TLB Invalidate by VA, EL1

The TLBI VAE1 characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 1 TLB entries for the given VA and ASID and the current VMID.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see VA.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

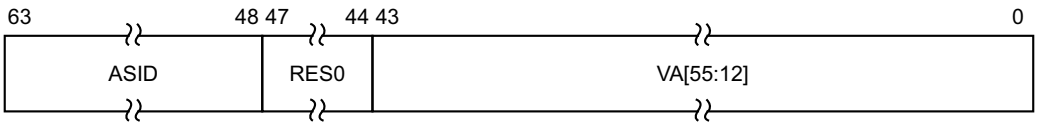
There are no configuration notes.

**Attributes**

TLBI VAE1 is a 64-bit system operation.

#### Field descriptions

The TLBI VAE1 input value bit assignments are:




**ASID, bits [63:48]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

**Bits [47:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

### Performing the TLBI VAE1 operation:

To perform the TLBI VAE1 operation:

```
TLBI VAE1, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0111 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.18 TLBI VAE1IS, TLB Invalidate by VA, EL1, Inner Shareable

The TLBI VAE1IS characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 1 TLB entries for the given VA and ASID, and the current VMID, on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see VA.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

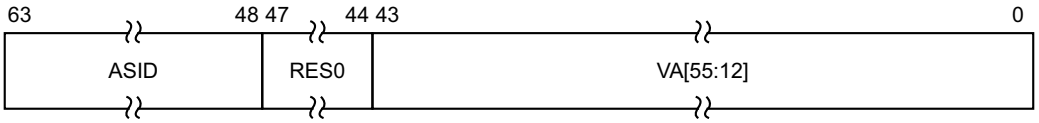
There are no configuration notes.

**Attributes**

TLBI VAE1IS is a 64-bit system operation.

#### Field descriptions

The TLBI VAE1IS input value bit assignments are:




**ASID, bits [63:48]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

**Bits [47:44]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

### Performing the TLBI VAE1IS operation:

To perform the TLBI VAE1IS operation:

```
TLBI VAE1IS, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.19 TLBI VAE2, TLB Invalidate by VA, EL2

The TLBI VAE2 characteristics are:

**Purpose**

Invalidate EL2 regime stage 1 TLB entries for the given VA.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | - |

Performing this operation from EL3 is UNDEFINED if EL2 does not exist.

For details of the scope of this instruction see VA.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

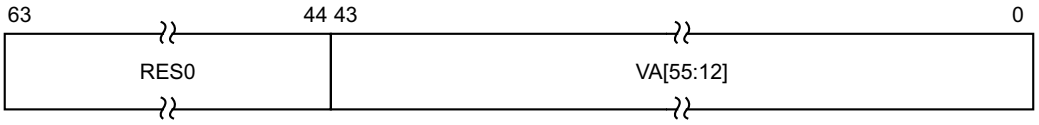
There are no configuration notes.

**Attributes**

TLBI VAE2 is a 64-bit system operation.

#### Field descriptions

The TLBI VAE2 input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the TLBI VAE2 operation:

To perform the TLBI VAE2 operation:

```
TLBI VAE2, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0111 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.20 TLBI VAE2IS, TLB Invalidate by VA, EL2, Inner Shareable

The TLBI VAE2IS characteristics are:

**Purpose**

Invalidate EL2 regime stage 1 TLB entries for the given VA on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | - |

Performing this operation from EL3 is UNDEFINED if EL2 does not exist.

For details of the scope of this instruction see VA.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

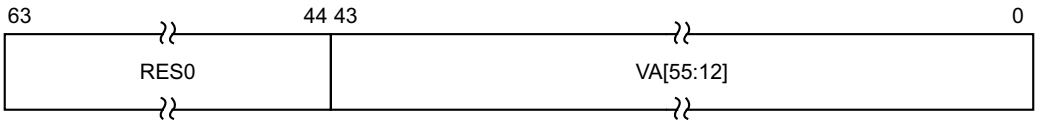
There are no configuration notes.

**Attributes**

TLBI VAE2IS is a 64-bit system operation.

#### Field descriptions

The TLBI VAE2IS input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the TLBI VAE2IS operation:

To perform the TLBI VAE2IS operation:

```
TLBI VAE2IS, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.21 TLBI VAE3, TLB Invalidate by VA, EL3

The TLBI VAE3 characteristics are:

**Purpose**

Invalidate EL3 regime stage 1 TLB entries for the given VA.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | WO | WO |

For details of the scope of this instruction see VA.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

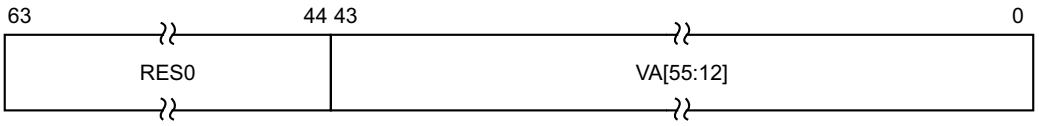
There are no configuration notes.

**Attributes**

TLBI VAE3 is a 64-bit system operation.

#### Field descriptions

The TLBI VAE3 input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

#### Performing the TLBI VAE3 operation:

To perform the TLBI VAE3 operation:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
TLBI VAE3, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 110 | 1000 | 0111 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.22 TLBI VAE3IS, TLB Invalidate by VA, EL3, Inner Shareable

The TLBI VAE3IS characteristics are:

**Purpose**

Invalidate EL3 regime stage 1 TLB entries for the given VA on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | WO | WO |

For details of the scope of this instruction see VA.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

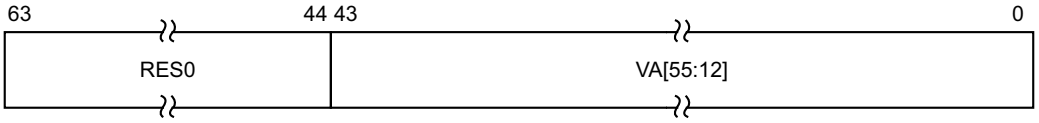
There are no configuration notes.

**Attributes**

TLBI VAE3IS is a 64-bit system operation.

#### Field descriptions

The TLBI VAE3IS input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

#### Performing the TLBI VAE3IS operation:

To perform the TLBI VAE3IS operation:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
TLBI VAE3IS, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 110 | 1000 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.23 TLBI VALE1, TLB Invalidate by VA, Last level, EL1

The TLBI VALE1 characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 1 TLB entries for the last level of translation table walk, the given VA and ASID, and the current VMID.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see VAL.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

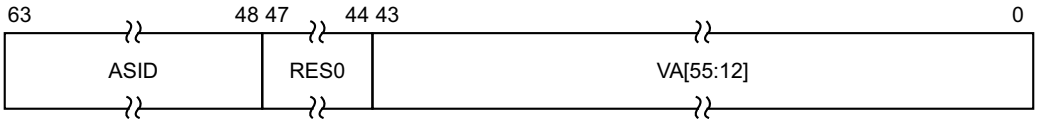
There are no configuration notes.

**Attributes**

TLBI VALE1 is a 64-bit system operation.

#### Field descriptions

The TLBI VALE1 input value bit assignments are:




**ASID, bits [63:48]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

**Bits [47:44]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

### Performing the TLBI VALE1 operation:

To perform the TLBI VALE1 operation:

```
TLBI VALE1, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0111 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.24 TLBI VALE1IS, TLB Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI VALE1IS characteristics are:

**Purpose**

Invalidate EL1&0 regime stage 1 TLB entries for the last level of translation table walk, the given VA and ASID, and the current VMID, on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see VAL.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

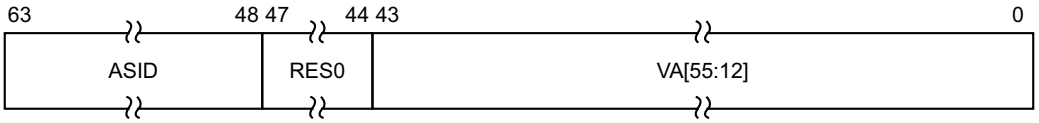
There are no configuration notes.

**Attributes**

TLBI VALE1IS is a 64-bit system operation.

#### Field descriptions

The TLBI VALE1IS input value bit assignments are:




**ASID, bits [63:48]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, but only 8 bits are being used in the context being invalidated, the upper bits are considered RES0 and must be written to 0 by software performing the TLB maintenance.

**Bits [47:44]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

### Performing the TLBI VALE1IS operation:

To perform the TLBI VALE1IS operation:

```
TLBI VALE1IS, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0011 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.25 TLBI VALE2, TLB Invalidate by VA, Last level, EL2

The TLBI VALE2 characteristics are:

**Purpose**

Invalidate EL2 regime stage 1 TLB entries for the last level of translation table walk and the given VA.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | - |

Performing this operation from EL3 is UNDEFINED if EL2 does not exist.

For details of the scope of this instruction see VAL.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

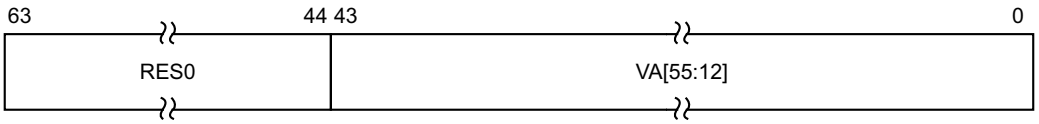
There are no configuration notes.

**Attributes**

TLBI VALE2 is a 64-bit system operation.

#### Field descriptions

The TLBI VALE2 input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the TLBI VALE2 operation:

To perform the TLBI VALE2 operation:

```
TLBI VALE2, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0111 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.26 TLBI VALE2IS, TLB Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI VALE2IS characteristics are:

**Purpose**

Invalidate EL2 regime stage 1 TLB entries for the last level of translation table walk and the given VA on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | - |

Performing this operation from EL3 is UNDEFINED if EL2 does not exist.

For details of the scope of this instruction see VAL.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

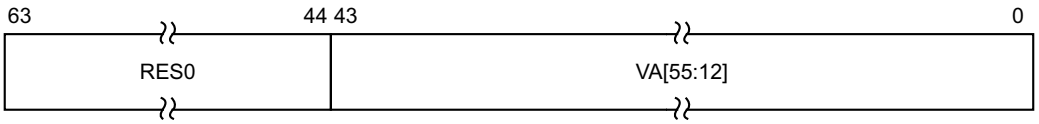
There are no configuration notes.

**Attributes**

TLBI VALE2IS is a 64-bit system operation.

#### Field descriptions

The TLBI VALE2IS input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the TLBI VALE2IS operation:

To perform the TLBI VALE2IS operation:

```
TLBI VALE2IS, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0011 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.27 TLBI VALE3, TLB Invalidate by VA, Last level, EL3

The TLBI VALE3 characteristics are:

**Purpose**

Invalidate EL3 regime stage 1 TLB entries for the last level of translation table walk and the given VA.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | WO | WO |

For details of the scope of this instruction see VAL.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

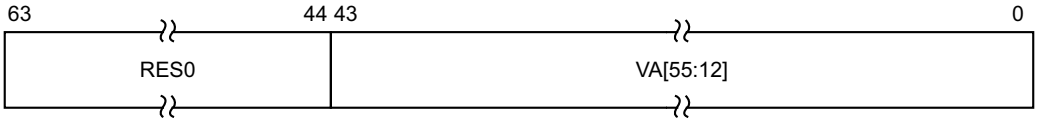
There are no configuration notes.

**Attributes**

TLBI VALE3 is a 64-bit system operation.

#### Field descriptions

The TLBI VALE3 input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

#### Performing the TLBI VALE3 operation:

To perform the TLBI VALE3 operation:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
TLBI VALE3, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 110 | 1000 | 0111 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.28 TLBI VALE3IS, TLB Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI VALE3IS characteristics are:

**Purpose**

Invalidate EL3 regime stage 1 TLB entries for the last level of translation table walk and the given VA on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | WO | WO |

For details of the scope of this instruction see VAL.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

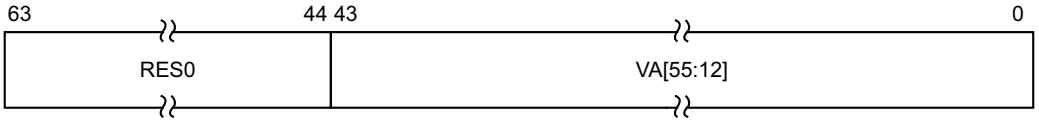
There are no configuration notes.

**Attributes**

TLBI VALE3IS is a 64-bit system operation.

#### Field descriptions

The TLBI VALE3IS input value bit assignments are:




**Bits [63:44]**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this operation.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

#### Performing the TLBI VALE3IS operation:

To perform the TLBI VALE3IS operation:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
TLBI VALE3IS, <Xt>
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 110 | 1000 | 0011 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.29 TLBI VMALLE1, TLB Invalidate by VMID, All at stage 1, EL1

The TLBI VMALLE1 characteristics are:

**Purpose**

Invalidate all EL1&0 regime stage 1 TLB entries for the current VMID.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see VMALL.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VMALLE1 is a 64-bit system operation.

#### Field descriptions

The TLBI VMALLE1 operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBI VMALLE1 operation:

To perform the TLBI VMALLE1 operation:

```
TLBI VMALLE1
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0111 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.30 TLBI VMALLE1IS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable

The TLBI VMALLE1IS characteristics are:

**Purpose**

Invalidate all EL1&0 regime stage 1 TLB entries for the current VMID on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

If EL3 is implemented, the value of SCR_EL3.NS determines whether the instruction invalidates the translations that are associated with Secure address space, or invalidates the translations associated with the Non-secure address space.

For details of the scope of this instruction see VMALL.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VMALLE1IS is a 64-bit system operation.

#### Field descriptions

The TLBI VMALLE1IS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBI VMALLE1IS operation:

To perform the TLBI VMALLE1IS operation:

```
TLBI VMALLE1IS
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 000 | 1000 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.31 TLBI VMALLS12E1, TLB Invalidate by VMID, All at Stage 1 and 2, EL1

The TLBI VMALLS12E1 characteristics are:

**Purpose**

Invalidate all EL1&0 regime stage 1 and 2 TLB entries for the current VMID.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If this instruction is executed at EL3 when the value of SCR_EL3.NS is 0, this instruction executes as a TLBI VMALLE1.

For details of the scope of this instruction see VMALLS12.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VMALLS12E1 is a 64-bit system operation.

#### Field descriptions

The TLBI VMALLS12E1 operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBI VMALLS12E1 operation:

To perform the TLBI VMALLS12E1 operation:

```
TLBI VMALLS12E1
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0111 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C5.5.32 TLBI VMALLS12E1IS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable

The TLBI VMALLS12E1IS characteristics are:

**Purpose**

Invalidate all EL1&0 regime stage 1 and 2 TLB entries for the current VMID on all PEs in the same Inner Shareable domain.

**Usage constraints**

This operation can be performed at the following exception levels:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If this instruction is executed at EL3 when the value of SCR_EL3.NS is 0, this instruction executes as a TLBI VMALLE1IS.

For details of the scope of this instruction see VMALLS12.

**Traps and Enables**

There are no traps or enables affecting this operation.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VMALLS12E1IS is a 64-bit system operation.

#### Field descriptions

The TLBI VMALLS12E1IS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBI VMALLS12E1IS operation:

To perform the TLBI VMALLS12E1IS operation:

```
TLBI VMALLS12E1IS
```

The operation is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 01 | 100 | 1000 | 0011 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter C6
# A64 Base Instruction Descriptions

This chapter describes the A64 base instructions.

It contains the following sections:


- *Introduction* on page C6-426.
- *Register size* on page C6-427.
- *Use of the PC* on page C6-428.
- *Use of the stack pointer* on page C6-429.
- *Condition flags and related instructions* on page C6-430.
- *Alphabetical list of instructions* on page C6-431.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## C6.1 Introduction

This chapter provides information on key aspects of the base instructions, and an alphabetic list of instructions from the following functional groups:

- Branch, Exception generation, and system instructions.
- Loads and stores associated with the general-purpose registers.
- Data processing (immediate).
- Data processing (register).

*A64 instruction index by encoding* on page C4-186 provides an overview of the instruction encodings as well as of the instruction classes within their functional groups.

The base instruction descriptions include:

- *Register size* on page C6-427.
- *Use of the PC* on page C6-428.
- *Use of the stack pointer* on page C6-429.
- *Condition flags and related instructions* on page C6-430.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.2 Register size

Most data processing, comparison, and conversion instructions that use the general-purpose registers as the source or destination operand have two instruction variants that operate on either a 32-bit or a 64-bit value.

Where a 32-bit instruction form is selected, the following holds:

- The upper 32 bits of the source registers are ignored.
- The upper 32 bits of the destination register are set to zero.
- Right shifts and right rotates inject at bit[31], not at bit[63].
- The condition flags, where set by the instruction, are computed from the lower 32 bits.

This distinction applies even when the results of a 32-bit instruction form are indistinguishable from the lower 32 bits computed by the equivalent 64-bit instruction form. For example, a 32-bit bitwise `ORR` could be performed using a 64-bit `ORR` and simply ignoring the top 32 bits of the result. However, the A64 instruction set includes separate 32-bit and 64-bit forms of the `ORR` instruction.

As well as distinct sign-extend or zero-extend instructions, the A64 instruction set also provides the ability to extend and shift the final source register of an `ADD`, `SUB`, `ADDS`, `or SUBS` instruction and the index register of a Load/Store instruction. This enables array index calculations involving a 64-bit array pointer and a 32-bit array index to be implemented efficiently.

The assembly language notation enables the distinct identification of registers holding 32-bit values and registers holding 64-bit values. See *Register names* on page C1-122 and *Register indexed addressing* on page C1-126.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## C6.3 Use of the PC

A64 instructions have limited access to the PC. The only instructions that can read the PC are those that generate a PC relative address:

- ADR and ADRP.
- The Load register (literal) instruction class.
- Direct branches that use an immediate offset.
- The unconditional branch with link instructions, BL and BLR, that use the PC to create the return link address.

Only explicit control flow instructions can modify the PC:

- Conditional and unconditional branch and return instructions.
- Exception generation and exception return instructions.

For more details on instructions that can modify the PC, see *Branches, Exception generating, and System instructions* on page C3-138.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## C6.4 Use of the stack pointer

A64 instructions can use the stack pointer only in a limited number of cases:

- Load/Store instructions use the current stack pointer as the base address:
  - When stack alignment checking is enabled by system software and the base register is SP, the current stack pointer must be initially quadword aligned, That is, it must be aligned to 16 bytes. Misalignment generates a Stack Alignment fault. See *Stack pointer alignment checking* on page D1-1531 for more information.
- Add and subtract data processing instructions in their immediate and extended register forms, use the current stack pointer as a source register or the destination register or both.
- Logical data processing instructions in their immediate form use the current stack pointer as the destination register.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## C6.5 Condition flags and related instructions

The A64 base instructions that use the condition flags as an input are:

- Conditional branch. The conditional branch instruction is `B.cond`.
- Add or subtract with carry. These instruction types include instructions to perform multi-precision arithmetic and calculate checksums. The add or subtract with carry instructions are `ADC`, `ADCS`, `SBC`, and `SBCS`, or an architectural alias for these instructions.
- Conditional select with increment, negate, or invert.This instruction type conditionally selects between one source register and a second, incremented, negated, inverted, or unmodified source register. The conditional select with increment, negate, or invert instructions are `CSINC`, `CSINV`, and `CSNEG`.

  These instructions also implement:

  — Conditional select or move. The condition flags select one of two source registers as the destination register. Short conditional sequences can be replaced by unconditional instructions followed by a conditional select, `CSEL`.

  — Conditional set. Conditionally selects between 0 and 1, or 0 and -1. This can be used to convert the condition flags to a Boolean value or mask in a general-purpose register, for example. These instructions include `CSET` and `CSETM`.
- Conditional compare. This instruction type sets the condition flags to the result of a comparison if the original condition is true, otherwise it sets the condition flags to an immediate value. It permits the flattening of nested conditional expressions without using conditional branches or performing Boolean arithmetic within the general-purpose registers.The conditional compare instructions are `CCMP` and `CCMN`.

The A64 base instructions that update the condition flags as an output are:

- Flag-setting data processing instructions, such as `ADCS`, `ADDS`, `ANDS`, `BICS`, `SBCS`, and `SUBS`, and the aliases `CMN`, `CMP`, and `TST`.
- Conditional compare instructions such as `CCMN`, `CCMP`.

The flags can be directly accessed for a read/write using the *NZCV, Condition Flags* on page C5-306.

The A64 base instructions also include conditional branch instructions that do not use the condition flags as an input:

- Compare and branch if a register is zero or nonzero, `CBZ` and `CBNZ`.
- Test a single bit in a register and branch if the bit is zero or nonzero, `TBZ` and `TBNZ`.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

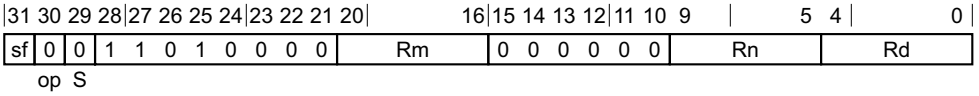
## C6.6 Alphabetical list of instructions

This section lists every instruction in the base category of the A64 instruction set. For details of the format used, see *Understanding the A64 instruction descriptions* on page C2-132.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## C6.6.1 ADC

Add with Carry adds two register values and the Carry flag value, and writes the result to the destination register.




### *32-bit variant*

Applies when `sf = 0`.

```
ADC <Wd>, <Wn>, <Wm>
```

### *64-bit variant*

Applies when `sf = 1`.

```
ADC <Xd>, <Xn>, <Xm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
```

### Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |

### Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;

if sub_op then
    operand2 = NOT(operand2);

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

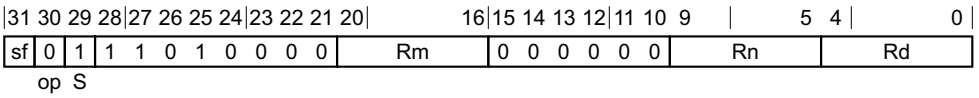
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.2 ADCS

Add with Carry, setting flags, adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.




### 32-bit variant

Applies when `sf = 0`.

```
ADCS <Wd>, <Wn>, <Wm>
```

### 64-bit variant

Applies when `sf = 1`.

```
ADCS <Xd>, <Xn>, <Xm>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
```

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;

if sub_op then
    operand2 = NOT(operand2);

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

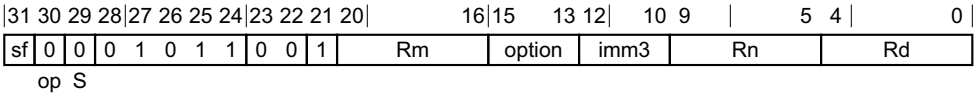
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.3 ADD (extended register)

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.




#### *32-bit variant*

Applies when `sf = 0`.

```
ADD <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

#### *64-bit variant*

Applies when `sf = 1`.

```
ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

### Assembler symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

- `W` when `option = 00x`
- `W` when `option = 010`
- `X` when `option = x11`
- `W` when `option = 10x`
- `W` when `option = 110`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

if d == 31 && !setflags then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

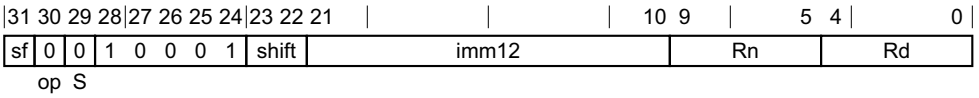
```
        SP[] = result;
    else
        X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.4 ADD (immediate)

Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (to/from SP). See the *Alias conditions* table for details of when each alias is preferred.




### *32-bit variant*

Applies when `sf = 0`.

```
ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

### *64-bit variant*

Applies when `sf = 1`.

```
ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
    when '1x' ReservedValue();
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| MOV (to/from SP) | `shift == '00' && imm12 == '000000000000' && (Rd == '11111' \|\| Rn == '11111')` |

## Assembler symbols

`<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "shift" field. It can have the following values:

LSL #0 when shift = 00

LSL #12 when shift = 01

The encoding shift = 1x is reserved.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

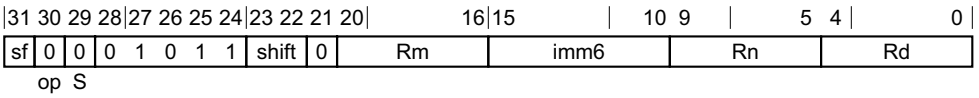
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.5 ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.




#### *32-bit variant*

Applies when `sf = 0`.

```
ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

#### *64-bit variant*

Applies when `sf = 1`.

```
ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

### Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

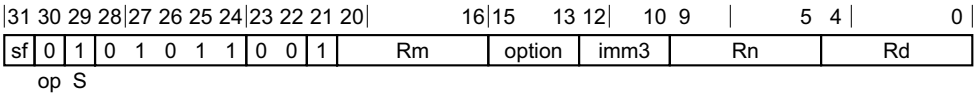
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.6 ADDS (extended register)

Add (extended register), setting flags, adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMN (extended register). See the *Alias conditions* table for details of when each alias is preferred.




#### *32-bit variant*

Applies when sf = 0.

```
ADDS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

#### *64-bit variant*

Applies when sf = 1.

```
ADDS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

### Alias conditions

| Alias | is preferred when |
|---|---|
| CMN (extended register) | Rd == '11111' |

### Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

W when option = 00x

W when option = 010

X when option = x11

W when option = 10x

W when option = 110

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

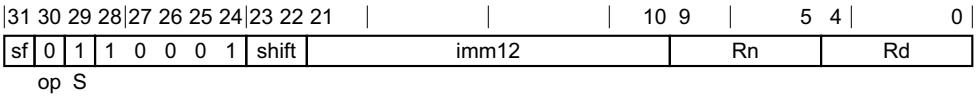
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.7 ADDS (immediate)

Add (immediate), setting flags, adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (immediate). See the *Alias conditions* table for details of when each alias is preferred.




#### *32-bit variant*

Applies when `sf = 0`.

```
ADDS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

#### *64-bit variant*

Applies when `sf = 1`.

```
ADDS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
    when '1x' ReservedValue();
```

### Alias conditions

| Alias | is preferred when |
|---|---|
| CMN (immediate) | `Rd == '11111'` |

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn\|WSP>` | Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn\|SP>` | Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field. |
| `<imm>` | Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field. |
| `<shift>` | Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "shift" field. It can have the following values:<br>`LSL #0` when `shift = 00` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`LSL #12` when `shift = 01`

The encoding `shift = 1x` is reserved.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

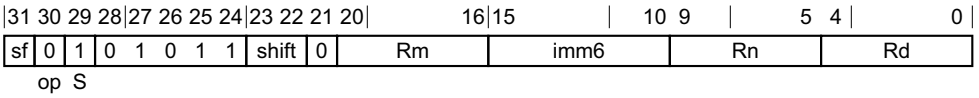
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.8 ADDS (shifted register)

Add (shifted register), setting flags, adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (shifted register). See the *Alias conditions* table for details of when each alias is preferred.




#### *32-bit variant*

Applies when `sf = 0`.

```
ADDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

#### *64-bit variant*

Applies when `sf = 1`.

```
ADDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

### Alias conditions

| Alias | is preferred when |
|---|---|
| CMN (shifted register) | `Rd == '11111'` |

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

LSL when shift = 00

LSR when shift = 01

ASR when shift = 10

The encoding shift = 11 is reserved.

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

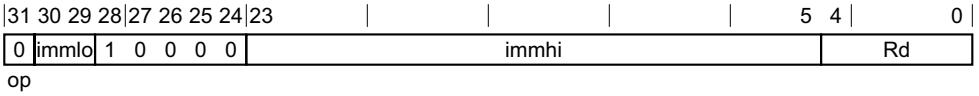
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.9 ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.




#### *Literal variant*

```
ADR <Xd>, <label>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
boolean page = (op == '1');
bits(64) imm;

if page then
    imm = SignExtend(immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(immhi:immlo, 64);
```

### Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<label> Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in "immhi:immlo".

### Operation

```
bits(64) base = PC[];

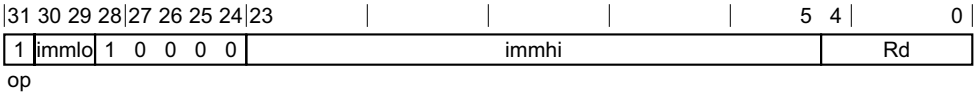
if page then
    base<11:0> = Zeros(12);

X[d] = base + imm;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.10 ADRP

Form PC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.




### *Literal variant*

```
ADRP <Xd>, <label>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
boolean page = (op == '1');
bits(64) imm;

if page then
    imm = SignExtend(immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(immhi:immlo, 64);
```

### Assembler symbols

| | |
|---|---|
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <label> | Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as "immhi:immlo" times 4096. |

### Operation

```
bits(64) base = PC[];

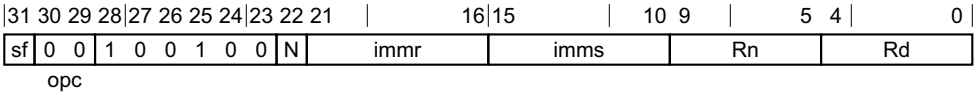
if page then
    base<11:0> = Zeros(12);

X[d] = base + imm;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.11 AND (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.




#### *32-bit variant*

Applies when `sf = 0 && N = 0`.

```
AND <Wd|WSP>, <Wn>, #<imm>
```

#### *64-bit variant*

Applies when `sf = 1`.

```
AND <Xd|SP>, <Xn>, #<imm>
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Assembler symbols

| | |
|---|---|
| <Wd\|WSP> | Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| <Xd\|SP> | Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| <imm> | For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".<br>For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr". |

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

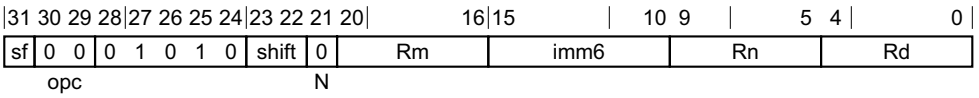
if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.12 AND (shifted register)

Bitwise AND (shifted register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.




### 32-bit variant

Applies when `sf = 0`.

```
AND <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit variant

Applies when `sf = 1`.

```
AND <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

## Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

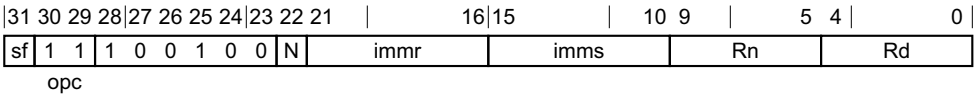
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.13 ANDS (immediate)

Bitwise AND (immediate), setting flags, performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (immediate). See the *Alias conditions* table for details of when each alias is preferred.




#### *32-bit variant*

Applies when `sf = 0 && N = 0`.

```
ANDS <Wd>, <Wn>, #<imm>
```

#### *64-bit variant*

Applies when `sf = 1`.

```
ANDS <Xd>, <Xn>, #<imm>
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

### Alias conditions

| Alias | is preferred when |
|---|---|
| TST (immediate) | `Rd == '11111'` |

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<imm>` | For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".<br>For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr". |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

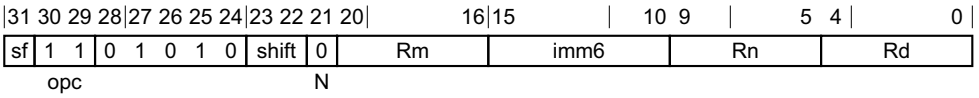
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.14 ANDS (shifted register)

Bitwise AND (shifted register), setting flags, performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (shifted register). See the *Alias conditions* table for details of when each alias is preferred.




#### *32-bit variant*

Applies when `sf = 0`.

```
ANDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

#### *64-bit variant*

Applies when `sf = 1`.

```
ANDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

### Alias conditions

| Alias | is preferred when |
|---|---|
| TST (shifted register) | `Rd == '11111'` |

### Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

LSL when shift = 00

LSR when shift = 01

ASR when shift = 10

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

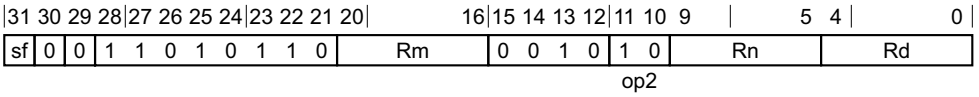

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.15 ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is an alias of the ASRV instruction. This means that:

- The encodings in this description are named to match the encodings of ASRV.
- The description of ASRV gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0`.

`ASR <Wd>, <Wn>, <Wm>`

is equivalent to

`ASRV <Wd>, <Wn>, <Wm>`

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1`.

`ASR <Xd>, <Xn>, <Xm>`

is equivalent to

`ASRV <Xd>, <Xn>, <Xm>`

and is always the preferred disassembly.

## Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

## Operation

The description of ASRV gives the operational pseudocode for this instruction.

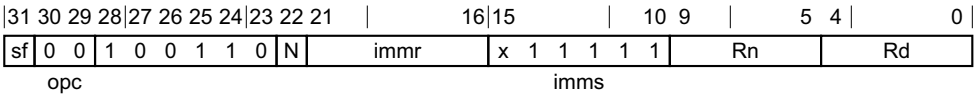

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.16 ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of the sign bit in the upper bits and zeros in the lower bits, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0 && N = 0 && imms = 011111`.

`ASR <Wd>, <Wn>, #<shift>`

is equivalent to

`SBFM <Wd>, <Wn>, #<shift>, #31`

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1 && N = 1 && imms = 111111`.

`ASR <Xd>, <Xn>, #<shift>`

is equivalent to

`SBFM <Xd>, <Xn>, #<shift>, #63`

and is always the preferred disassembly.

#### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<shift>` | For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field.<br>For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field. |

#### Operation

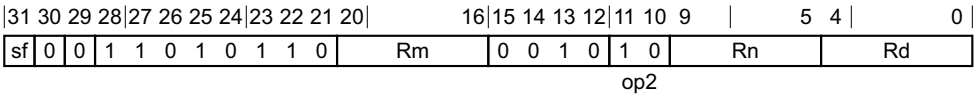
The description of SBFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.17 ASRV

Arithmetic Shift Right Variable shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ASR (register). The alias is always the preferred disassembly.




#### *32-bit variant*

Applies when `sf = 0`.

```
ASRV <Wd>, <Wn>, <Wm>
```

#### *64-bit variant*

Applies when `sf = 1`.

```
ASRV <Xd>, <Xn>, <Xm>
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

### Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field. |

### Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

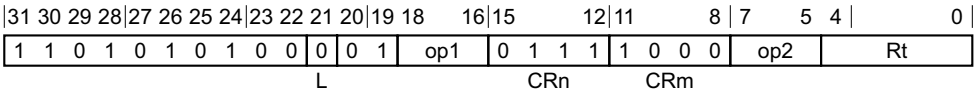

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.18 AT

Address Translate. For more information, see A64 system instructions for address translation.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.




### *System variant*

`AT <at_op>, <Xt>`

is equivalent to

`SYS #<op1>, C7, C8, #<op2>, <Xt>`

and is the preferred disassembly when `SysOp(op1,'0111','1000',op2) == Sys_AT`.

### Assembler symbols

`<at_op>` Is an AT operation name, as listed for the AT system operation group, encoded in the "op1:op2" field. It can have the following values:

| | |
|---|---|
| `S1E1R` | when `op1 = 000`, `op2 = 000` |
| `S1E1W` | when `op1 = 000`, `op2 = 001` |
| `S1E0R` | when `op1 = 000`, `op2 = 010` |
| `S1E0W` | when `op1 = 000`, `op2 = 011` |
| `S1E2R` | when `op1 = 100`, `op2 = 000` |
| `S1E2W` | when `op1 = 100`, `op2 = 001` |
| `S12E1R` | when `op1 = 100`, `op2 = 100` |
| `S12E1W` | when `op1 = 100`, `op2 = 101` |
| `S12E0R` | when `op1 = 100`, `op2 = 110` |
| `S12E0W` | when `op1 = 100`, `op2 = 111` |
| `S1E3R` | when `op1 = 110`, `op2 = 000` |
| `S1E3W` | when `op1 = 110`, `op2 = 001` |

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<Xt>` Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

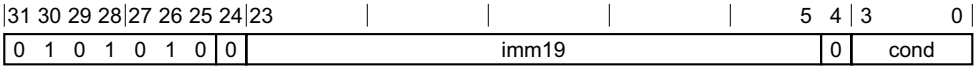
### Operation

The description of SYS gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.19 B.cond

Branch conditionally to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.




### *19-bit signed PC-relative branch offset variant*

```
B.<cond> <label>
```

### *Decode for this encoding*

```
bits(64) offset = SignExtend(imm19:'00', 64);
bits(4) condition = cond;
```

### Assembler symbols

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

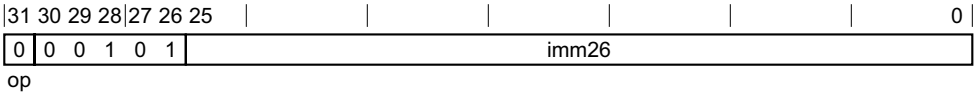
### Operation

```
if ConditionHolds(condition) then
    BranchTo(PC[] + offset, BranchType_JMP);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.20 B

Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.




### 26-bit signed PC-relative branch offset variant

```
B <label>
```

### Decode for this encoding

```
BranchType branch_type = if op == '1' then BranchType_CALL else BranchType_JMP;
bits(64) offset = SignExtend(imm26:'00', 64);
```

## Assembler symbols

<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

## Operation

```
if branch_type == BranchType_CALL then X[30] = PC[] + 4;

BranchTo(PC[] + offset, branch_type);
```

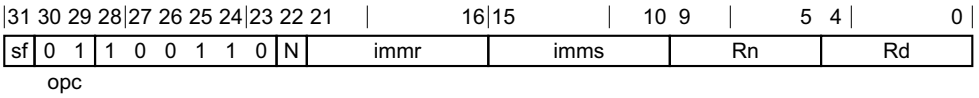

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.21 BFI

Bitfield Insert copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, leaving other bits unchanged.

This instruction is an alias of the BFM instruction. This means that:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0 && N = 0`.

```
BFI <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

### *64-bit variant*

Applies when `sf = 1 && N = 1`.

```
BFI <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<lsb>` | For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.<br>For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63. |
| `<width>` | For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.<br>For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>. |

### Operation

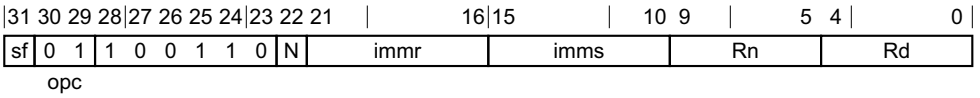
The description of BFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.22 BFM

Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, leaving other bits unchanged.

This instruction is used by the aliases BFI and BFXIL. See the *Alias conditions* on page C6-466 table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0 && N = 0`.

```
BFM <Wd>, <Wn>, #<immr>, #<imms>
```

### 64-bit variant

Applies when `sf = 1 && N = 1`.

```
BFM <Xd>, <Xn>, #<immr>, #<imms>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

case opc of
    when '00' inzero = TRUE;  extend = TRUE;    // SBFM
    when '01' inzero = FALSE; extend = FALSE;   // BFM
    when '10' inzero = TRUE;  extend = FALSE;   // UBFM
    when '11' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Alias conditions

| Alias | is preferred when |
|---|---|
| BFI | `UInt(imms) < UInt(immr)` |
| BFXIL | `UInt(imms) >= UInt(immr)` |

### Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

`<immr>` For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

`<imms>` For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

### Operation

```
bits(datasize) dst = if inzero then Zeros() else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

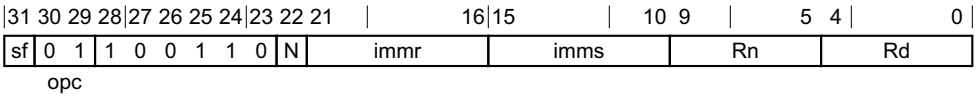

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.23 BFXIL

Bitfield extract and insert at low end copies any number of low-order bits from a source register into the same number of adjacent bits at the low end in the destination register, leaving other bits unchanged.

This instruction is an alias of the BFM instruction. This means that:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0 && N = 0`.

```
BFXIL <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

### *64-bit variant*

Applies when `sf = 1 && N = 1`.

```
BFXIL <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM  <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<lsb>` | For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.<br>For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63. |
| `<width>` | For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.<br>For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>. |

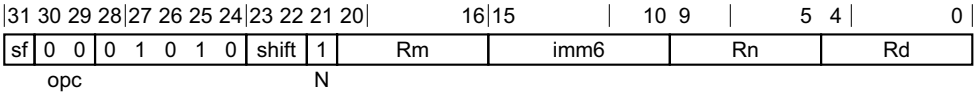
### Operation

The description of BFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.24 BIC (shifted register)

Bitwise Bit Clear (shifted register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.




#### *32-bit variant*

Applies when `sf = 0`.

```
BIC <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

#### *64-bit variant*

Applies when `sf = 1`.

```
BIC <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

### Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

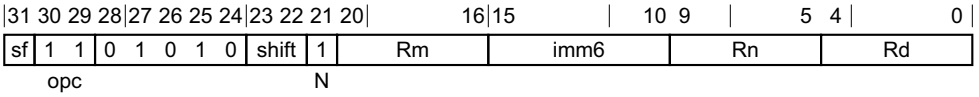
if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.25 BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags, performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.




### 32-bit variant

Applies when `sf = 0`.

```
BICS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit variant

Applies when `sf = 1`.

```
BICS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

## Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ASR when shift = 10

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

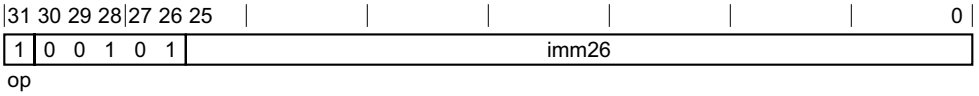
if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.26 BL

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.




### *26-bit signed PC-relative branch offset variant*

```
BL <label>
```

### *Decode for this encoding*

```
BranchType branch_type = if op == '1' then BranchType_CALL else BranchType_JMP;
bits(64) offset = SignExtend(imm26:'00', 64);
```

### Assembler symbols

| | |
|---|---|
| <label> | Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4. |

### Operation

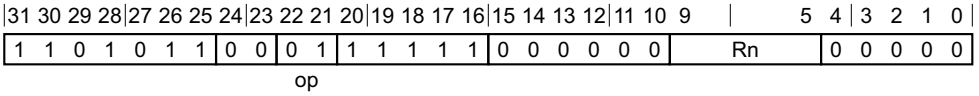
```
if branch_type == BranchType_CALL then X[30] = PC[] + 4;

BranchTo(PC[] + offset, branch_type);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.27 BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.




### *Integer variant*

```
BLR <Xn>
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
BranchType branch_type;

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();
```

## Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

## Operation

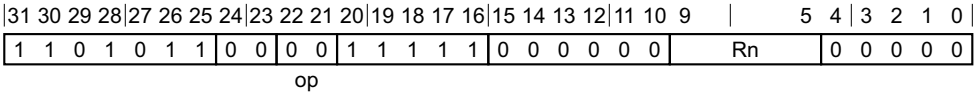
```
bits(64) target = X[n];

if branch_type == BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.28 BR

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.




### *Integer variant*

```
BR <Xn>
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
BranchType branch_type;

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();
```

### Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

### Operation

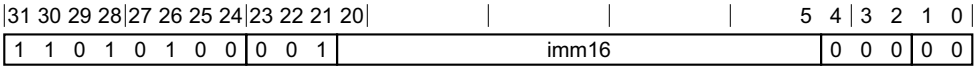
```
bits(64) target = X[n];

if branch_type == BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.29 BRK

Breakpoint instruction causes a Software Breakpoint Instruction exception. The PE records the exception in ESR_ELx, using the EC value 0x3c, and captures the value of the immediate argument in ESR_ELx.ISS.




### *System variant*

```
BRK #<imm>
```

### *Decode for this encoding*

```
bits(16) comment = imm16;
```

### Assembler symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

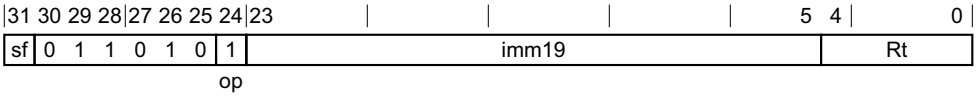
### Operation

```
AArch64.SoftwareBreakpoint(comment);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.30 CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.




### *32-bit variant*

Applies when `sf = 0`.

```
CBNZ <Wt>, <label>
```

### *64-bit variant*

Applies when `sf = 1`.

```
CBNZ <Xt>, <label>
```

### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field. |
| `<label>` | Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4. |

## Operation

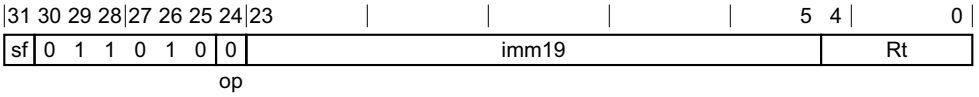
```
bits(datasize) operand1 = X[t];

if IsZero(operand1) == iszero then
    BranchTo(PC[] + offset, BranchType_JMP);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.31 CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.




### 32-bit variant

Applies when `sf = 0`.

```
CBZ <Wt>, <label>
```

### 64-bit variant

Applies when `sf = 1`.

```
CBZ <Xt>, <label>
```

### Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field. |
| `<label>` | Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4. |

## Operation

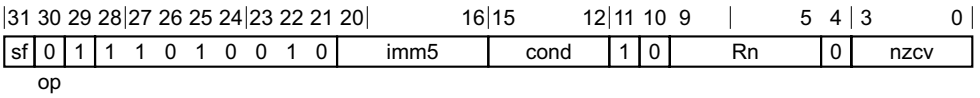
```
bits(datasize) operand1 = X[t];

if IsZero(operand1) == iszero then
    BranchTo(PC[] + offset, BranchType_JMP);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.32 CCMN (immediate)

Conditional Compare Negative (immediate) sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.




### *32-bit variant*

Applies when `sf = 0`.

```
CCMN <Wn>, #<imm>, #<nzcv>, <cond>
```

### *64-bit variant*

Applies when `sf = 1`.

```
CCMN <Xn>, #<imm>, #<nzcv>, <cond>
```

### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

## Assembler symbols

| | |
|---|---|
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <imm> | Is a five bit unsigned (positive) immediate encoded in the "imm5" field. |
| <nzcv> | Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field. |
| <cond> | Is one of the standard conditions, encoded in the "cond" field in the standard way. |

## Operation

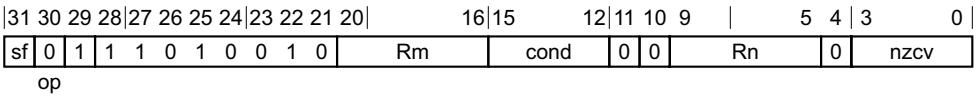
```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;
bit carry_in = '0';

if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.33 CCMN (register)

Conditional Compare Negative (register) sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.




### *32-bit variant*

Applies when `sf = 0`.

```
CCMN <Wn>, <Wm>, #<nzcv>, <cond>
```

### *64-bit variant*

Applies when `sf = 1`.

```
CCMN <Xn>, <Xm>, #<nzcv>, <cond>
```

### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

## Assembler symbols

| | |
|---|---|
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <nzcv> | Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field. |
| <cond> | Is one of the standard conditions, encoded in the "cond" field in the standard way. |

## Operation

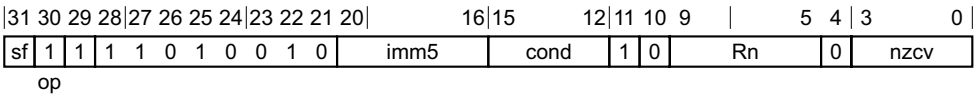
```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bit carry_in = '0';

if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.34 CCMP (immediate)

Conditional Compare (immediate) sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.




#### *32-bit variant*

Applies when `sf = 0`.

```
CCMP <Wn>, #<imm>, #<nzcv>, <cond>
```

#### *64-bit variant*

Applies when `sf = 1`.

```
CCMP <Xn>, #<imm>, #<nzcv>, <cond>
```

#### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

### Assembler symbols

| | |
|---|---|
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <imm> | Is a five bit unsigned (positive) immediate encoded in the "imm5" field. |
| <nzcv> | Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field. |
| <cond> | Is one of the standard conditions, encoded in the "cond" field in the standard way. |

### Operation

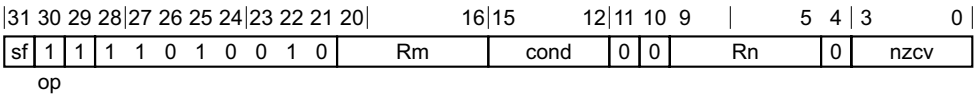
```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;
bit carry_in = '0';

if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.35 CCMP (register)

Conditional Compare (register) sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.




### *32-bit variant*

Applies when `sf = 0`.

```
CCMP <Wn>, <Wm>, #<nzcv>, <cond>
```

### *64-bit variant*

Applies when `sf = 1`.

```
CCMP <Xn>, <Xm>, #<nzcv>, <cond>
```

### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

## Assembler symbols

| | |
|---|---|
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<nzcv>` | Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field. |
| `<cond>` | Is one of the standard conditions, encoded in the "cond" field in the standard way. |

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bit carry_in = '0';

if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

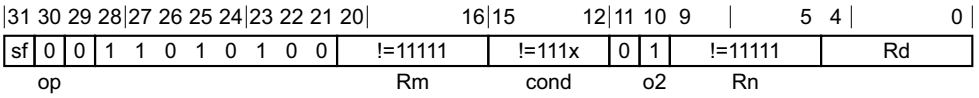

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.36 CINC

Conditional Increment returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSINC instruction. This means that:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0`.

`CINC <Wd>, <Wn>, <cond>`

is equivalent to

`CSINC <Wd>, <Wn>, <Wn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

#### *64-bit variant*

Applies when `sf = 1`.

`CINC <Xd>, <Xn>, <cond>`

is equivalent to

`CSINC <Xd>, <Xn>, <Xn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields. |
| `<cond>` | Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted. |

### Operation

The description of CSINC gives the operational pseudocode for this instruction.

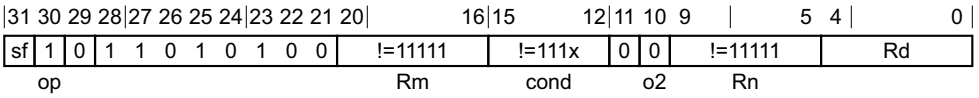

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.37 CINV

Conditional Invert returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSINV instruction. This means that:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

`CINV <Wd>, <Wn>, <cond>`

is equivalent to

`CSINV <Wd>, <Wn>, <Wn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

### *64-bit variant*

Applies when `sf = 1`.

`CINV <Xd>, <Xn>, <cond>`

is equivalent to

`CSINV <Xd>, <Xn>, <Xn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields. |
| `<cond>` | Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted. |

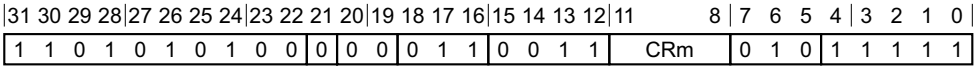
## Operation

The description of CSINV gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.38 CLREX

Clear Exclusive clears the local monitor of the executing PE.




#### *System variant*

```
CLREX {#<imm>}
```

#### *Decode for this encoding*

```
// CRm field is ignored
```

#### Assembler symbols

<imm> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

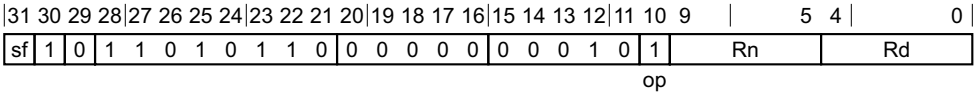
#### Operation

```
ClearExclusiveLocal(ProcessorID());
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.39 CLS

Count leading sign bits : `Rd = CLS(Rn)`




### *32-bit variant*

Applies when `sf = 0`.

```
CLS <Wd>, <Wn>
```

### *64-bit variant*

Applies when `sf = 1`.

```
CLS <Xd>, <Xn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
CountOp opcode = if op == '0' then CountOp_CLZ else CountOp_CLS;
```

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |

## Operation

```
integer result;
bits(datasize) operand1 = X[n];

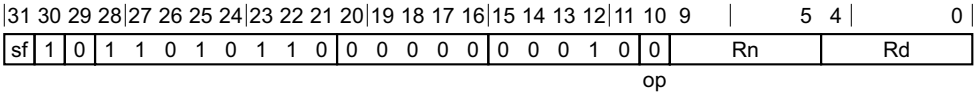
if opcode == CountOp_CLZ then
    result = CountLeadingZeroBits(operand1);
else
    result = CountLeadingSignBits(operand1);

X[d] = result<datasize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.40 CLZ

Count leading zero bits : `Rd = CLZ(Rn)`




### *32-bit variant*

Applies when `sf = 0`.

```
CLZ <Wd>, <Wn>
```

### *64-bit variant*

Applies when `sf = 1`.

```
CLZ <Xd>, <Xn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
CountOp opcode = if op == '0' then CountOp_CLZ else CountOp_CLS;
```

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |

## Operation

```
integer result;
bits(datasize) operand1 = X[n];

if opcode == CountOp_CLZ then
    result = CountLeadingZeroBits(operand1);
else
    result = CountLeadingSignBits(operand1);

X[d] = result<datasize-1:0>;
```

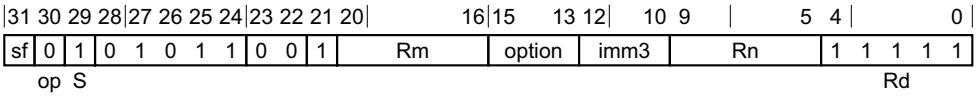

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.41 CMN (extended register)

Compare Negative (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ADDS (extended register) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

```
CMN <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

is equivalent to

```
ADDS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

```
CMN <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

is equivalent to

```
ADDS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

### Assembler symbols

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

| | |
|---|---|
| W | when `option = 00x` |
| W | when `option = 010` |
| X | when `option = x11` |
| W | when `option = 10x` |
| W | when `option = 110` |

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

The description of ADDS (extended register) gives the operational pseudocode for this instruction.

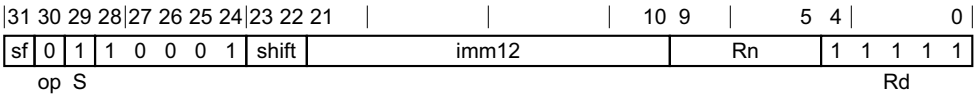

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.42 CMN (immediate)

Compare Negative (immediate) adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ADDS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

`CMN <Wn|WSP>, #<imm>{, <shift>}`

is equivalent to

`ADDS WZR, <Wn|WSP>, #<imm> {, <shift>}`

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

`CMN <Xn|SP>, #<imm>{, <shift>}`

is equivalent to

`ADDS XZR, <Xn|SP>, #<imm> {, <shift>}`

and is always the preferred disassembly.

### Assembler symbols

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "shift" field. It can have the following values:

`LSL #0` when `shift = 00`

`LSL #12` when `shift = 01`

The encoding `shift = 1x` is reserved.

### Operation

The description of ADDS (immediate) gives the operational pseudocode for this instruction.

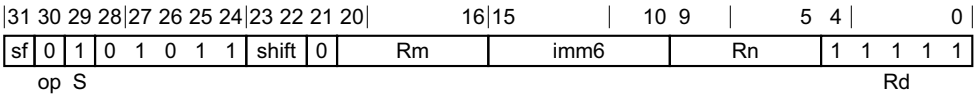

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.43 CMN (shifted register)

Compare Negative (shifted register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ADDS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0`.

```
CMN <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS WZR, <Wn>, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1`.

```
CMN <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS XZR, <Xn>, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

### Assembler symbols

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL` when `shift = 00`

`LSR` when `shift = 01`

`ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

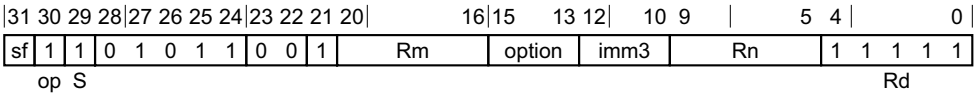

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.44 CMP (extended register)

Compare (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the SUBS (extended register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (extended register).
- The description of SUBS (extended register) gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

```
CMP <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

is equivalent to

```
SUBS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

```
CMP <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

is equivalent to

```
SUBS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

### Assembler symbols

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

| | |
|---|---|
| W | when `option = 00x` |
| W | when `option = 010` |
| X | when `option = x11` |
| W | when `option = 10x` |
| W | when `option = 110` |

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

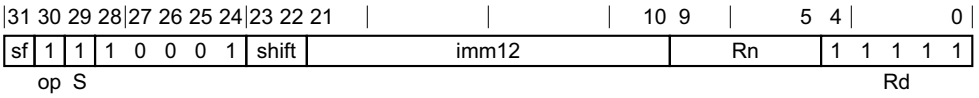

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.45 CMP (immediate)

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the SUBS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

```
CMP <Wn|WSP>, #<imm>{, <shift>}
```

is equivalent to

```
SUBS WZR, <Wn|WSP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

```
CMP <Xn|SP>, #<imm>{, <shift>}
```

is equivalent to

```
SUBS XZR, <Xn|SP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

### Assembler symbols

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "shift" field. It can have the following values:

`LSL #0` when `shift = 00`

`LSL #12` when `shift = 01`

The encoding `shift = 1x` is reserved.

### Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

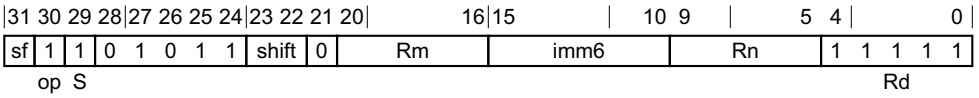

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.46 CMP (shifted register)

Compare (shifted register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the SUBS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

```
CMP <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS WZR, <Wn>, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

```
CMP <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS XZR, <Xn>, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

### Assembler symbols

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

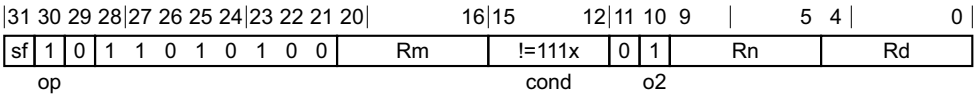

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.47 CNEG

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSNEG instruction. This means that:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

`CNEG <Wd>, <Wn>, <cond>`

is equivalent to

`CSNEG <Wd>, <Wn>, <Wn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

### *64-bit variant*

Applies when `sf = 1`.

`CNEG <Xd>, <Xn>, <cond>`

is equivalent to

`CSNEG <Xd>, <Xn>, <Xn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields. |
| `<cond>` | Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted. |

### Operation

The description of CSNEG gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

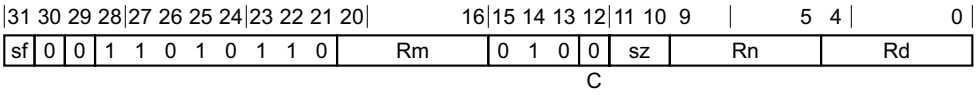
## C6.6.48 CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In ARMv8-A, this is an OPTIONAL instruction.

**Note**

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.




### CRC32B variant

Applies when sf = 0 && sz = 00.

```
CRC32B <Wd>, <Wn>, <Wm>
```

### CRC32H variant

Applies when sf = 0 && sz = 01.

```
CRC32H <Wd>, <Wn>, <Wm>
```

### CRC32W variant

Applies when sf = 0 && sz = 10.

```
CRC32W <Wd>, <Wn>, <Wm>
```

### CRC32X variant

Applies when sf = 1 && sz = 11.

```
CRC32X <Wd>, <Wn>, <Xm>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UnallocatedEncoding();
if sf == '0' && sz == '11' then UnallocatedEncoding();
integer size = 8 << UInt(sz);   // 2-bit size field -> 8, 16, 32, 64
boolean crc32c = (C == '1');
```

### Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field. |
| <Wm> | Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
if !HaveCRCExt() then
    UnallocatedEncoding();

bits(32)      acc     = X[n];   // accumulator
bits(size)    val     = X[m];   // input value
bits(32)      poly    = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;

bits(32+size) tempacc = BitReverse(acc) : Zeros(size);
bits(size+32) tempval = BitReverse(val) : Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

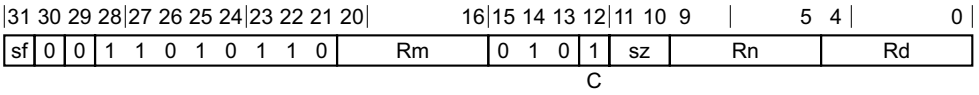
## C6.6.49 CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In ARMv8-A, this is an OPTIONAL instruction.

**Note**

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.




### *CRC32CB variant*

Applies when sf = 0 && sz = 00.

```
CRC32CB <Wd>, <Wn>, <Wm>
```

### *CRC32CH variant*

Applies when sf = 0 && sz = 01.

```
CRC32CH <Wd>, <Wn>, <Wm>
```

### *CRC32CW variant*

Applies when sf = 0 && sz = 10.

```
CRC32CW <Wd>, <Wn>, <Wm>
```

### *CRC32CX variant*

Applies when sf = 1 && sz = 11.

```
CRC32CX <Wd>, <Wn>, <Xm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UnallocatedEncoding();
if sf == '0' && sz == '11' then UnallocatedEncoding();
integer size = 8 << UInt(sz);   // 2-bit size field -> 8, 16, 32, 64
boolean crc32c = (C == '1');
```

### Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field. |
| <Wm> | Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
if !HaveCRCExt() then
    UnallocatedEncoding();

bits(32)      acc     = X[n];   // accumulator
bits(size)    val     = X[m];   // input value
bits(32)      poly    = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;

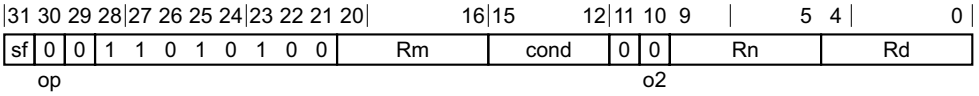
bits(32+size) tempacc = BitReverse(acc) : Zeros(size);
bits(size+32) tempval = BitReverse(val) : Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.50 CSEL

Conditional Select returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register.




### 32-bit variant

Applies when `sf = 0`.

```
CSEL <Wd>, <Wn>, <Wm>, <cond>
```

### 64-bit variant

Applies when `sf = 1`.

```
CSEL <Xd>, <Xn>, <Xm>, <cond>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');
```

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <cond> | Is one of the standard conditions, encoded in the "cond" field in the standard way. |

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(condition) then
    result = operand1;
else
    result = operand2;
    if else_inv then result = NOT(result);
    if else_inc then result = result + 1;

X[d] = result;
```

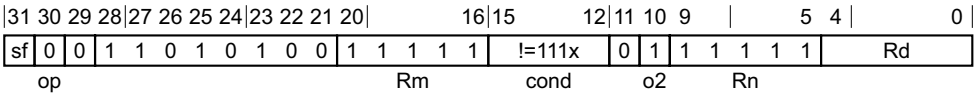

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.51 CSET

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This instruction is an alias of the CSINC instruction. This means that:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

```
CSET <Wd>, <cond>
```

is equivalent to

```
CSINC <Wd>, WZR, WZR, invert(<cond>)
```

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

```
CSET <Xd>, <cond>
```

is equivalent to

```
CSINC <Xd>, XZR, XZR, invert(<cond>)
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<cond>` | Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted. |

### Operation

The description of CSINC gives the operational pseudocode for this instruction.

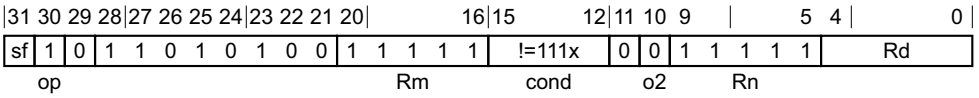

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.52 CSETM

Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This instruction is an alias of the CSINV instruction. This means that:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0`.

```
CSETM <Wd>, <cond>
```

is equivalent to

```
CSINV <Wd>, WZR, WZR, invert(<cond>)
```

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1`.

```
CSETM <Xd>, <cond>
```

is equivalent to

```
CSINV <Xd>, XZR, XZR, invert(<cond>)
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<cond>` | Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted. |

### Operation

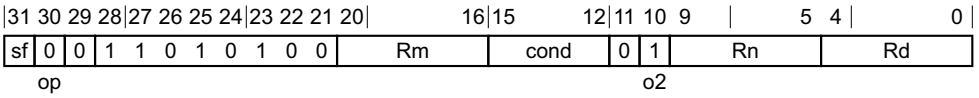
The description of CSINV gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.53 CSINC

Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases CINC and CSET. See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0`.

```
CSINC <Wd>, <Wn>, <Wm>, <cond>
```

### 64-bit variant

Applies when `sf = 1`.

```
CSINC <Xd>, <Xn>, <Xm>, <cond>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| CINC | `Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm` |
| CSET | `Rm == '11111' && cond != '111x' && Rn == '11111'` |

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<cond>` | Is one of the standard conditions, encoded in the "cond" field in the standard way. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(condition) then
    result = operand1;
else
    result = operand2;
    if else_inv then result = NOT(result);
    if else_inc then result = result + 1;

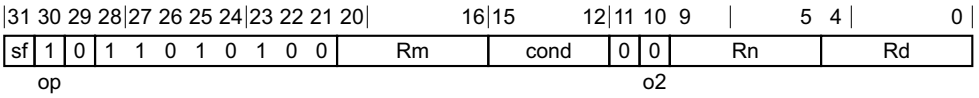
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.54 CSINV

Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases CINV and CSETM. See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0`.

```
CSINV <Wd>, <Wn>, <Wm>, <cond>
```

### 64-bit variant

Applies when `sf = 1`.

```
CSINV <Xd>, <Xn>, <Xm>, <cond>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| CINV | `Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm` |
| CSETM | `Rm == '11111' && cond != '111x' && Rn == '11111'` |

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<cond>` | Is one of the standard conditions, encoded in the "cond" field in the standard way. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(condition) then
    result = operand1;
else
    result = operand2;
    if else_inv then result = NOT(result);
    if else_inc then result = result + 1;

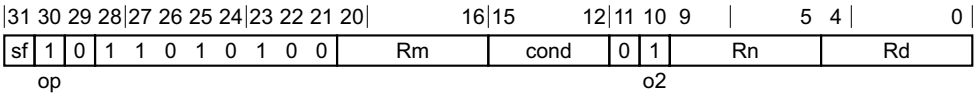
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.55 CSNEG

Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias CNEG. See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0`.

```
CSNEG <Wd>, <Wn>, <Wm>, <cond>
```

### 64-bit variant

Applies when `sf = 1`.

```
CSNEG <Xd>, <Xn>, <Xm>, <cond>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| CNEG | `cond != '111x' && Rn == Rm` |

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<cond>` | Is one of the standard conditions, encoded in the "cond" field in the standard way. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(condition) then
    result = operand1;
else
    result = operand2;
    if else_inv then result = NOT(result);
    if else_inc then result = result + 1;

X[d] = result;
```

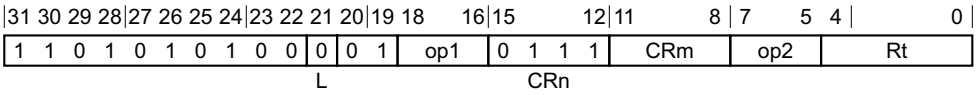

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.56 DC

Data Cache operation. For more information, see A64 system instructions for cache maintenance.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.




### *System variant*

```
DC <dc_op>, <Xt>
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1,'0111',CRm,op2) == Sys_DC`.

### Assembler symbols

`<dc_op>` Is a DC operation name, as listed for the DC system operation group, encoded in the "op1:CRm:op2" field. It can have the following values:

| | |
|---|---|
| `IVAC` | when `op1 = 000, CRm = 0110, op2 = 001` |
| `ISW` | when `op1 = 000, CRm = 0110, op2 = 010` |
| `CSW` | when `op1 = 000, CRm = 1010, op2 = 010` |
| `CISW` | when `op1 = 000, CRm = 1110, op2 = 010` |
| `ZVA` | when `op1 = 011, CRm = 0100, op2 = 001` |
| `CVAC` | when `op1 = 011, CRm = 1010, op2 = 001` |
| `CVAU` | when `op1 = 011, CRm = 1011, op2 = 001` |
| `CIVAC` | when `op1 = 011, CRm = 1110, op2 = 001` |

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<Xt>` Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

### Operation

The description of SYS gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.57 DCPS1

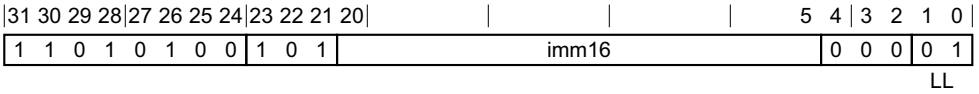
Debug Change PE State to EL1 allows the debugger to move the PE into EL1 from a lower Exception Level or to a specific mode at the current Exception Level.

If the PE is at EL1 or lower, then the PE enters EL1h.

If the PE is at an Exception Level higher than EL1, then the PE does not change Exception Level but selects use of the stack pointer for the current Exception Level by updating PSTATE.SP.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of this instructions, see DCPS.




### *System variant*

```
DCPS1 {#<imm>}
```

### *Decode for this encoding*

```
bits(2) target_level = LL;
if LL == '00' then UnallocatedEncoding();
if !Halted() then AArch64.UndefinedFault();
```

### Assembler symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

### Operation

```
DCPSInstruction(target_level);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.58 DCPS2

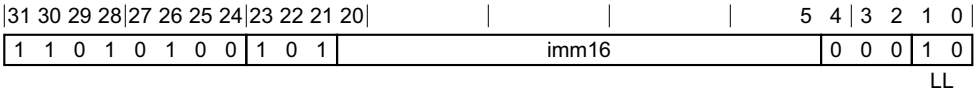
Debug Change PE State to EL2 allows the debugger to move the PE into EL2 from a lower Exception Level or to a specific mode at the current Exception Level.

If the PE is at EL2 or lower, then the PE enters EL2h.

If the PE is at an Exception Level higher than EL2, then the PE does not change Exception Level but selects use of the stack pointer for the current Exception Level by updating PSTATE.SP.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of this instructions, see DCPS.




### *System variant*

```
DCPS2 {#<imm>}
```

### *Decode for this encoding*

```
bits(2) target_level = LL;
if LL == '00' then UnallocatedEncoding();
if !Halted() then AArch64.UndefinedFault();
```

### Assembler symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

### Operation

```
DCPSInstruction(target_level);
```

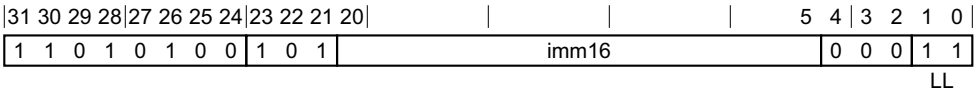

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.59 DCPS3

Debug Change PE State to EL3 allows the debugger to move the PE into EL3 from a lower Exception Level or to a specific mode at the current Exception Level. The PE enters EL3h.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of this instructions, see DCPS.




### *System variant*

```
DCPS3 {#<imm>}
```

### *Decode for this encoding*

```
bits(2) target_level = LL;
if LL == '00' then UnallocatedEncoding();
if !Halted() then AArch64.UndefinedFault();
```

### Assembler symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

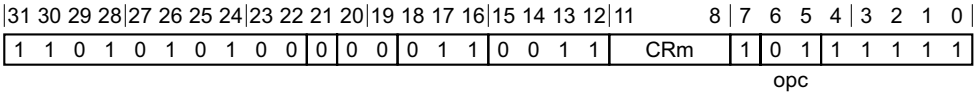
### Operation

```
DCPSInstruction(target_level);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.60 DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier (DMB)* on page B2-87.




### System variant

```
DMB <option>|#<imm>
```

### Decode for this encoding

```
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
    when '00' op = MemBarrierOp_DSB;
    when '01' op = MemBarrierOp_DMB;
    when '10' op = MemBarrierOp_ISB;
    otherwise UnallocatedEncoding();

case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
    otherwise
        types = MBReqTypes_All;
        domain = MBReqDomain_FullSystem;
```

### Assembler symbols

`<option>` Specifies the limitation on the barrier operation. Values are:

`SY` Full system is the required shareability domain, reads and writes are the required access types in both *Group A* on page B2-87 and *Group B* on page B2-87. This option is referred to as the full system DMB. Encoded as `CRm` = `0b1111`.

`ST` Full system is the required shareability domain, writes are the required access type in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as `CRm` = `0b1110`.

`LD` Full system is the required shareability domain, reads are the required access type in *Group A* on page B2-87, and reads and writes are the required access types in *Group B* on page B2-87. Encoded as `CRm` = `0b1101`.

`ISH` Inner Shareable is the required shareability domain, reads and writes are the required access types in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as `CRm` = `0b1011`.

`ISHST` Inner Shareable is the required shareability domain, writes are the required access type in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as `CRm` = `0b1010`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISHLD — Inner Shareable is the required shareability domain, reads are the required access type in *Group A* on page B2-87, and reads and writes are the required access types in *Group B* on page B2-87. Encoded as CRm = 0b1001.

NSH — Non-shareable is the required shareability domain, reads and writes are the required access types in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as CRm = 0b0111.

NSHST — Non-shareable is the required shareability domain, writes are the required access type in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as CRm = 0b0110.

NSHLD — Non-shareable is the required shareability domain, reads are the required access type in *Group A* on page B2-87, and reads and writes are the required access types in *Group B* on page B2-87. Encoded as CRm = 0b0101.

OSH — Outer Shareable is the required shareability domain, reads and writes are the required access types in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as CRm = 0b0011.

OSHST — Outer Shareable is the required shareability domain, writes are the required access type in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as CRm = 0b0010.

OSHLD — Outer Shareable is the required shareability domain, reads are the required access type in *Group A* on page B2-87, and reads and writes are the required access types in *Group B* on page B2-87. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system operation, but software must not rely on this behavior.

<imm> — Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

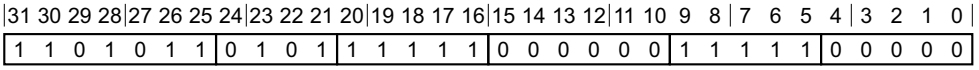
## Operation

```
case op of
    when MemBarrierOp_DSB
        DataSynchronizationBarrier(domain, types);
    when MemBarrierOp_DMB
        DataMemoryBarrier(domain, types);
    when MemBarrierOp_ISB
        InstructionSynchronizationBarrier();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.61 DRPS

Debug restore process state




### *System variant*

DRPS

### *Decode for this encoding*

```
if !Halted() || PSTATE.EL == EL0 then UnallocatedEncoding();
```

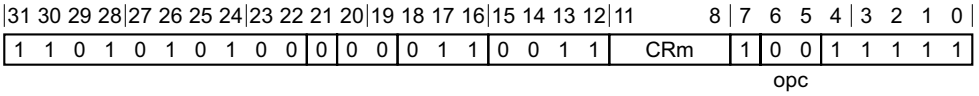
## Operation

```
DRPSInstruction();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.62 DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier (DSB)* on page B2-88.




### *System variant*

```
DSB <option>|#<imm>
```

### *Decode for this encoding*

```
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
    when '00' op = MemBarrierOp_DSB;
    when '01' op = MemBarrierOp_DMB;
    when '10' op = MemBarrierOp_ISB;
    otherwise UnallocatedEncoding();

case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
    otherwise
        types = MBReqTypes_All;
        domain = MBReqDomain_FullSystem;
```

### Assembler symbols

`<option>` Specifies the limitation on the barrier operation. Values are:

- `SY` Full system is the required shareability domain, reads and writes are the required access types in both *Group A* on page B2-87 and *Group B* on page B2-87. This option is referred to as the full system DMB. Encoded as `CRm` = `0b1111`.
- `ST` Full system is the required shareability domain, writes are the required access type in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as `CRm` = `0b1110`.
- `LD` Full system is the required shareability domain, reads are the required access type in *Group A* on page B2-87, and reads and writes are the required access types in *Group B* on page B2-87. Encoded as `CRm` = `0b1101`.
- `ISH` Inner Shareable is the required shareability domain, reads and writes are the required access types in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as `CRm` = `0b1011`.
- `ISHST` Inner Shareable is the required shareability domain, writes are the required access type in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as `CRm` = `0b1010`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISHLD Inner Shareable is the required shareability domain, reads are the required access type in *Group A* on page B2-87, and reads and writes are the required access types in *Group B* on page B2-87. Encoded as CRm = 0b1001.

NSH Non-shareable is the required shareability domain, reads and writes are the required access types in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as CRm = 0b0111.

NSHST Non-shareable is the required shareability domain, writes are the required access type in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as CRm = 0b0110.

NSHLD Non-shareable is the required shareability domain, reads are the required access type in *Group A* on page B2-87, and reads and writes are the required access types in *Group B* on page B2-87. Encoded as CRm = 0b0101.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as CRm = 0b0011.

OSHST Outer Shareable is the required shareability domain, writes are the required access type in both *Group A* on page B2-87 and *Group B* on page B2-87. Encoded as CRm = 0b0010.

OSHLD Outer Shareable is the required shareability domain, reads are the required access type in *Group A* on page B2-87, and reads and writes are the required access types in *Group B* on page B2-87. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system operation, but software must not rely on this behavior.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

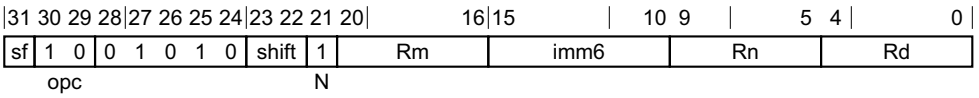
## Operation

```
case op of
    when MemBarrierOp_DSB
        DataSynchronizationBarrier(domain, types);
    when MemBarrierOp_DMB
        DataMemoryBarrier(domain, types);
    when MemBarrierOp_ISB
        InstructionSynchronizationBarrier();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.63 EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.




### 32-bit variant

Applies when `sf = 0`.

```
EON <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit variant

Applies when `sf = 1`.

```
EON <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

## Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

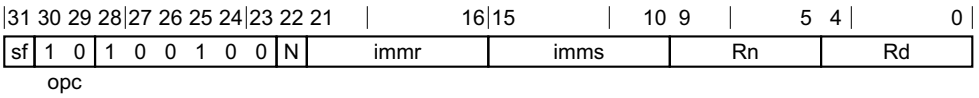
if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.64 EOR (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.




### *32-bit variant*

Applies when `sf = 0 && N = 0`.

```
EOR <Wd|WSP>, <Wn>, #<imm>
```

### *64-bit variant*

Applies when `sf = 1`.

```
EOR <Xd|SP>, <Xn>, #<imm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Assembler symbols

| | |
|---|---|
| `<Wd\|WSP>` | Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd\|SP>` | Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<imm>` | For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".<br>For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr". |

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

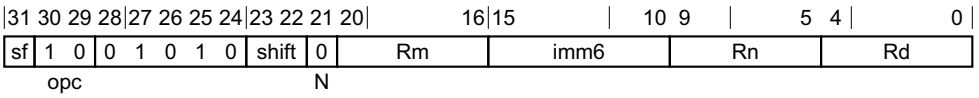
if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.65 EOR (shifted register)

Bitwise Exclusive OR (shifted register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.




### 32-bit variant

Applies when `sf = 0`.

```
EOR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit variant

Applies when `sf = 1`.

```
EOR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

## Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

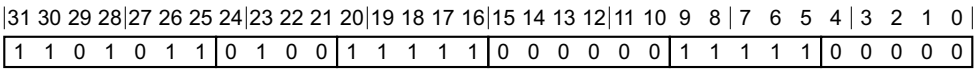

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.66 ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores PSTATE from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state* on page D1-1553.

ERET will cause an Undefined Instruction exception if executed in EL0.




### *System variant*

ERET

### *Decode for this encoding*

```
if PSTATE.EL == EL0 then UnallocatedEncoding();
```

### Operation

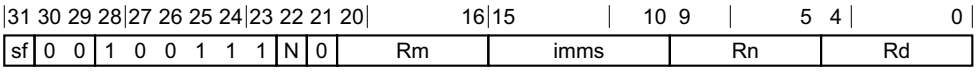
```
AArch64.ExceptionReturn(ELR[], SPSR[]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.67 EXTR

Extract register extracts a register from a pair of registers.

This instruction is used by the alias ROR (immediate). See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0 && N = 0 && imms = 0xxxxx`.

```
EXTR <Wd>, <Wn>, <Wm>, #<lsb>
```

### 64-bit variant

Applies when `sf = 1 && N = 1`.

```
EXTR <Xd>, <Xn>, <Xm>, #<lsb>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
integer lsb;

if N != sf then UnallocatedEncoding();
if sf == '0' && imms<5> == '1' then ReservedValue();
lsb = UInt(imms);
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| ROR (immediate) | `Rn == Rm` |

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <lsb> | For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field.<br>For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(2*datasize) concat = operand1:operand2;

result = concat<lsb+datasize-1:lsb>;

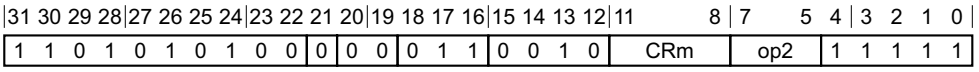
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.68 HINT

Hint instruction.

This instruction is used by the aliases NOP, SEVL, SEV, WFE, WFI, and YIELD. See the *Alias conditions* table for details of when each alias is preferred.




### *System variant*

```
HINT #<imm>
```

### *Decode for this encoding*

```
SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    otherwise  op = SystemHintOp_NOP;
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| NOP | `CRm == '0000' && op2 == '000'` |
| SEVL | `CRm == '0000' && op2 == '101'` |
| SEV | `CRm == '0000' && op2 == '100'` |
| WFE | `CRm == '0000' && op2 == '010'` |
| WFI | `CRm == '0000' && op2 == '011'` |
| YIELD | `CRm == '0000' && op2 == '001'` |

## Assembler symbols

<imm> Is a 7-bit unsigned immediate, in the range 0 to 127, encoded in "CRm:op2".

## Operation

```
case op of
    when SystemHintOp_YIELD
        Hint_Yield();

    when SystemHintOp_WFE
        if EventRegistered() then
            ClearEventRegister();
        else
            if PSTATE.EL == EL0 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                AArch64.CheckForWFxTrap(EL1, TRUE);
            if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
                AArch64.CheckForWFxTrap(EL2, TRUE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                AArch64.CheckForWFxTrap(EL3, TRUE);
            WaitForEvent();

    when SystemHintOp_WFI
        if !InterruptPending() then
            if PSTATE.EL == EL0 then
                AArch64.CheckForWFxTrap(EL1, FALSE);
            if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
                AArch64.CheckForWFxTrap(EL2, FALSE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                AArch64.CheckForWFxTrap(EL3, FALSE);
            WaitForInterrupt();

    when SystemHintOp_SEV
        SendEvent();

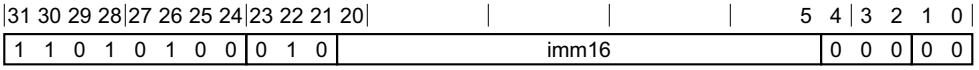
    when SystemHintOp_SEVL
        EventRegisterSet();

    otherwise // do nothing
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.69 HLT

Halt instruction causes a Halting Instruction debug event to occur.




### *System variant*

```
HLT #<imm>
```

### *Decode for this encoding*

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UndefinedFault();
```

### Assembler symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

### Operation

```
Halt(DebugHalt_HaltInstruction);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.70 HVC

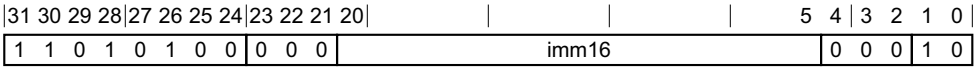
Hypervisor Call causes an exception to EL2. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction:

- Is UNDEFINED in ELO, and Secure EL1.
- When SCR_EL3.HCE is set to 0, generates an Undefined Instruction exception.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception, using the EC value 0x16, and the value of the immediate argument. This is reported in:

- ESR_ELx, if the exception is taken to an Exception level that is using AArch64.
- HSR, if the exception is taken to an Exception level that is using AArch32. See *Use of the HSR* on page G4-4105.




#### *System variant*

```
HVC #<imm>
```

#### *Decode for this encoding*

```
bits(16) imm = imm16;
```

### Assembler symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

### Operation

```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && IsSecure()) then
    UnallocatedEncoding();

hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);
if hvc_enable == '0' then
    AArch64.UndefinedFault();
else
    AArch64.CallHypervisor(imm);
```

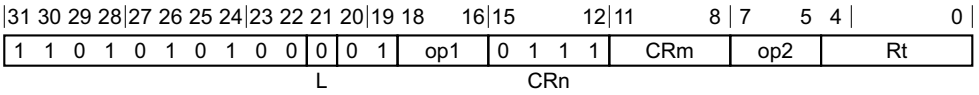

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.71 IC

Instruction Cache operation. For more information, see A64 system instructions for cache maintenance.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.




### *System variant*

```
IC <ic_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1,'0111',CRm,op2) == Sys_IC`.

### Assembler symbols

`<ic_op>` Is an IC operation name, as listed for the IC system operation pages, encoded in the "op1:CRm:op2" field. It can have the following values:

| | |
|---|---|
| `IALLUIS` | when `op1 = 000`, `CRm = 0001`, `op2 = 000` |
| `IALLU` | when `op1 = 000`, `CRm = 0101`, `op2 = 000` |
| `IVAU` | when `op1 = 011`, `CRm = 0101`, `op2 = 001` |

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<Xt>` Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

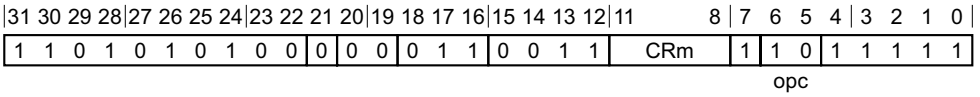
### Operation

The description of SYS gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.72 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE, so that all instructions following the `ISB` are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context changing operations executed before the `ISB` instruction are visible to the instructions fetched after the `ISB`. Context changing operations include changing the ASID, TLB maintenance instructions, and all changes to the System registers. In addition, any branches that appear in program order after the `ISB` instruction are written into the branch prediction logic with the context that is visible after the `ISB` instruction. This is needed to ensure correct execution of the instruction stream. For more information, see *Memory barriers* on page B2-86.




#### *System variant*

```
ISB {<option>|#<imm>}
```

#### *Decode for this encoding*

```
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
    when '00' op = MemBarrierOp_DSB;
    when '01' op = MemBarrierOp_DMB;
    when '10' op = MemBarrierOp_ISB;
    otherwise UnallocatedEncoding();

case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
    otherwise
        types = MBReqTypes_All;
        domain = MBReqDomain_FullSystem;
```

### Assembler symbols

`<option>` Specifies an optional limitation on the barrier operation. Values are:

`SY` Full system barrier operation, encoded as `CRm` = `0b1111`. Can be omitted.

All other encodings of `CRm` are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

`<imm>` Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

### Operation

```
case op of
    when MemBarrierOp_DSB
        DataSynchronizationBarrier(domain, types);
```

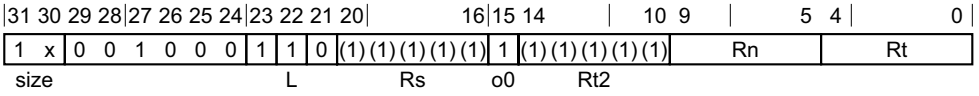

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when MemBarrierOp_DMB
    DataMemoryBarrier(domain, types);
when MemBarrierOp_ISB
    InstructionSynchronizationBarrier();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.73 LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




#### *32-bit variant*

Applies when `size = 10`.

```
LDAR <Wt>, [<Xn|SP>{,#0}]
```

#### *64-bit variant*

Applies when `size = 11`.

```
LDAR <Xt>, [<Xn|SP>{,#0}]
```

#### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

if o0 == '0' then UnallocatedEncoding();

AccType acctype = AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

### Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

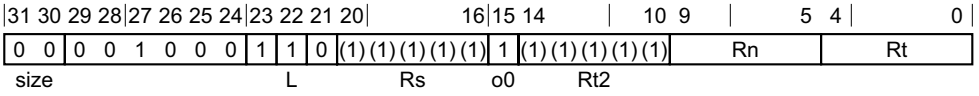
```
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.74 LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
LDARB <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

if o0 == '0' then UnallocatedEncoding();

AccType acctype = AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

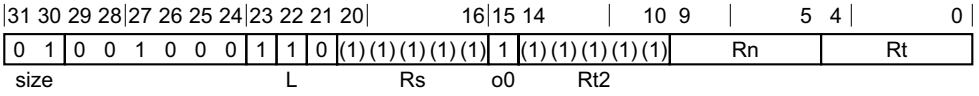
case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.75 LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
LDARH <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

if o0 == '0' then UnallocatedEncoding();

AccType acctype = AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

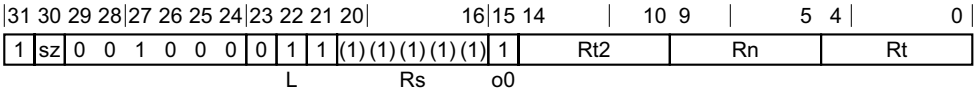
case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.76 LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-107. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### 32-bit variant

Applies when `sz = 0`.

```
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

### 64-bit variant

Applies when `sz = 1`.

```
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

### Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDAXP* on page K1-5505.

## Assembler symbols

| | |
|---|---|
| `<Wt1>` | Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Wt2>` | Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| `<Xt1>` | Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt2>` | Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

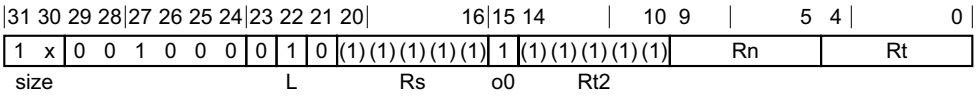
```
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t]  = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t]  = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t]  = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t]  = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.77 LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-107. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *32-bit variant*

Applies when `size = 10`.

```
LDAXR <Wt>, [<Xn|SP>{,#0}]
```

### *64-bit variant*

Applies when `size = 11`.

```
LDAXR <Xt>, [<Xn|SP>{,#0}]
```

### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when Constraint_NOP        EndOfInstruction();

    if memop == MemOp_STORE then
        if s == t || (pair && s == t2) then
            Constraint c = ConstrainUnpredictable();
            assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
            case c of
                when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
                when Constraint_NONE       rt_unknown = FALSE;   // store original value
                when Constraint_UNDEF      UnallocatedEncoding();
                when Constraint_NOP        EndOfInstruction();
        if s == n && n != 31 then
            Constraint c = ConstrainUnpredictable();
            assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
            case c of
                when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
                when Constraint_NONE       rn_unknown = FALSE;   // address is original base
                when Constraint_UNDEF      UnallocatedEncoding();
                when Constraint_NOP        EndOfInstruction();

    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    elsif rn_unknown then
        address = bits(64) UNKNOWN;
    else
        address = X[n];

    case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = X[t];
                bits(datasize DIV 2) el2 = X[t2];
                data = if BigEndian() then el1 : el2 else el2 : el1;
            else
                data = X[t];

            bit status = '1';
            // Check whether the Exclusive Monitors are set to include the
            // physical memory locations corresponding to virtual address
            // range [address, address+dbytes-1].
            if AArch64.ExclusiveMonitorsPass(address, dbytes) then
                // This atomic write will be rejected if it does not refer
                // to the same physical locations after address translation.
                Mem[address, dbytes, acctype] = data;
                status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

        when MemOp_LOAD
            // Tell the Exclusive Monitors to record a sequence of one or more atomic
            // memory reads from virtual address range [address, address+dbytes-1].
            // The Exclusive Monitor will only be set if all the reads are from the
            // same dbytes-aligned physical address, to allow for the possibility of
            // an atomicity break if the translation is changed between reads.
            AArch64.SetExclusiveMonitors(address, dbytes);

            if pair then
                if rt_unknown then
                    // ConstrainedUNPREDICTABLE case
                    X[t]  = bits(datasize) UNKNOWN;
                elsif elsize == 32 then
                    // 32-bit load exclusive pair (atomic)
                    data = Mem[address, dbytes, acctype];
                    if BigEndian() then
                        X[t]  = data<datasize-1:elsize>;
                        X[t2] = data<elsize-1:0>;
```

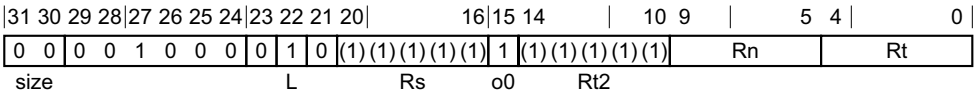

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            else
                X[t]  = data<elsize-1:0>;
                X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t]  = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.78 LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-107. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
LDAXRB <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
```

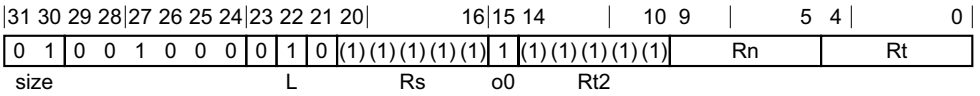

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t]  = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.79 LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-107. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
LDAXRH <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

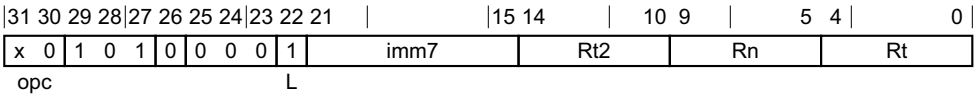
```
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t]  = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.80 LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see *Load/Store addressing modes* on page C1-126. For information about Non-temporal pair instructions, see *Load/Store Non-temporal Pair* on page C3-145.




### 32-bit variant

Applies when opc = 00.

```
LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit variant

Applies when opc = 10.

```
LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

### Decode for all variants of this encoding

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

## Assembler symbols

| | |
|---|---|
| <Wt1> | Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Wt2> | Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <Xt1> | Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xt2> | Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <imm> | For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.<br>For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        X[t]  = data1;
        X[t2] = data2;

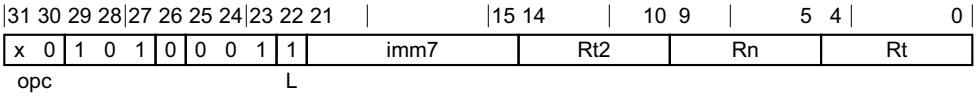
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.81 LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index




#### *32-bit variant*

Applies when `opc = 00`.

```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

#### *64-bit variant*

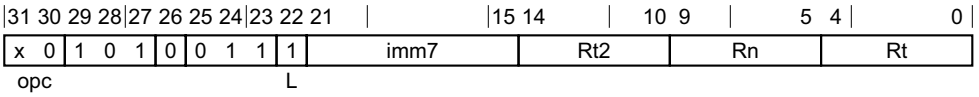
Applies when `opc = 10`.

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

#### *Decode for all variants of this encoding*

```
boolean wback  = TRUE;
boolean postindex = TRUE;
```

### Pre-index




#### *32-bit variant*

Applies when `opc = 00`.

```
LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

#### *64-bit variant*

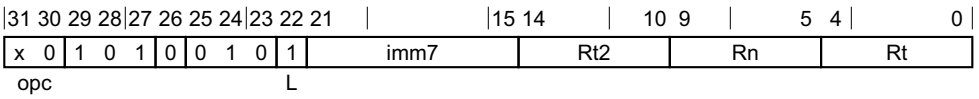
Applies when `opc = 10`.

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

#### *Decode for all variants of this encoding*

```
boolean wback  = TRUE;
boolean postindex = FALSE;
```

### Signed offset





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *32-bit variant*

Applies when opc = 00.

```
LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

#### *64-bit variant*

Applies when opc = 10.

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

#### *Decode for all variants of this encoding*

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDP* on page K1-5506.

### Assembler symbols

| | |
|---|---|
| <Wt1> | Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Wt2> | Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <Xt1> | Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xt2> | Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <imm> | For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.<br>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.<br>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.<br>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;   // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        if signed then
            X[t]  = SignExtend(data1, 64);
            X[t2] = SignExtend(data2, 64);
        else
            X[t]  = data1;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            X[t2] = data2;

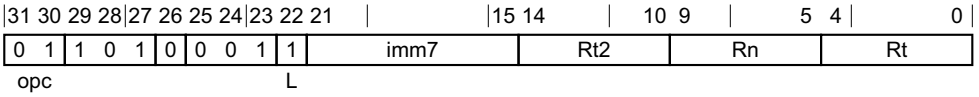
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.82 LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index



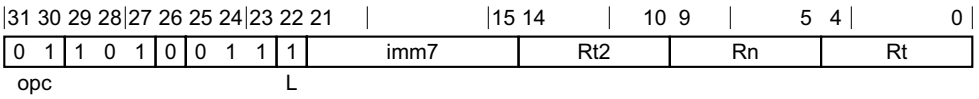

#### *Post-index variant*

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

#### *Decode for this encoding*

```
boolean wback  = TRUE;
boolean postindex = TRUE;
```

### Pre-index



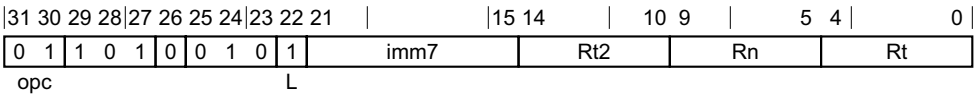

#### *Pre-index variant*

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

#### *Decode for this encoding*

```
boolean wback  = TRUE;
boolean postindex = FALSE;
```

### Signed offset




#### *Signed offset variant*

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

#### *Decode for this encoding*

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDPSW* on page K1-5507.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

### Operation for all encodings

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;   // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        if signed then
            X[t]  = SignExtend(data1, 64);
            X[t2] = SignExtend(data2, 64);
        else
            X[t]  = data1;
            X[t2] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.83 LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

### Post-index




#### *32-bit variant*

Applies when `size = 10`.

```
LDR <Wt>, [<Xn|SP>], #<simm>
```

#### *64-bit variant*

Applies when `size = 11`.

```
LDR <Xt>, [<Xn|SP>], #<simm>
```

#### *Decode for all variants of this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index




#### *32-bit variant*

Applies when `size = 10`.

```
LDR <Wt>, [<Xn|SP>, #<simm>]!
```

#### *64-bit variant*

Applies when `size = 11`.

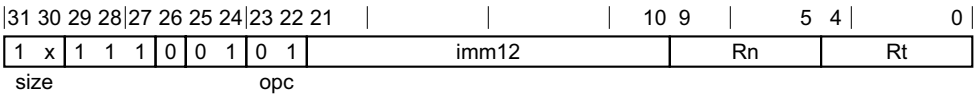
```
LDR <Xt>, [<Xn|SP>, #<simm>]!
```

#### *Decode for all variants of this encoding*

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Unsigned offset




#### *32-bit variant*

Applies when `size = 10`.

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

#### *64-bit variant*

Applies when `size = 11`.

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]
```

#### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDR (immediate)* on page K1-5507.

### Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<simm>` | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| `<pimm>` | For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.<br>For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
```

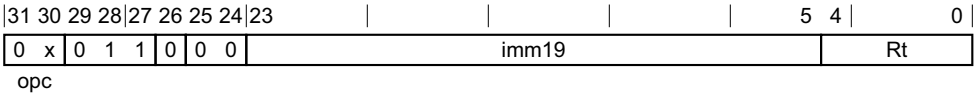

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.84 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *32-bit variant*

Applies when `opc = 00`.

```
LDR <Wt>, <label>
```

### *64-bit variant*

Applies when `opc = 01`.

```
LDR <Xt>, <label>
```

### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field. |
| `<label>` | Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4. |

## Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

case memop of
    when MemOp_LOAD
        data = Mem[address, size, AccType_NORMAL];
        if signed then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

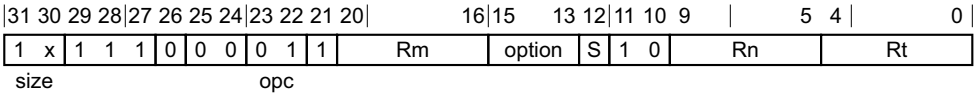
```
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.85 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *32-bit variant*

Applies when `size = 10`.

```
LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *64-bit variant*

Applies when `size = 11`.

```
LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#2` when `S = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#3` when `S = 1`

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

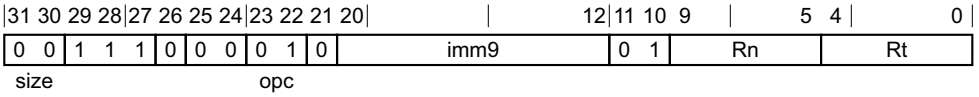
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.86 LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index




#### *Post-index variant*

```
LDRB <Wt>, [<Xn|SP>], #<simm>
```

#### *Decode for this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index



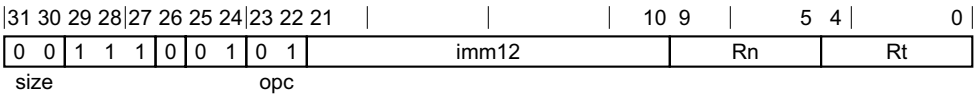

#### *Pre-index variant*

```
LDRB <Wt>, [<Xn|SP>, #<simm>]!
```

#### *Decode for this encoding*

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset




#### *Unsigned offset variant*

```
LDRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

#### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRB (immediate)* on page K1-5508.

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| <pimm> | Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

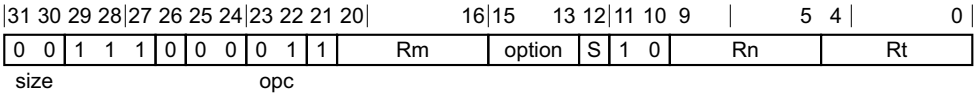
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.87 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Extended register variant*

Applies when `option != 011`.

```
LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### *Shifted register variant*

Applies when `option = 011`.

```
LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend specifier, encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

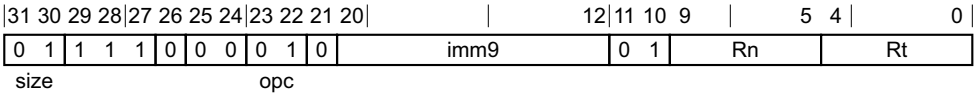
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.88 LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index




#### *Post-index variant*

```
LDRH <Wt>, [<Xn|SP>], #<simm>
```

#### *Decode for this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index



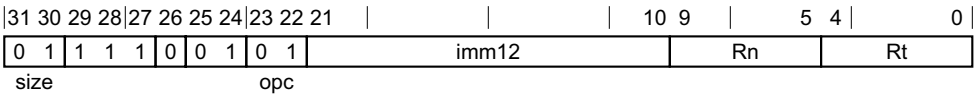

#### *Pre-index variant*

```
LDRH <Wt>, [<Xn|SP>, #<simm>]!
```

#### *Decode for this encoding*

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset




#### *Unsigned offset variant*

```
LDRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

#### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRH (immediate)* on page K1-5508.

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| <pimm> | Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

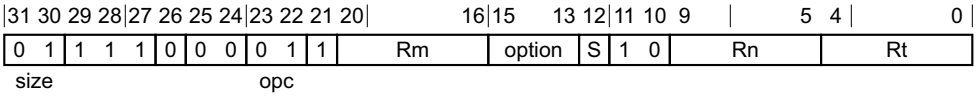
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.89 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### 32-bit variant

```
LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRH (register)* on page K1-5447.

## Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#1` when `S = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            else
                data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

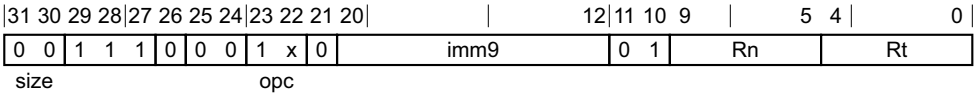
    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.90 LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate byte offset, loads a byte from memory, sign-extends it to either 32 or 64 bits, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index




#### *32-bit variant*

Applies when `opc = 11`.

```
LDRSB <Wt>, [<Xn|SP>], #<simm>
```

#### *64-bit variant*

Applies when `opc = 10`.

```
LDRSB <Xt>, [<Xn|SP>], #<simm>
```

#### *Decode for all variants of this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index




#### *32-bit variant*

Applies when `opc = 11`.

```
LDRSB <Wt>, [<Xn|SP>, #<simm>]!
```

#### *64-bit variant*

Applies when `opc = 10`.

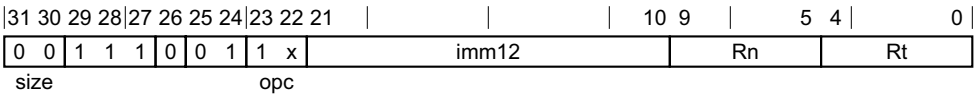
```
LDRSB <Xt>, [<Xn|SP>, #<simm>]!
```

#### *Decode for all variants of this encoding*

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Unsigned offset




#### *32-bit variant*

Applies when opc = 11.

```
LDRSB <Wt>, [<Xn|SP>{, #<pimm>}]
```

#### *64-bit variant*

Applies when opc = 10.

```
LDRSB <Xt>, [<Xn|SP>{, #<pimm>}]
```

#### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSB (immediate)* on page K1-5508.

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xt> | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| <pimm> | Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

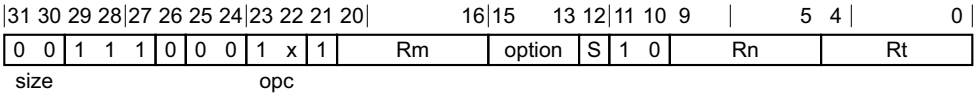
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.91 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *32-bit with extended register offset variant*

Applies when `opc = 11 && option != 011`.

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### *32-bit with shifted register offset variant*

Applies when `opc = 11 && option = 011`.

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### *64-bit with extended register offset variant*

Applies when `opc = 10 && option != 011`.

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### *64-bit with shifted register offset variant*

Applies when `opc = 10 && option = 011`.

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend specifier, encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `SXTW` when `option = 110`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

SXTX when option = 111

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

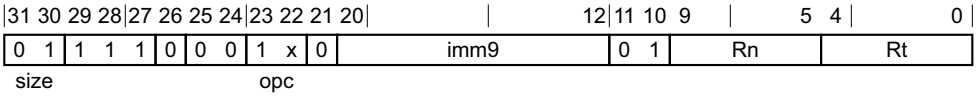
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.92 LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index




#### *32-bit variant*

Applies when `opc = 11`.

```
LDRSH <Wt>, [<Xn|SP>], #<simm>
```

#### *64-bit variant*

Applies when `opc = 10`.

```
LDRSH <Xt>, [<Xn|SP>], #<simm>
```

#### *Decode for all variants of this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index




#### *32-bit variant*

Applies when `opc = 11`.

```
LDRSH <Wt>, [<Xn|SP>, #<simm>]!
```

#### *64-bit variant*

Applies when `opc = 10`.

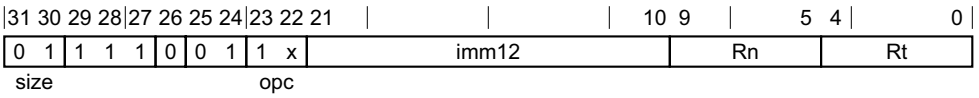
```
LDRSH <Xt>, [<Xn|SP>, #<simm>]!
```

#### *Decode for all variants of this encoding*

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Unsigned offset




#### *32-bit variant*

Applies when opc = 11.

```
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]
```

#### *64-bit variant*

Applies when opc = 10.

```
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]
```

#### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSH (immediate)* on page K1-5509.

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xt> | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| <pimm> | Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

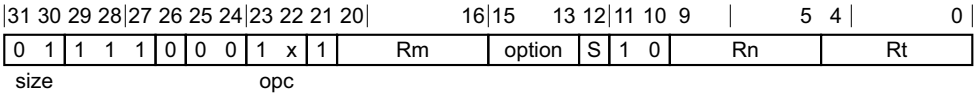
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.93 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### 32-bit variant

Applies when opc = 11.

```
LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-bit variant

Applies when opc = 10.

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- UXTW when option = 010
- LSL when option = 011
- SXTW when option = 110
- SXTX when option = 111

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- #0 when S = 0
- #1 when S = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            else
                data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

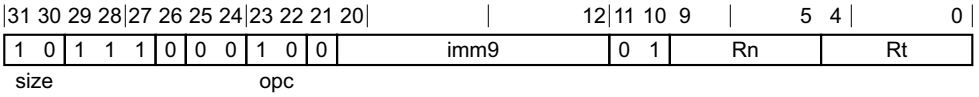
    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.94 LDRSW (immediate)

Load Register Signed Word (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index




#### *Post-index variant*

```
LDRSW <Xt>, [<Xn|SP>], #<simm>
```

#### *Decode for this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index



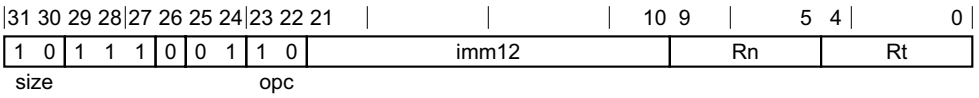

#### *Pre-index variant*

```
LDRSW <Xt>, [<Xn|SP>, #<simm>]!
```

#### *Decode for this encoding*

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset




#### *Unsigned offset variant*

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]
```

#### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSW (immediate)* on page K1-5509.

### Assembler symbols

| | |
|---|---|
| <Xt> | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| <pimm> | Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

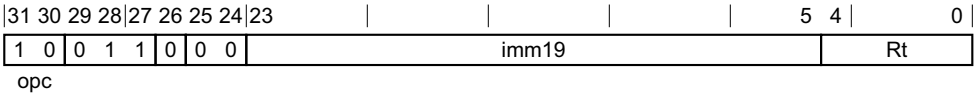
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.95 LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes on page C1-126*.




### *Literal variant*

```
LDRSW <Xt>, <label>
```

### *Decode for this encoding*

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

## Assembler symbols

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

## Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

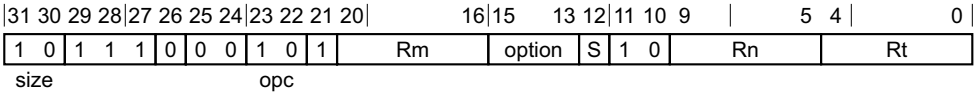
case memop of
    when MemOp_LOAD
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.96 LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *64-bit variant*

```
LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

### Assembler symbols

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#2` when `S = 1`

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.97 LDTR

Load Register (unprivileged) loads a word or doubleword from an address in memory, zero-extends it, and writes the result to a register. The address used for the load is calculated from a base register and an immediate offset. When executed at EL1, the memory access is restricted as if execution was at EL0. Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *32-bit variant*

Applies when `size = 10`.

```
LDTR <Wt>, [<Xn|SP>{, #<simm>}]
```

### *64-bit variant*

Applies when `size = 11`.

```
LDTR <Xt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<simm>` | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    SP[] = address;
else
    X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.98 LDTRB

Load Register Byte (unprivileged) loads a byte from an address in memory, zero-extends it, and writes the result to a register. The address used for the load is calculated from a base register and an immediate offset. When executed at EL1, the memory access is restricted as if execution was at EL0. Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Unscaled offset variant*

```
LDTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.99 LDTRH

Load Register Halfword (unprivileged) loads a halfword from an address in memory, zero-extends it, and writes the result to a register. The address used for the load is calculated from a base register and an immediate offset. When executed at EL1, the memory access is restricted as if execution was at EL0. Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Unscaled offset variant*

```
LDTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.100 LDTRSB

Load Register Signed Byte (unprivileged) loads a signed byte from an address in memory, zero-extends it, and writes the result to a register. The address used for the load is calculated from a base register and an immediate offset. When executed at EL1, the memory access is restricted as if execution was at EL0. Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *32-bit variant*

Applies when `opc = 11`.

```
LDTRSB <Wt>, [<Xn|SP>{, #<simm>}]
```

### *64-bit variant*

Applies when `opc = 10`.

```
LDTRSB <Xt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<simm>` | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.101 LDTRSH

Load Register Signed Halfword (unprivileged) loads a signed halfword from an address in memory, zero-extends it, and writes the result to a register. The address used for the load is calculated from a base register and an immediate offset. When executed at EL1, the memory access is restricted as if execution was at EL0. Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *32-bit variant*

Applies when `opc = 11`.

```
LDTRSH <Wt>, [<Xn|SP>{, #<simm>}]
```

### *64-bit variant*

Applies when `opc = 10`.

```
LDTRSH <Xt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<simm>` | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    SP[] = address;
else
    X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.102 LDTRSW

Load Register Signed Word (unprivileged) loads a signed word from an address in memory, zero-extends it, and writes the result to a register. The address used for the load is calculated from a base register and an immediate offset. When executed at EL1, the memory access is restricted as if execution was at EL0. Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Unscaled offset variant*

```
LDTRSW <Xt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler symbols

| | |
|---|---|
| <Xt> | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

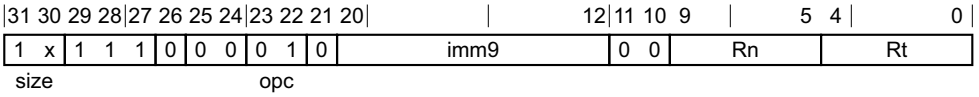
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.103 LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *32-bit variant*

Applies when `size = 10`.

```
LDUR <Wt>, [<Xn|SP>{, #<simm>}]
```

### *64-bit variant*

Applies when `size = 11`.

```
LDUR <Xt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<simm>` | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

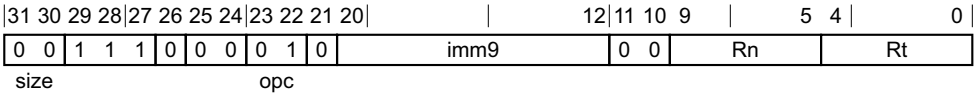
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.104 LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Unscaled offset variant*

```
LDURB <Wt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

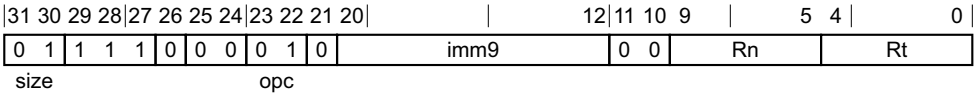
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.105 LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Unscaled offset variant*

```
LDURH <Wt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

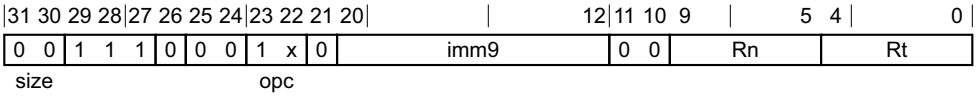
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.106 LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *32-bit variant*

Applies when `opc = 11`.

```
LDURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

### *64-bit variant*

Applies when `opc = 10`.

```
LDURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<simm>` | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

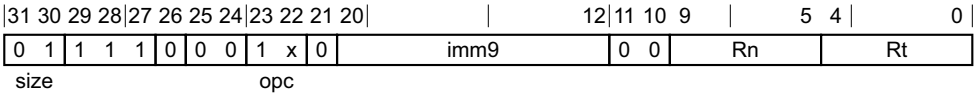
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.107 LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### 32-bit variant

Applies when opc = 11.

```
LDURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

### 64-bit variant

Applies when opc = 10.

```
LDURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

### Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xt> | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

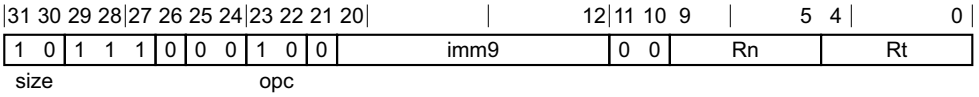
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.108 LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Unscaled offset variant*

```
LDURSW <Xt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler symbols

| | |
|---|---|
| <Xt> | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

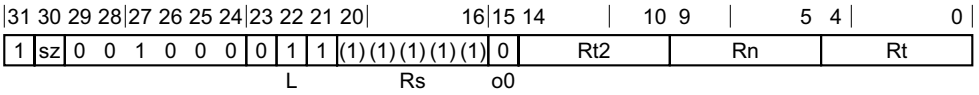
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.109 LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-107. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### 32-bit variant

Applies when `sz = 0`.

```
LDXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

### 64-bit variant

Applies when `sz = 1`.

```
LDXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

### Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDXP* on page K1-5509.

## Assembler symbols

| | |
|---|---|
| `<Wt1>` | Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Wt2>` | Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| `<Xt1>` | Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt2>` | Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

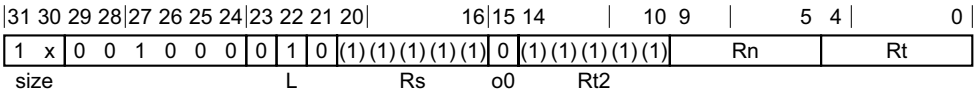
```
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t]  = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t]  = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t]  = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t]  = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.110 LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-107. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### 32-bit variant

Applies when size = 10.

```
LDXR <Wt>, [<Xn|SP>{,#0}]
```

### 64-bit variant

Applies when size = 11.

```
LDXR <Xt>, [<Xn|SP>{,#0}]
```

### Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xt> | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
```

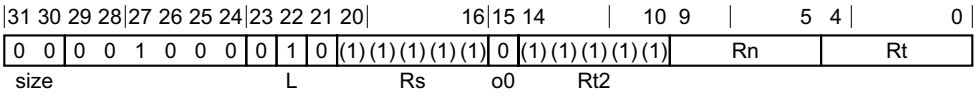

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            X[t]  = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t]  = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.111 LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-107. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




#### *No offset variant*

```
LDXRB <Wt>, [<Xn|SP>{,#0}]
```

#### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
```

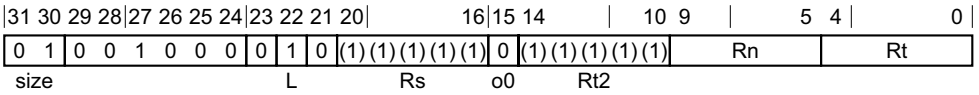

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        X[t2] = Mem[address + 8, 8, acctype];
else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.112 LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-107. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
LDXRH <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        X[t2] = Mem[address + 8, 8, acctype];
else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
```

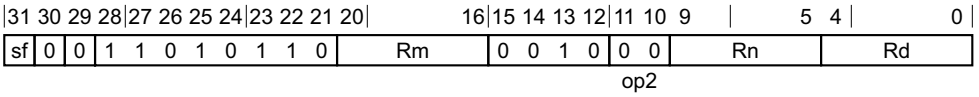

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.113 LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is an alias of the LSLV instruction. This means that:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0`.

`LSL <Wd>, <Wn>, <Wm>`

is equivalent to

`LSLV <Wd>, <Wn>, <Wm>`

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1`.

`LSL <Xd>, <Xn>, <Xm>`

is equivalent to

`LSLV <Xd>, <Xn>, <Xm>`

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field. |

### Operation

The description of LSLV gives the operational pseudocode for this instruction.

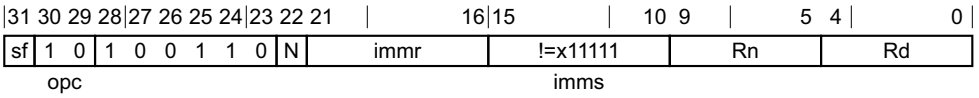

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.114 LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0 && N = 0 && imms != 011111`.

`LSL <Wd>, <Wn>, #<shift>`

is equivalent to

`UBFM <Wd>, <Wn>, #(-<shift> MOD 32), #(31-<shift>)`

and is the preferred disassembly when `imms + 1 == immr`.

### *64-bit variant*

Applies when `sf = 1 && N = 1 && imms != 111111`.

`LSL <Xd>, <Xn>, #<shift>`

is equivalent to

`UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)`

and is the preferred disassembly when `imms + 1 == immr`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<shift>` | For the 32-bit variant: is the shift amount, in the range 0 to 31.<br>For the 64-bit variant: is the shift amount, in the range 0 to 63. |

### Operation

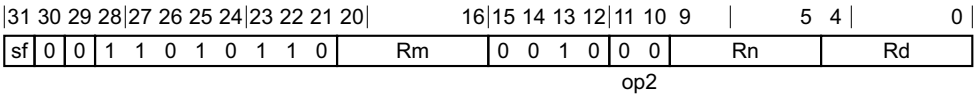
The description of UBFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.115 LSLV

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register). The alias is always the preferred disassembly.




### 32-bit variant

Applies when `sf = 0`.

```
LSLV <Wd>, <Wn>, <Wm>
```

### 64-bit variant

Applies when `sf = 1`.

```
LSLV <Xd>, <Xn>, <Xm>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field. |

## Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

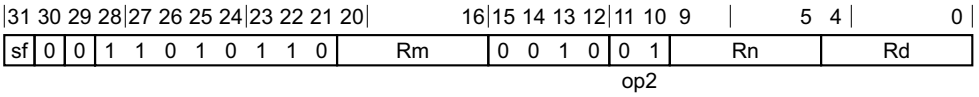

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.116 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is an alias of the LSRV instruction. This means that:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

`LSR <Wd>, <Wn>, <Wm>`

is equivalent to

`LSRV <Wd>, <Wn>, <Wm>`

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

`LSR <Xd>, <Xn>, <Xm>`

is equivalent to

`LSRV <Xd>, <Xn>, <Xm>`

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field. |

### Operation

The description of LSRV gives the operational pseudocode for this instruction.

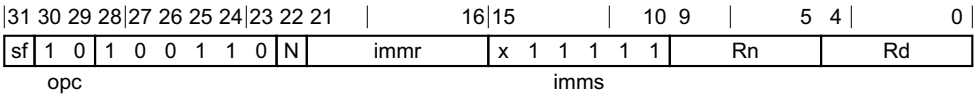

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.117 LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0 && N = 0 && imms = 011111`.

```
LSR <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1 && N = 1 && imms = 111111`.

```
LSR <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<shift>` | For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field.<br>For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field. |

### Operation

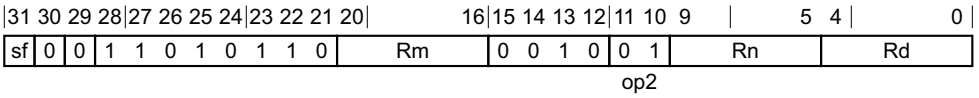
The description of UBFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.118 LSRV

Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register). The alias is always the preferred disassembly.




### 32-bit variant

Applies when `sf = 0`.

```
LSRV <Wd>, <Wn>, <Wm>
```

### 64-bit variant

Applies when `sf = 1`.

```
LSRV <Xd>, <Xn>, <Xm>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

## Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

## Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

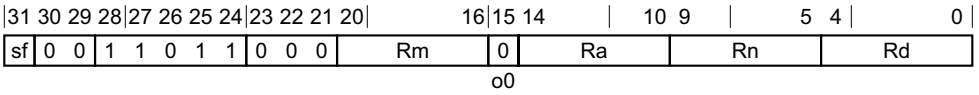
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.119 MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL. See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0`.

```
MADD <Wd>, <Wn>, <Wm>, <Wa>
```

### 64-bit variant

Applies when `sf = 1`.

```
MADD <Xd>, <Xn>, <Xm>, <Xa>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
integer datasize = destsize;
boolean sub_op = (o0 == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| MUL | `Ra == '11111'` |

## Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Wa>` Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

if sub_op then
    result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
else
    result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d] = result<destsize-1:0>;
```

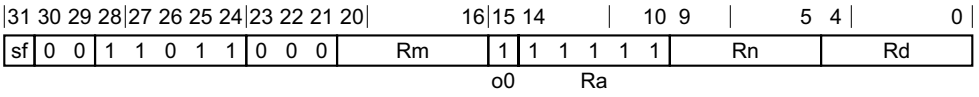

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.120 MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This instruction is an alias of the MSUB instruction. This means that:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0`.

```
MNEG <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MSUB <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1`.

```
MNEG <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MSUB <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

### Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

### Operation

The description of MSUB gives the operational pseudocode for this instruction.

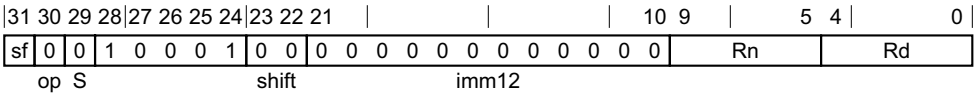

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.121 MOV (to/from SP)

Move between register and stack pointer : `Rd = Rn`

This instruction is an alias of the ADD (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

```
MOV <Wd|WSP>, <Wn|WSP>
```

is equivalent to

```
ADD <Wd|WSP>, <Wn|WSP>, #0
```

and is the preferred disassembly when `(Rd == '11111' || Rn == '11111')`.

### *64-bit variant*

Applies when `sf = 1`.

```
MOV <Xd|SP>, <Xn|SP>
```

is equivalent to

```
ADD <Xd|SP>, <Xn|SP>, #0
```

and is the preferred disassembly when `(Rd == '11111' || Rn == '11111')`.

### Assembler symbols

| | |
|---|---|
| `<Wd\|WSP>` | Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field. |
| `<Wn\|WSP>` | Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field. |
| `<Xd\|SP>` | Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field. |
| `<Xn\|SP>` | Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field. |

### Operation

The description of ADD (immediate) gives the operational pseudocode for this instruction.

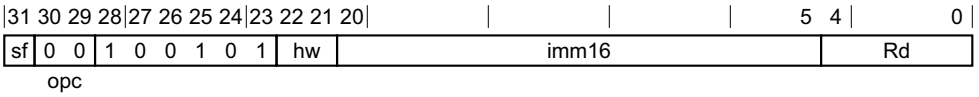

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.122 MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This instruction is an alias of the MOVN instruction. This means that:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0`.

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVN <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)`.

#### *64-bit variant*

Applies when `sf = 1`.

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVN <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<imm>` | For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xffff0000 and 0x0000ffff<br>For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw". |
| `<shift>` | For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.<br>For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16. |

### Operation

The description of MOVN gives the operational pseudocode for this instruction.

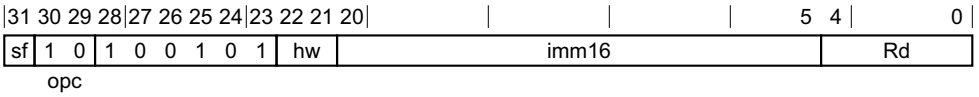

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.123 MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This instruction is an alias of the MOVZ instruction. This means that:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVZ <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

### *64-bit variant*

Applies when `sf = 1`.

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVZ <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<imm>` | For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw".<br>For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw". |
| `<shift>` | For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.<br>For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16. |

### Operation

The description of MOVZ gives the operational pseudocode for this instruction.

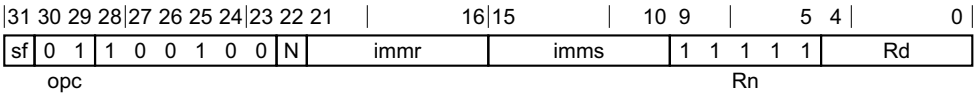

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.124 MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This instruction is an alias of the ORR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0 && N = 0`.

```
MOV <Wd|WSP>, #<imm>
```

is equivalent to

```
ORR <Wd|WSP>, WZR, #<imm>
```

and is the preferred disassembly when `! MoveWidePreferred(sf, N, imms, immr)`.

#### *64-bit variant*

Applies when `sf = 1`.

```
MOV <Xd|SP>, #<imm>
```

is equivalent to

```
ORR <Xd|SP>, XZR, #<imm>
```

and is the preferred disassembly when `! MoveWidePreferred(sf, N, imms, immr)`.

### Assembler symbols

`<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<imm>` For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

### Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

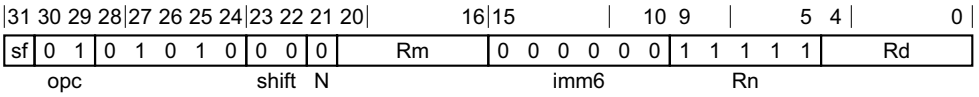

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.125 MOV (register)

Move (register) copies the value in a source register to the destination register.

This instruction is an alias of the ORR (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

```
MOV <Wd>, <Wm>
```

is equivalent to

```
ORR <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

```
MOV <Xd>, <Xm>
```

is equivalent to

```
ORR <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wm>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field. |

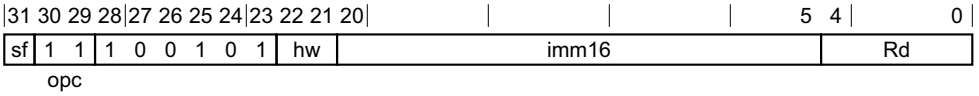
### Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.126 MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.




### 32-bit variant

Applies when `sf = 0`.

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

### 64-bit variant

Applies when `sf = 1`.

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
bits(16) imm = imm16;
integer pos;
MoveWideOp opcode;

case opc of
    when '00' opcode = MoveWideOp_N;
    when '10' opcode = MoveWideOp_Z;
    when '11' opcode = MoveWideOp_K;
    otherwise UnallocatedEncoding();

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <imm> | Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. |
| <shift> | For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.<br>For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16. |

## Operation

```
bits(datasize) result;

if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

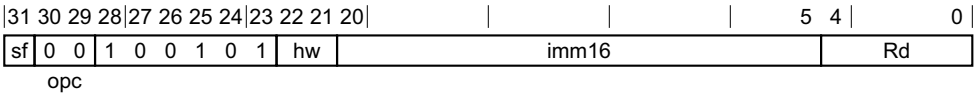
```
result<pos+15:pos> = imm;
if opcode == MoveWideOp_N then
    result = NOT(result);
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.127 MOVN

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (inverted wide immediate). See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when sf = 0.

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

### 64-bit variant

Applies when sf = 1.

```
MOVN <Xd>, #<imm>{, LSL #<shift>}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
bits(16) imm = imm16;
integer pos;
MoveWideOp opcode;

case opc of
    when '00' opcode = MoveWideOp_N;
    when '10' opcode = MoveWideOp_Z;
    when '11' opcode = MoveWideOp_K;
    otherwise UnallocatedEncoding();

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

## Alias conditions

| Alias | of variant | is preferred when |
|---|---|---|
| MOV (inverted wide immediate) | 64-bit | ! (IsZero(imm16) && hw != '00') |
| MOV (inverted wide immediate) | 32-bit | ! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16) |

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <imm> | Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. |
| <shift> | For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

## Operation

```
bits(datasize) result;

if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();

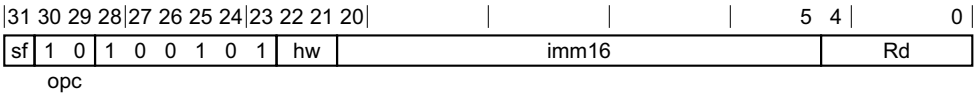
result<pos+15:pos> = imm;
if opcode == MoveWideOp_N then
    result = NOT(result);
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.128 MOVZ

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (wide immediate). See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when sf = 0.

```
MOVZ <Wd>, #<imm>{, LSL #<shift>}
```

### 64-bit variant

Applies when sf = 1.

```
MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
bits(16) imm = imm16;
integer pos;
MoveWideOp opcode;

case opc of
    when '00' opcode = MoveWideOp_N;
    when '10' opcode = MoveWideOp_Z;
    when '11' opcode = MoveWideOp_K;
    otherwise UnallocatedEncoding();

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| MOV (wide immediate) | ! (IsZero(imm16) && hw != '00') |

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <imm> | Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. |
| <shift> | For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.<br>For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(datasize) result;

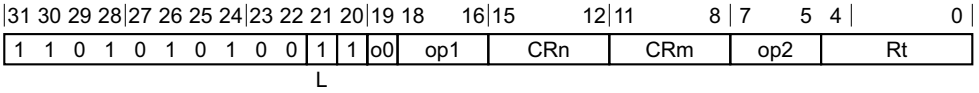
if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N then
    result = NOT(result);
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.129 MRS

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.




### *System variant*

```
MRS <Xt>, <systemreg>
```

### *Decode for this encoding*

```
CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
boolean read = (L == '1');
```

## Assembler symbols

| | |
|---|---|
| <Xt> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field. |
| <systemreg> | Is a System register name, encoded in the "o0:op1:CRn:CRm:op2". |

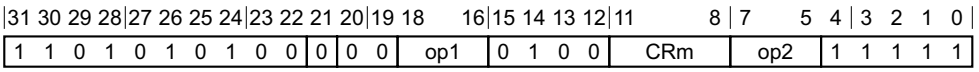
## Operation

```
if read then
    X[t] = System_Get(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    System_Put(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.130 MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE, namely D, A, I, F, and SP. For more information, see PSTATE.




### *System variant*

```
MSR <pstatefield>, #<imm>
```

### *Decode for this encoding*

```
CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');

bits(4) operand = CRm;
PSTATEField field;
case op1:op2 of
    when '000 101' field = PSTATEField_SP;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    otherwise      UnallocatedEncoding();

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if op1 == '011' && PSTATE.EL == EL0 && SCTLR_EL1.UMA == '0' then
    AArch64.SystemRegisterTrap(EL1, '00', op2, op1, '0100', '11111', CRm, '0');
```

### Assembler symbols

<pstatefield> Is a PSTATE field name, encoded in the "op1:op2" field. It can have the following values:

SPSel when op1 = 000, op2 = 101

DAIFSet when op1 = 011, op2 = 110

DAIFClr when op1 = 011, op2 = 111

The following encodings are reserved:

- op1 = 000, op2 = 0xx.
- op1 = 000, op2 = 100.
- op1 = 000, op2 = 11x.
- op1 = 001, op2 = xxx.
- op1 = 010, op2 = xxx.
- op1 = 011, op2 = 0xx.
- op1 = 011, op2 = 10x.
- op1 = 1xx, op2 = xxx.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

### Operation

```
case field of
    when PSTATEField_SP
        PSTATE.SP = operand<0>;
    when PSTATEField_DAIFSet
        PSTATE.D = PSTATE.D OR operand<3>;
```

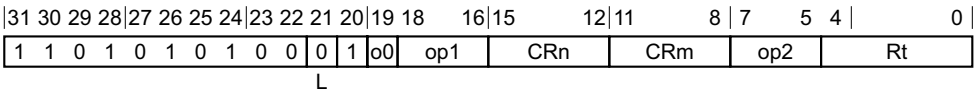

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        PSTATE.A = PSTATE.A OR operand<2>;
        PSTATE.I = PSTATE.I OR operand<1>;
        PSTATE.F = PSTATE.F OR operand<0>;
    when PSTATEField_DAIFClr
        PSTATE.D = PSTATE.D AND NOT(operand<3>);
        PSTATE.A = PSTATE.A AND NOT(operand<2>);
        PSTATE.I = PSTATE.I AND NOT(operand<1>);
        PSTATE.F = PSTATE.F AND NOT(operand<0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.131 MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.




### *System variant*

```
MSR <systemreg>, <Xt>
```

### *Decode for this encoding*

```
CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
boolean read = (L == '1');
```

### Assembler symbols

| | |
|---|---|
| <systemreg> | Is a System register name, encoded in the "o0:op1:CRn:CRm:op2". |
| <Xt> | Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field. |

### Operation

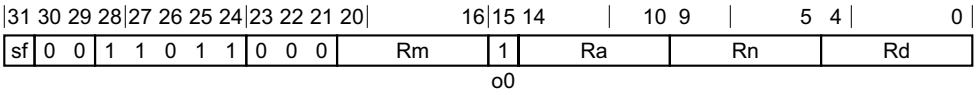
```
if read then
    X[t] = System_Get(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    System_Put(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.132 MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG. See the *Alias conditions* table for details of when each alias is preferred.




### *32-bit variant*

Applies when `sf = 0`.

```
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

### *64-bit variant*

Applies when `sf = 1`.

```
MSUB <Xd>, <Xn>, <Xm>, <Xa>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
integer datasize = destsize;
boolean sub_op = (o0 == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| MNEG | `Ra == '11111'` |

## Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Wa>` Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

if sub_op then
    result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
else
    result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d] = result<destsize-1:0>;
```

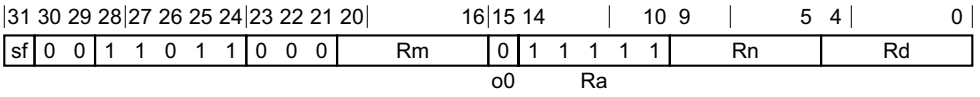

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.133 MUL

Multiply : `Rd = Rn * Rm`

This instruction is an alias of the MADD instruction. This means that:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

`MUL <Wd>, <Wn>, <Wm>`

is equivalent to

`MADD <Wd>, <Wn>, <Wm>, WZR`

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

`MUL <Xd>, <Xn>, <Xm>`

is equivalent to

`MADD <Xd>, <Xn>, <Xm>, XZR`

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |

### Operation

The description of MADD gives the operational pseudocode for this instruction.

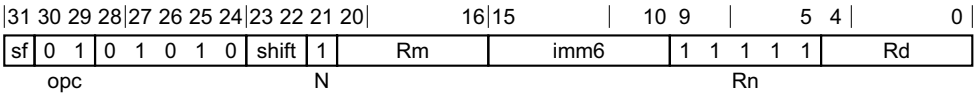

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.134 MVN

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This instruction is an alias of the ORN (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode for this instruction.




### 32-bit variant

Applies when `sf = 0`.

```
MVN <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

### 64-bit variant

Applies when `sf = 1`.

```
MVN <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

### Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`
- `ROR` when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

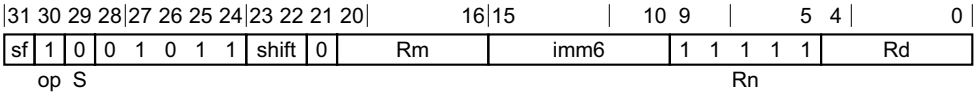

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.135 NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This instruction is an alias of the SUB (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0`.

```
NEG <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUB  <Wd>, WZR, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1`.

```
NEG <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUB  <Xd>, XZR, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

### Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

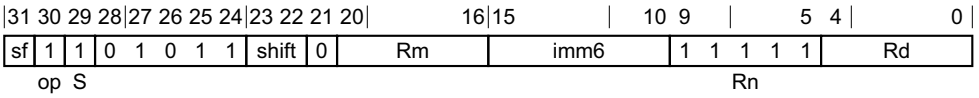

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.136 NEGS

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is an alias of the SUBS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

```
NEGS <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Wd>, WZR, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

```
NEGS <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Xd>, XZR, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

### Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

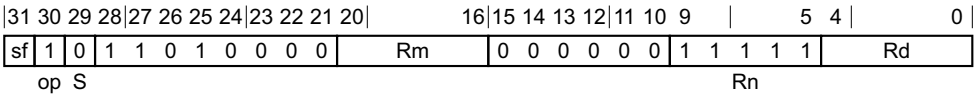

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.137 NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This instruction is an alias of the SBC instruction. This means that:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

`NGC <Wd>, <Wm>`

is equivalent to

`SBC <Wd>, WZR, <Wm>`

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

`NGC <Xd>, <Xm>`

is equivalent to

`SBC <Xd>, XZR, <Xm>`

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wm>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field. |

### Operation

The description of SBC gives the operational pseudocode for this instruction.

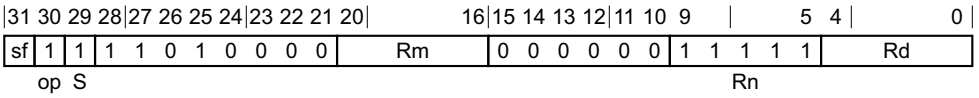

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.138 NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is an alias of the SBCS instruction. This means that:

- The encodings in this description are named to match the encodings of SBCS.
- The description of SBCS gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0`.

`NGCS <Wd>, <Wm>`

is equivalent to

`SBCS <Wd>, WZR, <Wm>`

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1`.

`NGCS <Xd>, <Xm>`

is equivalent to

`SBCS <Xd>, XZR, <Xm>`

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wm>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field. |

### Operation

The description of SBCS gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.139 NOP

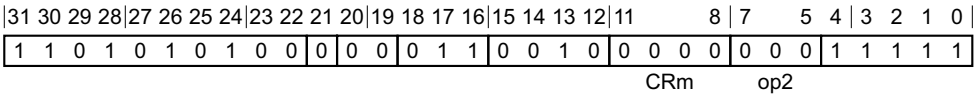
No Operation does nothing. This instruction can be used for instruction alignment purposes.

**Note**

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

This instruction is an alias of the HINT instruction. This means that:

- The encodings in this description are named to match the encodings of HINT.
- The description of HINT gives the operational pseudocode for this instruction.




### *System variant*

```
NOP
```

is equivalent to

```
HINT #0
```

and is always the preferred disassembly.

### Operation

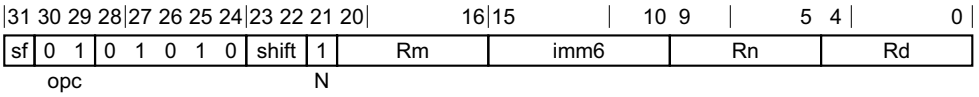
The description of HINT gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.140 ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN. See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0`.

```
ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit variant

Applies when `sf = 1`.

```
ORN <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| MVN | `Rn == '11111'` |

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

LSL when shift = 00

LSR when shift = 01

ASR when shift = 10

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

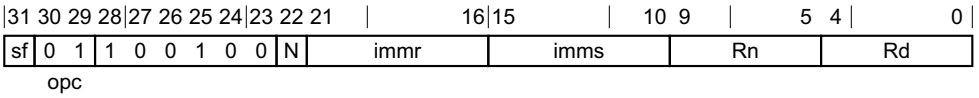
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.141 ORR (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias MOV (bitmask immediate). See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0 && N = 0`.

```
ORR <Wd|WSP>, <Wn>, #<imm>
```

### 64-bit variant

Applies when `sf = 1`.

```
ORR <Xd|SP>, <Xn>, #<imm>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| MOV (bitmask immediate) | `Rn == '11111' && ! MoveWidePreferred(sf, N, imms, immr)` |

## Assembler symbols

| | |
|---|---|
| `<Wd\|WSP>` | Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd\|SP>` | Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".

For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

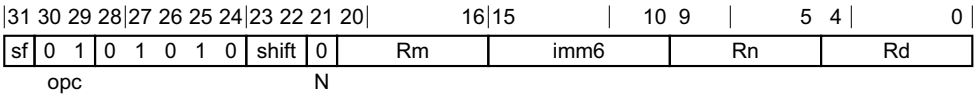
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.142 ORR (shifted register)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MOV (register). See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0`.

```
ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit variant

Applies when `sf = 1`.

```
ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| MOV (register) | `shift == '00' && imm6 == '000000' && Rn == '11111'` |

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

LSL when shift = 00

LSR when shift = 01

ASR when shift = 10

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

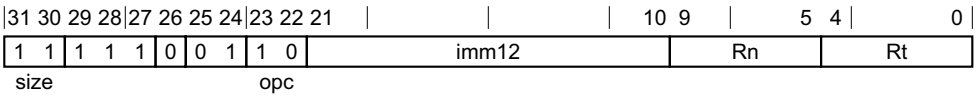

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.143 PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-151.

For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




#### *Unsigned offset variant*

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

#### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

### Assembler symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>. <type> is one of:

PLD Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1 Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2 Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3 Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory* on page C3-151. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            Mem[address, datasize DIV 8, acctype] = data;

        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```

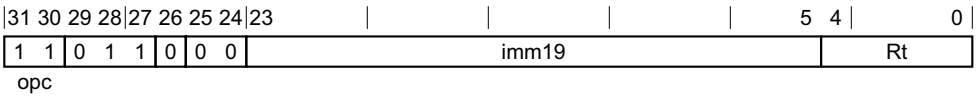

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.144 PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-151.

For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Literal variant*

```
PRFM (<prfop>|#<imm5>), <label>
```

### *Decode for this encoding*

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

### Assembler symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>. <type> is one of:

- PLD — Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
- PLI — Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
- PST — Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

- L1 — Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
- L2 — Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
- L3 — Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

- KEEP — Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
- STRM — Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information on these prefetch operations, see *Prefetch memory* on page C3-151. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

## Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

case memop of
    when MemOp_LOAD
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

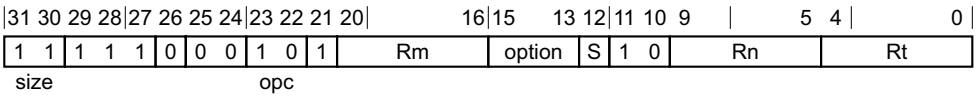

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.145 PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-151.

For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Integer variant*

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

### Assembler symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>. <type> is one of:

- PLD — Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
- PLI — Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
- PST — Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

- L1 — Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
- L2 — Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
- L3 — Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

- KEEP — Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
- STRM — Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory* on page C3-151. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

UXTW when option = 010

LSL when option = 011

SXTW when option = 110

SXTX when option = 111

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

#0 when S = 0

#3 when S = 1

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.146 PRFM (unscaled offset)

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-151.

For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




#### *Unscaled offset variant*

```
PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}]
```

#### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>. <type> is one of:

PLD Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1 Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2 Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3 Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory* on page C3-151. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            Mem[address, datasize DIV 8, acctype] = data;

        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);

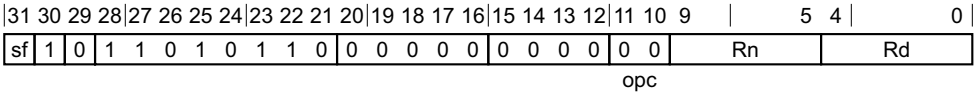
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.147 RBIT

Reverse Bits reverses the bit order in a register.




### *32-bit variant*

Applies when `sf = 0`.

```
RBIT <Wd>, <Wn>
```

### *64-bit variant*

Applies when `sf = 1`.

```
RBIT <Xd>, <Xn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

boolean is_rbit = opc == '00';
integer container_size;
case opc of
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |

## Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

if is_rbit then
    for i = 0 to datasize-1
        result<datasize-1-i> = operand<i>;
else
    integer containers = datasize DIV container_size;
    integer elements_per_container = container_size DIV 8;
    integer index = 0;
    integer rev_index;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

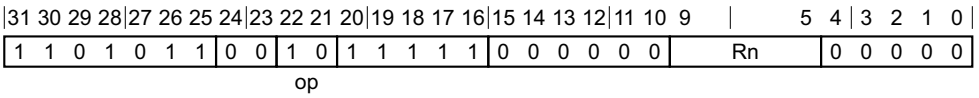
```
    for c = 0 to containers-1
        rev_index = index + ((elements_per_container - 1) * 8);
        for e = 0 to elements_per_container-1
            result<rev_index + 7:rev_index> = operand<index + 7:index>;
            index = index + 8;
            rev_index = rev_index - 8;

X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.148 RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.




### *Integer variant*

```
RET {<Xn>}
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
BranchType branch_type;

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();
```

## Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

## Operation

```
bits(64) target = X[n];

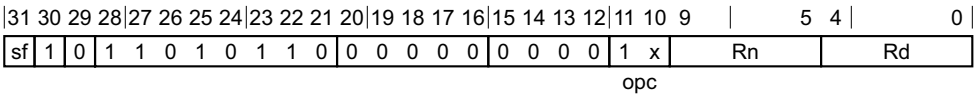
if branch_type == BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.149 REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64.




### 32-bit variant

Applies when `sf = 0 && opc = 10`.

```
REV <Wd>, <Wn>
```

### 64-bit variant

Applies when `sf = 1 && opc = 11`.

```
REV <Xd>, <Xn>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

boolean is_rbit = opc == '00';
integer container_size;
case opc of
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |

## Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

if is_rbit then
    for i = 0 to datasize-1
        result<datasize-1-i> = operand<i>;
else
    integer containers = datasize DIV container_size;
    integer elements_per_container = container_size DIV 8;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

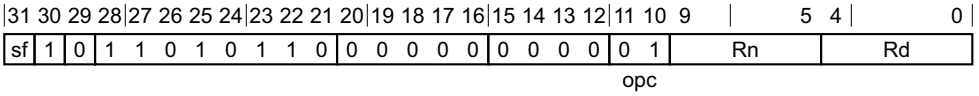
```
    integer index = 0;
    integer rev_index;
    for c = 0 to containers-1
        rev_index = index + ((elements_per_container - 1) * 8);
        for e = 0 to elements_per_container-1
            result<rev_index + 7:rev_index> = operand<index + 7:index>;
            index = index + 8;
            rev_index = rev_index - 8;

X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.150 REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.




### 32-bit variant

Applies when `sf = 0`.

```
REV16 <Wd>, <Wn>
```

### 64-bit variant

Applies when `sf = 1`.

```
REV16 <Xd>, <Xn>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

boolean is_rbit = opc == '00';
integer container_size;
case opc of
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |

## Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

if is_rbit then
    for i = 0 to datasize-1
        result<datasize-1-i> = operand<i>;
else
    integer containers = datasize DIV container_size;
    integer elements_per_container = container_size DIV 8;
    integer index = 0;
    integer rev_index;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

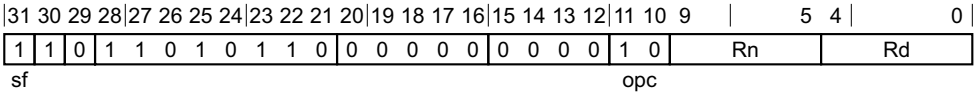
```
    for c = 0 to containers-1
        rev_index = index + ((elements_per_container - 1) * 8);
        for e = 0 to elements_per_container-1
            result<rev_index + 7:rev_index> = operand<index + 7:index>;
            index = index + 8;
            rev_index = rev_index - 8;

X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.151 REV32

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.




### 64-bit variant

```
REV32 <Xd>, <Xn>
```

### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

boolean is_rbit = opc == '00';
integer container_size;
case opc of
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

## Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

if is_rbit then
    for i = 0 to datasize-1
        result<datasize-1-i> = operand<i>;
else
    integer containers = datasize DIV container_size;
    integer elements_per_container = container_size DIV 8;
    integer index = 0;
    integer rev_index;
    for c = 0 to containers-1
        rev_index = index + ((elements_per_container - 1) * 8);
        for e = 0 to elements_per_container-1
            result<rev_index + 7:rev_index> = operand<index + 7:index>;
            index = index + 8;
            rev_index = rev_index - 8;

X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

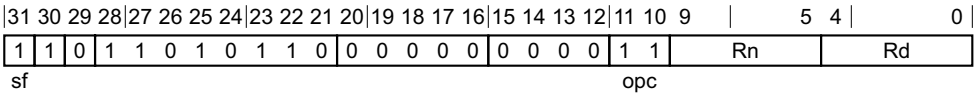
## C6.6.152 REV64

Reverse Bytes reverses the byte order in a 64-bit general-purpose register.

It is OPTIONAL whether an assembler supports this pseudo-instruction.

This instruction is an alias of the REV instruction. This means that:

- The encodings in this description are named to match the encodings of REV.
- The description of REV gives the operational pseudocode for this instruction.




### *64-bit variant*

```
REV64 <Xd>, <Xn>
```

is equivalent to

```
REV   <Xd>, <Xn>
```

and is never the preferred disassembly.

### Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

### Operation

The description of REV gives the operational pseudocode for this instruction.

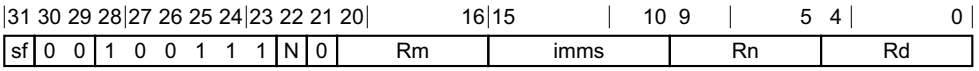

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.153 ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This instruction is an alias of the EXTR instruction. This means that:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0 && N = 0 && imms = 0xxxxx`.

`ROR <Wd>, <Ws>, #<shift>`

is equivalent to

`EXTR <Wd>, <Ws>, <Ws>, #<shift>`

and is the preferred disassembly when `Rn == Rm`.

#### *64-bit variant*

Applies when `sf = 1 && N = 1`.

`ROR <Xd>, <Xs>, #<shift>`

is equivalent to

`EXTR <Xd>, <Xs>, <Xs>, #<shift>`

and is the preferred disassembly when `Rn == Rm`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Ws>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xs>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields. |
| `<shift>` | For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field.<br>For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field. |

### Operation

The description of EXTR gives the operational pseudocode for this instruction.

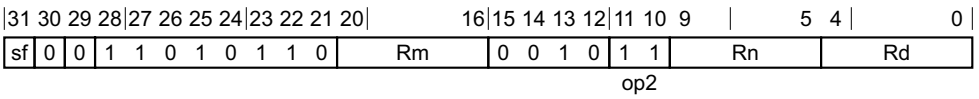

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.154 ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is an alias of the RORV instruction. This means that:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0`.

`ROR <Wd>, <Wn>, <Wm>`

is equivalent to

`RORV <Wd>, <Wn>, <Wm>`

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1`.

`ROR <Xd>, <Xn>, <Xm>`

is equivalent to

`RORV <Xd>, <Xn>, <Xm>`

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field. |

### Operation

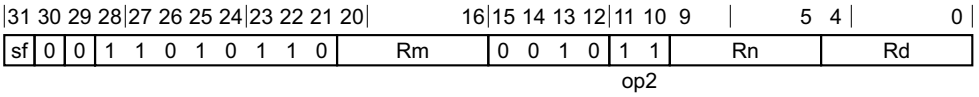
The description of RORV gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.155 RORV

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register). The alias is always the preferred disassembly.




### 32-bit variant

Applies when `sf = 0`.

```
RORV <Wd>, <Wn>, <Wm>
```

### 64-bit variant

Applies when `sf = 1`.

```
RORV <Xd>, <Xn>, <Xm>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field. |

## Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

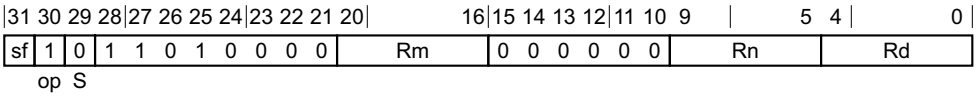
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.156 SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC. See the *Alias conditions* table for details of when each alias is preferred.




### *32-bit variant*

Applies when `sf = 0`.

```
SBC <Wd>, <Wn>, <Wm>
```

### *64-bit variant*

Applies when `sf = 1`.

```
SBC <Xd>, <Xn>, <Xm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| NGC | `Rn == '11111'` |

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if sub_op then
    operand2 = NOT(operand2);

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

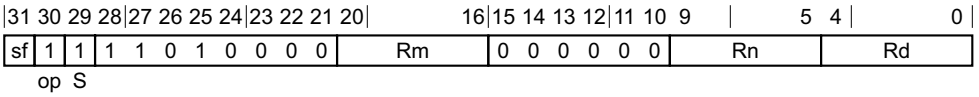
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.157 SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS. See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0`.

```
SBCS <Wd>, <Wn>, <Wm>
```

### 64-bit variant

Applies when `sf = 1`.

```
SBCS <Xd>, <Xn>, <Xm>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| NGCS | `Rn == '11111'` |

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if sub_op then
    operand2 = NOT(operand2);

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

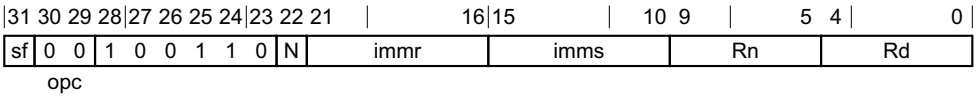

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.158 SBFIZ

Signed Bitfield Insert in Zero zeroes the destination register and copies any number of contiguous bits from a source register into any position in the destination register, sign-extending the most significant bit of the transferred value.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0 && N = 0`.

`SBFIZ <Wd>, <Wn>, #<lsb>, #<width>`

is equivalent to

`SBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)`

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

### *64-bit variant*

Applies when `sf = 1 && N = 1`.

`SBFIZ <Xd>, <Xn>, #<lsb>, #<width>`

is equivalent to

`SBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)`

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<lsb>` | For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.<br>For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63. |
| `<width>` | For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.<br>For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>. |

### Operation

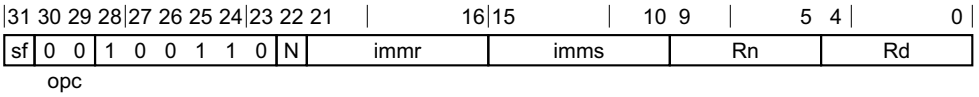
The description of SBFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.159 SBFM

Signed Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, shifting in copies of the sign bit in the upper bits and zeros in the lower bits.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW. See the *Alias conditions* on page C6-711 table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0 && N = 0`.

```
SBFM <Wd>, <Wn>, #<immr>, #<imms>
```

### 64-bit variant

Applies when `sf = 1 && N = 1`.

```
SBFM <Xd>, <Xn>, #<immr>, #<imms>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

case opc of
    when '00' inzero = TRUE;  extend = TRUE;    // SBFM
    when '01' inzero = FALSE; extend = FALSE;   // BFM
    when '10' inzero = TRUE;  extend = FALSE;   // UBFM
    when '11' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Alias conditions

| Alias | of variant | is preferred when |
|---|---|---|
| ASR (immediate) | 32-bit | `imms == '011111'` |
| ASR (immediate) | 64-bit | `imms == '111111'` |
| SBFIZ | - | `UInt(imms) < UInt(immr)` |
| SBFX | - | `BFXPreferred(sf, opc<1>, imms, immr)` |
| SXTB | - | `immr == '000000' && imms == '000111'` |
| SXTH | - | `immr == '000000' && imms == '001111'` |
| SXTW | - | `immr == '000000' && imms == '011111'` |

### Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

`<immr>` For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

`<imms>` For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

### Operation

```
bits(datasize) dst = if inzero then Zeros() else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

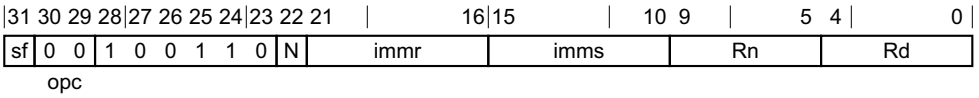

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.160 SBFX

Signed Bitfield Extract extracts any number of adjacent bits at any position from a register, sign-extends them to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0 && N = 0`.

`SBFX <Wd>, <Wn>, #<lsb>, #<width>`

is equivalent to

`SBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)`

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

### *64-bit variant*

Applies when `sf = 1 && N = 1`.

`SBFX <Xd>, <Xn>, #<lsb>, #<width>`

is equivalent to

`SBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)`

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<lsb>` | For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.<br>For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63. |
| `<width>` | For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.<br>For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>. |

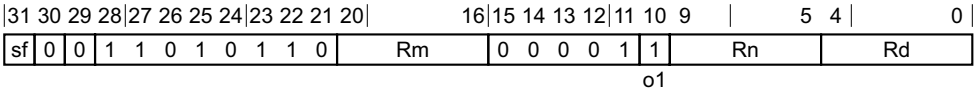
### Operation

The description of SBFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.161 SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.




### 32-bit variant

Applies when `sf = 0`.

```
SDIV <Wd>, <Wn>, <Wm>
```

### 64-bit variant

Applies when `sf = 1`.

```
SDIV <Xd>, <Xn>, <Xm>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean unsigned = (o1 == '0');
```

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, unsigned)) / Real(Int(operand2, unsigned)));

X[d] = result<datasize-1:0>;
```

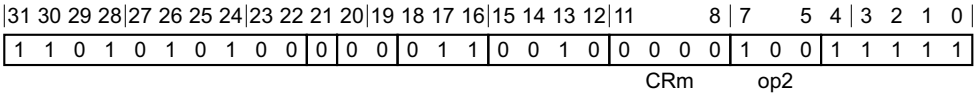

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.162 SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event* on page D1-1611.

This instruction is an alias of the HINT instruction. This means that:

- The encodings in this description are named to match the encodings of HINT.
- The description of HINT gives the operational pseudocode for this instruction.




#### *System variant*

```
SEV
```

is equivalent to

```
HINT #4
```

and is always the preferred disassembly.

### Operation

The description of HINT gives the operational pseudocode for this instruction.

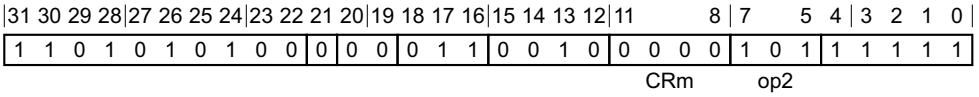

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.163 SEVL

Send Event Local is a hint instruction. It causes an event to be signaled locally without the requirement to affect other PEs in the multiprocessor system. It can prime a wait-loop which starts with a `WFE` instruction.

This instruction is an alias of the HINT instruction. This means that:

- The encodings in this description are named to match the encodings of HINT.
- The description of HINT gives the operational pseudocode for this instruction.




#### *System variant*

`SEVL`

is equivalent to

`HINT #5`

and is always the preferred disassembly.

### Operation

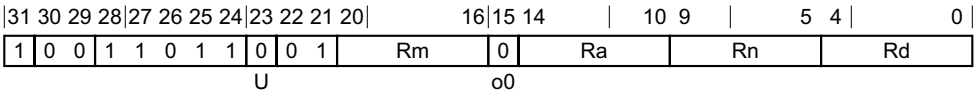
The description of HINT gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.164 SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL. See the *Alias conditions* table for details of when each alias is preferred.




### 64-bit variant

```
SMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| SMULL | `Ra == '11111'` |

## Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

if sub_op then
    result = Int(operand3, unsigned) - (Int(operand1, unsigned) * Int(operand2, unsigned));
else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    result = Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.165 SMC

Secure Monitor Call causes an exception to EL3.

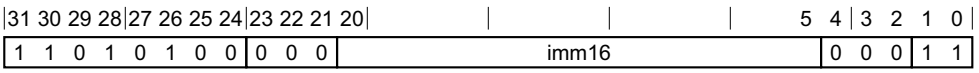
SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.

If the values of HCR_EL2.TSC and SCR_EL3.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, using the EC value 0x17, that is taken to EL3. When EL3 is using AArch32, this exception is taken to Monitor mode.

If the value of HCR_EL2.TSC is 1, execution of an SMC instruction in a Non-secure EL1 state generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD. When EL2 is using AArch32, this is a Hyp Trap exception that is taken to Hyp mode. For more information, see *Traps to EL2 of Non-secure EL1 execution of SMC instructions* on page D1-1590.

If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is:

- UNDEFINED in Non-secure state.
- CONSTRAINED UNPREDICTABLE if executed in Secure state at EL1 or higher.




### *System variant*

```
SMC #<imm>
```

### *Decode for this encoding*

```
bits(16) imm = imm16;
```

## Assembler symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

## Operation

```
if !HaveEL(EL3) || PSTATE.EL == EL0 then
    UnallocatedEncoding();

AArch64.CheckForSMCTrap(imm);

if SCR_EL3.SMD == '1' then
    // SMC disabled
    AArch64.UndefinedFault();
else
    AArch64.CallSecureMonitor(imm);
```

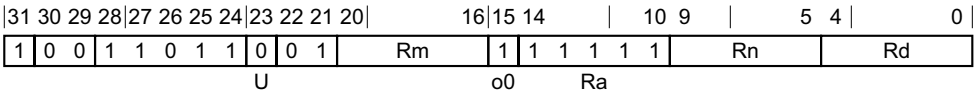

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.166 SMNEGL

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This instruction is an alias of the SMSUBL instruction. This means that:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode for this instruction.




### *64-bit variant*

```
SMNEGL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMSUBL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

### Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

### Operation

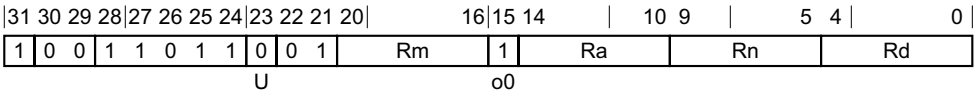
The description of SMSUBL gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.167 SMSUBL

Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL. See the *Alias conditions* table for details of when each alias is preferred.




### 64-bit variant

```
SMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| SMNEGL | `Ra == '11111'` |

## Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

if sub_op then
    result = Int(operand3, unsigned) - (Int(operand1, unsigned) * Int(operand2, unsigned));
else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

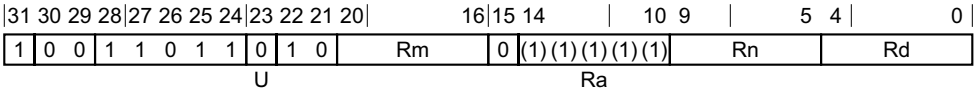
```
    result = Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.168 SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.




### 64-bit variant

```
SMULH <Xd>, <Xn>, <Xm>
```

### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);           // ignored by UMULH/SMULH
integer destsize = 64;
integer datasize = destsize;
boolean unsigned = (U == '1');
```

## Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

integer result;

result = Int(operand1, unsigned) * Int(operand2, unsigned);

X[d] = result<127:64>;
```

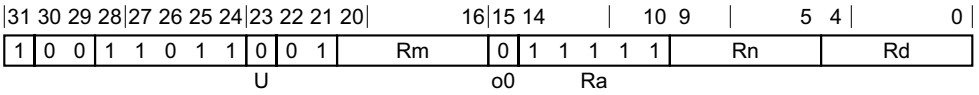

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.169 SMULL

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This instruction is an alias of the SMADDL instruction. This means that:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode for this instruction.




### *64-bit variant*

```
SMULL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMADDL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

### Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

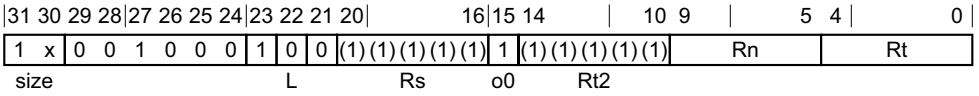
### Operation

The description of SMADDL gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.170 STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




#### *32-bit variant*

Applies when size = 10.

```
STLR <Wt>, [<Xn|SP>{,#0}]
```

#### *64-bit variant*

Applies when size = 11.

```
STLR <Xt>, [<Xn|SP>{,#0}]
```

#### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

if o0 == '0' then UnallocatedEncoding();

AccType acctype = AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xt> | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;
```

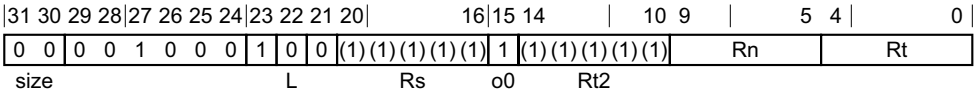

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when MemOp_LOAD
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.171 STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
STLRB <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

if o0 == '0' then UnallocatedEncoding();

AccType acctype = AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

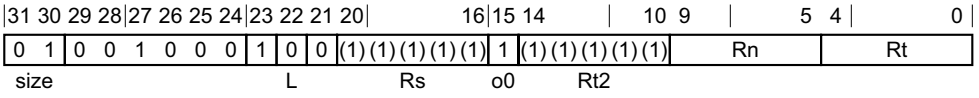
case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.172 STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
STLRH <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

if o0 == '0' then UnallocatedEncoding();

AccType acctype = AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

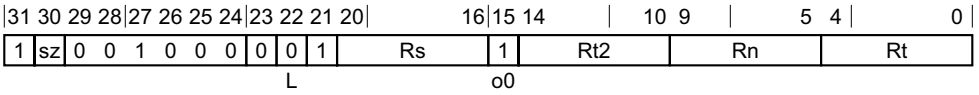
case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.173 STLXP

Store-Release Exclusive Pair Of Registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-107. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




#### 32-bit variant

Applies when `sz = 0`.

```
STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

#### 64-bit variant

Applies when `sz = 1`.

```
STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

#### Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STLXP* on page K1-5510.

### Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

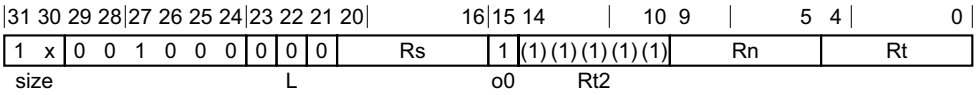
    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.174 STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-107. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### 32-bit variant

Applies when `size = 10`.

```
STLXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

### 64-bit variant

Applies when `size = 11`.

```
STLXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

### Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STLXR* on page K1-5510.

## Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

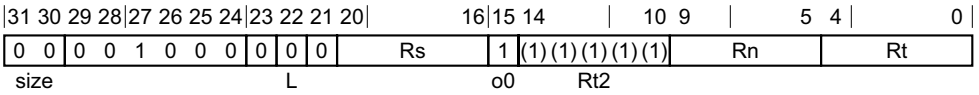
    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.175 STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-107. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
STLXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STLXRB* on page K1-5510.

### Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

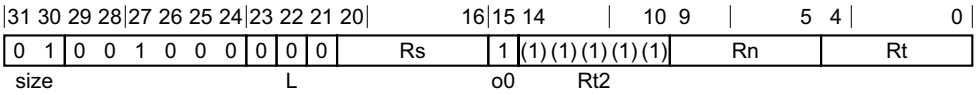
```
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t]  = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t]  = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t]  = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t]  = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.176 STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-107. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
STLXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STLXRH* on page K1-5511.

## Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

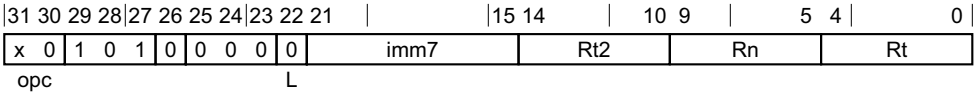
    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.177 STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes* on page C1-126. For information about Non-temporal pair instructions, see *Load/Store Non-temporal Pair* on page C3-145.




### *32-bit variant*

Applies when opc = 00.

```
STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

### *64-bit variant*

Applies when opc = 10.

```
STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

### *Decode for all variants of this encoding*

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

## Assembler symbols

| | |
|---|---|
| <Wt1> | Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Wt2> | Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <Xt1> | Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xt2> | Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <imm> | For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.<br>For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        X[t]  = data1;
        X[t2] = data2;

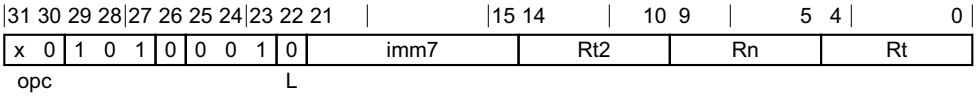
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.178 STP

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index




#### *32-bit variant*

Applies when `opc = 00`.

```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

#### *64-bit variant*

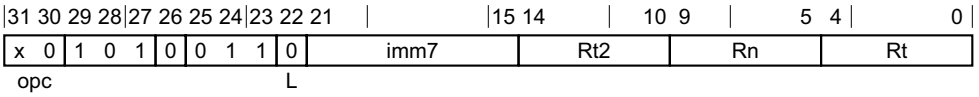
Applies when `opc = 10`.

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

#### *Decode for all variants of this encoding*

```
boolean wback  = TRUE;
boolean postindex = TRUE;
```

### Pre-index




#### *32-bit variant*

Applies when `opc = 00`.

```
STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

#### *64-bit variant*

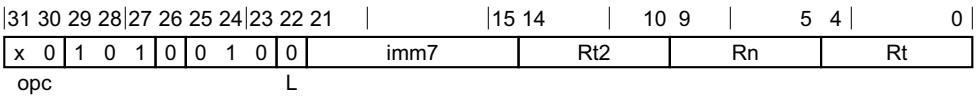
Applies when `opc = 10`.

```
STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

#### *Decode for all variants of this encoding*

```
boolean wback  = TRUE;
boolean postindex = FALSE;
```

### Signed offset





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *32-bit variant*

Applies when opc = 00.

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

#### *64-bit variant*

Applies when opc = 10.

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

#### *Decode for all variants of this encoding*

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STP* on page K1-5510.

### Assembler symbols

| | |
|---|---|
| <Wt1> | Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Wt2> | Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <Xt1> | Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xt2> | Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <imm> | For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.<br>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.<br>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.<br>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;   // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        if signed then
            X[t]  = SignExtend(data1, 64);
            X[t2] = SignExtend(data2, 64);
        else
            X[t]  = data1;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            X[t2] = data2;

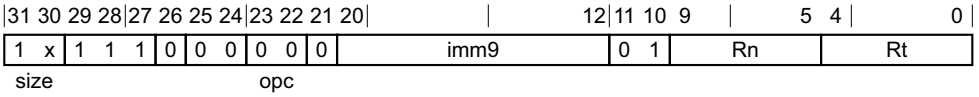
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.179 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index




#### *32-bit variant*

Applies when `size = 10`.

```
STR <Wt>, [<Xn|SP>], #<simm>
```

#### *64-bit variant*

Applies when `size = 11`.

```
STR <Xt>, [<Xn|SP>], #<simm>
```

#### *Decode for all variants of this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index




#### *32-bit variant*

Applies when `size = 10`.

```
STR <Wt>, [<Xn|SP>, #<simm>]!
```

#### *64-bit variant*

Applies when `size = 11`.

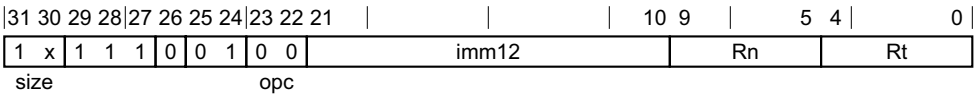
```
STR <Xt>, [<Xn|SP>, #<simm>]!
```

#### *Decode for all variants of this encoding*

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Unsigned offset




#### *32-bit variant*

Applies when size = 10.

```
STR <Wt>, [<Xn|SP>{, #<pimm>}]
```

#### *64-bit variant*

Applies when size = 11.

```
STR <Xt>, [<Xn|SP>{, #<pimm>}]
```

#### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xt> | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| <pimm> | For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.<br>For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

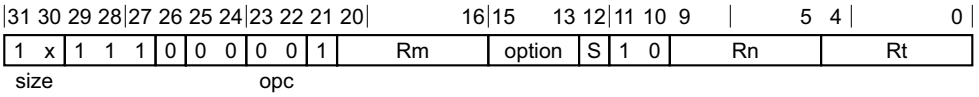
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.180 STR (register)

Store Register (register) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset. The offset can be optionally shifted and extended.




### *32-bit variant*

Applies when `size = 10`.

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *64-bit variant*

Applies when `size = 11`.

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

#0 when S = 0

#2 when S = 1

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

#0 when S = 0

#3 when S = 1

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

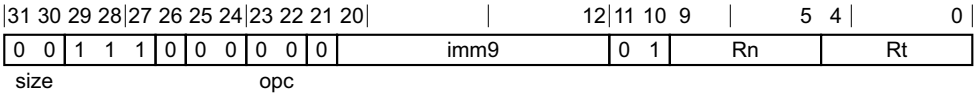
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.181 STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index




#### *Post-index variant*

```
STRB <Wt>, [<Xn|SP>], #<simm>
```

#### *Decode for this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index



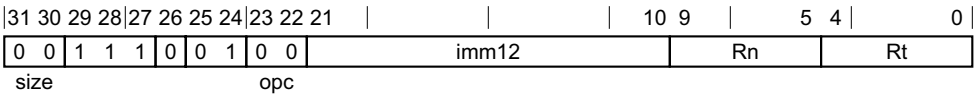

#### *Pre-index variant*

```
STRB <Wt>, [<Xn|SP>, #<simm>]!
```

#### *Decode for this encoding*

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset




#### *Unsigned offset variant*

```
STRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

#### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRB (immediate)* on page K1-5511.

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| <pimm> | Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

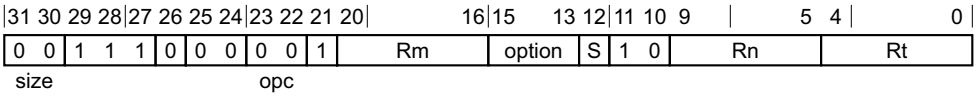
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.182 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an immediate offset, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset. The offset can be optionally shifted and extended.




### *Extended register variant*

Applies when `option != 011`.

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### *Shifted register variant*

Applies when `option = 011`.

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend specifier, encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

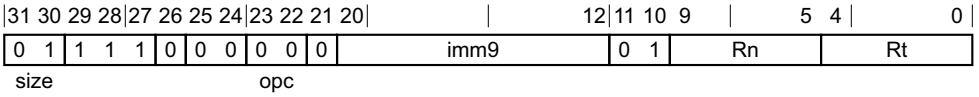
    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.183 STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

### Post-index




#### *Post-index variant*

```
STRH <Wt>, [<Xn|SP>], #<simm>
```

#### *Decode for this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index



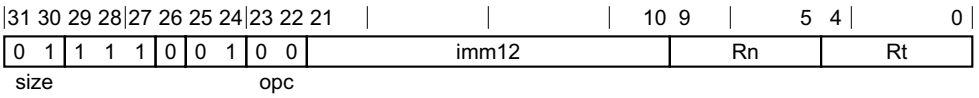

#### *Pre-index variant*

```
STRH <Wt>, [<Xn|SP>, #<simm>]!
```

#### *Decode for this encoding*

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset




#### *Unsigned offset variant*

```
STRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

#### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRH (immediate)* on page K1-5512.

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| <pimm> | Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

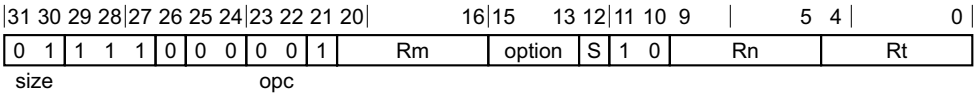
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.184 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an immediate offset, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.




### *32-bit variant*

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#1` when `S = 1`

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.185 STTR

Store Register (unprivileged) stores a word or doubleword from an address in memory and zero-extends it. The address used for the store is calculated from a base register and an immediate offset. When executed at EL1, the memory access is restricted as if execution was at EL0. Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *32-bit variant*

Applies when `size = 10`.

```
STTR <Wt>, [<Xn|SP>{, #<simm>}]
```

### *64-bit variant*

Applies when `size = 11`.

```
STTR <Xt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<simm>` | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.186 STTRB

Store Register Byte (unprivileged) stores a byte from an address in memory and zero-extends it. The address used for the store is calculated from a base register and an immediate offset. When executed at EL1, the memory access is restricted as if execution was at EL0. Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *Unscaled offset variant*

```
STTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.187 STTRH

Store Register Halfword (unprivileged) stores a halfword from an address in memory and zero-extends it. The address used for the store is calculated from a base register and an immediate offset. When executed at EL1, the memory access is restricted as if execution was at EL0. Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Unscaled offset variant*

```
STTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

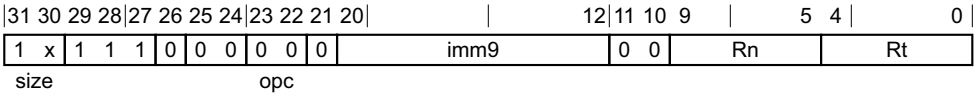
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.188 STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *32-bit variant*

Applies when `size = 10`.

```
STUR <Wt>, [<Xn|SP>{, #<simm>}]
```

### *64-bit variant*

Applies when `size = 11`.

```
STUR <Xt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| `<Wt>` | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xt>` | Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<simm>` | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

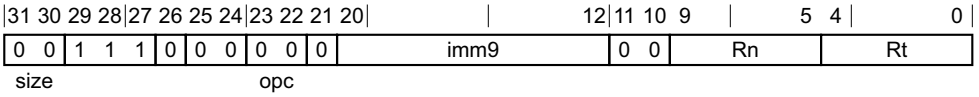
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.189 STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### Unscaled offset variant

```
STURB <Wt>, [<Xn|SP>{, #<simm>}]
```

### Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

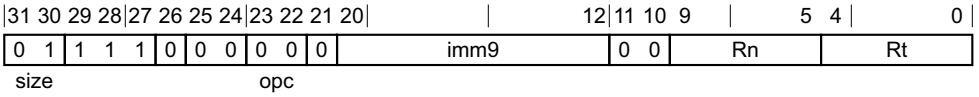
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.190 STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes* on page C1-126.




### *Unscaled offset variant*

```
STURH <Wt>, [<Xn|SP>{, #<simm>}]
```

### *Decode for this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler symbols

| | |
|---|---|
| <Wt> | Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

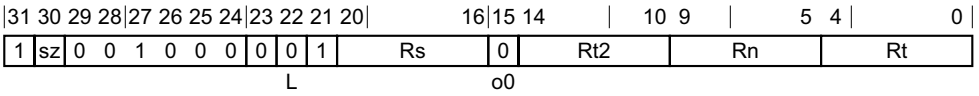
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.191 STXP

Store Exclusive Pair Of Registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-107. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### 32-bit variant

Applies when `sz = 0`.

```
STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

### 64-bit variant

Applies when `sz = 1`.

```
STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

### Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STXP* on page K1-5512.

## Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Wt1>` Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when MemOp_STORE
    if rt_unknown then
        data = bits(datasize) UNKNOWN;
    elsif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian() then el1 : el2 else el2 : el1;
    else
        data = X[t];

    bit status = '1';
    // Check whether the Exclusive Monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);

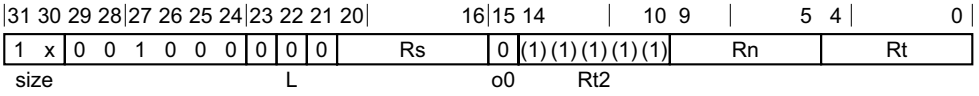
when MemOp_LOAD
    // Tell the Exclusive Monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusive Monitor will only be set if all the reads are from the
    // same dbytes-aligned physical address, to allow for the possibility of
    // an atomicity break if the translation is changed between reads.
    AArch64.SetExclusiveMonitors(address, dbytes);

    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case
            X[t]  = bits(datasize) UNKNOWN;
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian() then
                X[t]  = data<datasize-1:elsize>;
                X[t2] = data<elsize-1:0>;
            else
                X[t]  = data<elsize-1:0>;
                X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t]  = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.192 STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-107. For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### 32-bit variant

Applies when `size = 10`.

```
STXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

### 64-bit variant

Applies when `size = 11`.

```
STXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

### Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STXR* on page K1-5512.

## Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

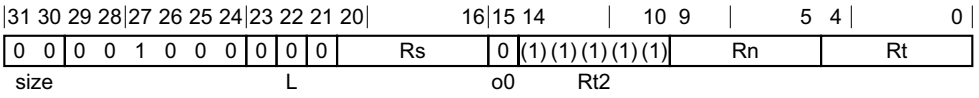
        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.193 STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-107. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
STXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STXRB* on page K1-5512.

### Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

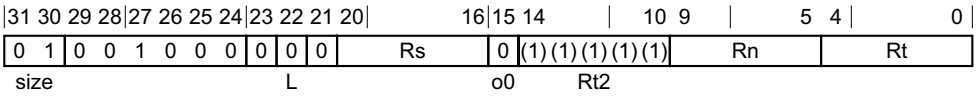
if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t]  = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t]  = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t]  = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t]  = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.194 STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-107. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes* on page C1-126.




### *No offset variant*

```
STXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

### *Decode for this encoding*

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

`0` If the operation updates memory.

`1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable();
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusive Monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusive Monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusive Monitor will only be set if all the reads are from the
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

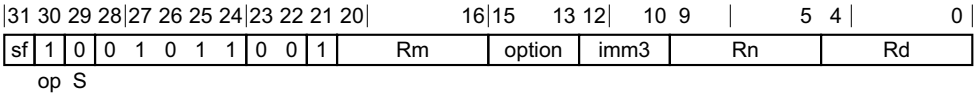
```
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t]  = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t]  = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t]  = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t]  = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.195 SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.




### 32-bit variant

Applies when sf = 0.

```
SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

### 64-bit variant

Applies when sf = 1.

```
SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

## Assembler symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

| | |
|---|---|
| W | when option = 00x |
| W | when option = 010 |
| X | when option = x11 |
| W | when option = 10x |
| W | when option = 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

if d == 31 && !setflags then
```

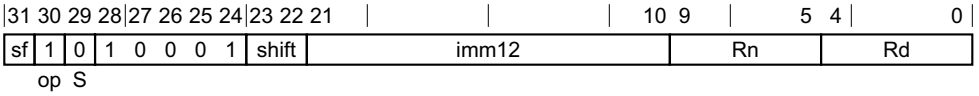

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.196 SUB (immediate)

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.




#### *32-bit variant*

Applies when `sf = 0`.

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

#### *64-bit variant*

Applies when `sf = 1`.

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
    when '1x' ReservedValue();
```

### Assembler symbols

`<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "shift" field. It can have the following values:

`LSL #0` when `shift = 00`

`LSL #12` when `shift = 01`

The encoding `shift = 1x` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

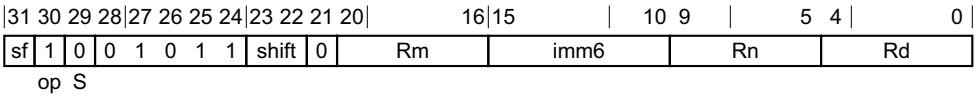
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.197 SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias NEG (shifted register). See the *Alias conditions* table for details of when each alias is preferred.




### *32-bit variant*

Applies when `sf = 0`.

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### *64-bit variant*

Applies when `sf = 1`.

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| NEG (shifted register) | `Rn == '11111'` |

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Wm>` | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

LSL when shift = 00

LSR when shift = 01

ASR when shift = 10

The encoding shift = 11 is reserved.

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

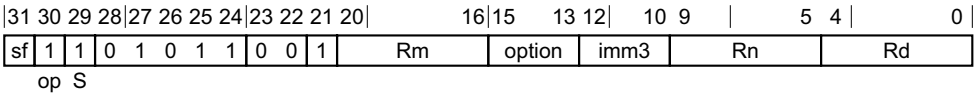
X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.198 SUBS (extended register)

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMP (extended register). See the *Alias conditions* table for details of when each alias is preferred.




### *32-bit variant*

Applies when `sf = 0`.

```
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

### *64-bit variant*

Applies when `sf = 1`.

```
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| CMP (extended register) | `Rd == '11111'` |

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn\|WSP> | Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn\|SP> | Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

W when option = 00x

W when option = 010

X when option = x11

W when option = 10x

W when option = 110

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

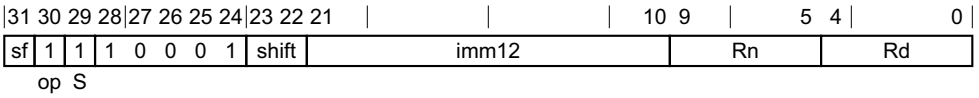
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.199 SUBS (immediate)

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMP (immediate). See the *Alias conditions* table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0`.

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

### 64-bit variant

Applies when `sf = 1`.

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
    when '1x' ReservedValue();
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| CMP (immediate) | `Rd == '11111'` |

## Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "shift" field. It can have the following values:

`LSL #0` when `shift = 00`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`LSL #12` when `shift = 01`

The encoding `shift = 1x` is reserved.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

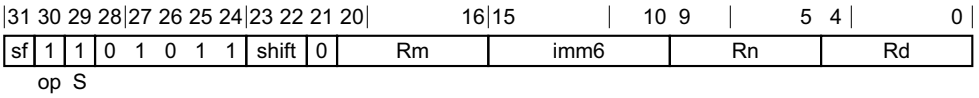
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.200 SUBS (shifted register)

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases CMP (shifted register) and NEGS. See the *Alias conditions* table for details of when each alias is preferred.




### *32-bit variant*

Applies when `sf = 0`.

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### *64-bit variant*

Applies when `sf = 1`.

```
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| CMP (shifted register) | `Rd == '11111'` |
| NEGS | `Rn == '11111'` |

## Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

LSL when shift = 00

LSR when shift = 01

ASR when shift = 10

The encoding shift = 11 is reserved.

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

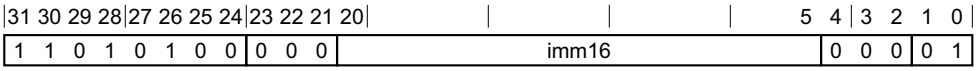

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.201 SVC

Supervisor Call causes an exception to be taken to EL1.

On executing an SVC instruction, the PE records the exception as a Supervisor Call exception, using the EC value 0x15, and the value of the immediate argument. This is reported in:

- ESR_ELx, if the exception is taken to an Exception level that is using AArch64.
- HSR, if the exception is taken to an Exception level that is using AArch32. See *Use of the HSR* on page G4-4105.




### *System variant*

```
SVC #<imm>
```

### *Decode for this encoding*

```
bits(16) imm = imm16;
```

### Assembler symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

### Operation

```
AArch64.CallSupervisor(imm);
```

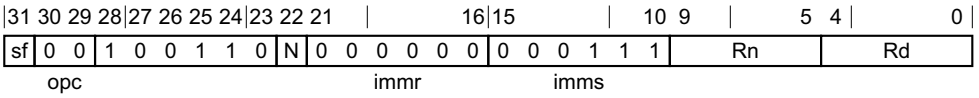

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.202 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0 && N = 0`.

```
SXTB <Wd>, <Wn>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1 && N = 1`.

```
SXTB <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #7
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| <Wn> | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |

### Operation

The description of SBFM gives the operational pseudocode for this instruction.

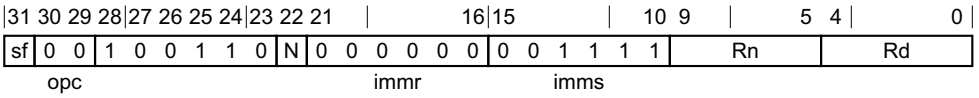

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.203 SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0 && N = 0`.

`SXTH <Wd>, <Wn>`

is equivalent to

`SBFM <Wd>, <Wn>, #0, #15`

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1 && N = 1`.

`SXTH <Xd>, <Wn>`

is equivalent to

`SBFM <Xd>, <Xn>, #0, #15`

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |

### Operation

The description of SBFM gives the operational pseudocode for this instruction.

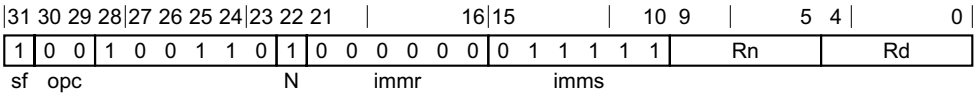

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.204 SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.




### *64-bit variant*

```
SXTW <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #31
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |

### Operation

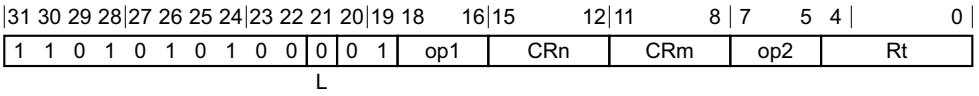
The description of SBFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.205 SYS

System instruction. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-271 for the encodings of System instructions.

This instruction is used by the aliases AT, DC, IC, and TLBI. See the *Alias conditions* table for details of when each alias is preferred.




### System variant

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

### Decode for this encoding

```
CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 1;
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
boolean has_result = (L == '1');
```

### Alias conditions

| Alias | is preferred when |
|---|---|
| AT | `CRn == '0111' && CRm == '1000' && SysOp(op1,'0111','1000',op2) == Sys_AT` |
| DC | `CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_DC` |
| IC | `CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_IC` |
| TLBI | `CRn == '1000' && SysOp(op1,'1000',CRm,op2) == Sys_TLBI` |

### Assembler symbols

| | |
|---|---|
| <op1> | Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field. |
| <Cn> | Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field. |
| <Cm> | Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field. |
| <op2> | Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field. |
| <Xt> | Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

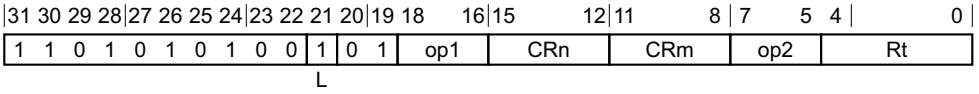
## Operation

```
if has_result then
    X[t] = SysOp_R(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    SysOp_W(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.206 SYSL

System instruction with result. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-271 for the encodings of System instructions.




### System variant

```
SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

### Decode for this encoding

```
CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 1;
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
boolean has_result = (L == '1');
```

## Assembler symbols

| | |
|---|---|
| <Xt> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field. |
| <op1> | Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field. |
| <Cn> | Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field. |
| <Cm> | Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field. |
| <op2> | Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field. |

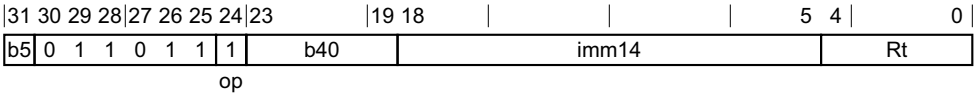
## Operation

```
if has_result then
    X[t] = SysOp_R(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    SysOp_W(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.207 TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.




### 14-bit signed PC-relative branch offset variant

```
TBNZ <R><t>, #<imm>, <label>
```

### Decode for this encoding

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bit bit_val = op;
bits(64) offset = SignExtend(imm14:'00', 64);
```

## Assembler symbols

<R> Is a width specifier, encoded in the "b5" field. It can have the following values:

W when b5 = 0

X when b5 = 1

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

## Operation

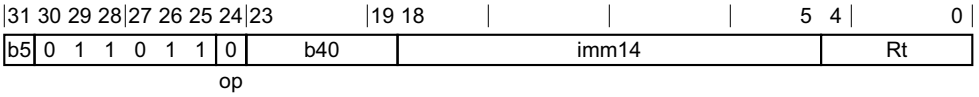
```
bits(datasize) operand = X[t];

if operand<bit_pos> == bit_val then
    BranchTo(PC[] + offset, BranchType_JMP);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.208 TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.




### 14-bit signed PC-relative branch offset variant

```
TBZ <R><t>, #<imm>, <label>
```

### Decode for this encoding

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bit bit_val = op;
bits(64) offset = SignExtend(imm14:'00', 64);
```

## Assembler symbols

<R> Is a width specifier, encoded in the "b5" field. It can have the following values:

W when b5 = 0

X when b5 = 1

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

## Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == bit_val then
    BranchTo(PC[] + offset, BranchType_JMP);
```

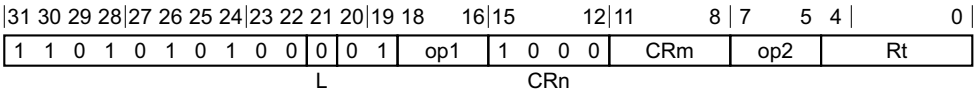

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.209 TLBI

TLB Invalidate operation. For more information, see A64 system instructions for TLB maintenance.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.




### *System variant*

```
TLBI <tlbi_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, C8, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1,'1000',CRm,op2) == Sys_TLBI`.

### Assembler symbols

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<tlbi_op>` Is a TLBI operation name, as listed for the TLBI system operation group, encoded in the "op1:CRm:op2" field. It can have the following values:

| | |
|---|---|
| `VMALLE1IS` | when `op1 = 000, CRm = 0011, op2 = 000` |
| `VAE1IS` | when `op1 = 000, CRm = 0011, op2 = 001` |
| `ASIDE1IS` | when `op1 = 000, CRm = 0011, op2 = 010` |
| `VAAE1IS` | when `op1 = 000, CRm = 0011, op2 = 011` |
| `VALE1IS` | when `op1 = 000, CRm = 0011, op2 = 101` |
| `VAALE1IS` | when `op1 = 000, CRm = 0011, op2 = 111` |
| `VMALLE1` | when `op1 = 000, CRm = 0111, op2 = 000` |
| `VAE1` | when `op1 = 000, CRm = 0111, op2 = 001` |
| `ASIDE1` | when `op1 = 000, CRm = 0111, op2 = 010` |
| `VAAE1` | when `op1 = 000, CRm = 0111, op2 = 011` |
| `VALE1` | when `op1 = 000, CRm = 0111, op2 = 101` |
| `VAALE1` | when `op1 = 000, CRm = 0111, op2 = 111` |
| `IPAS2E1IS` | when `op1 = 100, CRm = 0000, op2 = 001` |
| `IPAS2LE1IS` | when `op1 = 100, CRm = 0000, op2 = 101` |
| `ALLE2IS` | when `op1 = 100, CRm = 0011, op2 = 000` |
| `VAE2IS` | when `op1 = 100, CRm = 0011, op2 = 001` |
| `ALLE1IS` | when `op1 = 100, CRm = 0011, op2 = 100` |
| `VALE2IS` | when `op1 = 100, CRm = 0011, op2 = 101` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| | |
|---|---|
| VMALLS12E1IS | when op1 = 100, CRm = 0011, op2 = 110 |
| IPAS2E1 | when op1 = 100, CRm = 0100, op2 = 001 |
| IPAS2LE1 | when op1 = 100, CRm = 0100, op2 = 101 |
| ALLE2 | when op1 = 100, CRm = 0111, op2 = 000 |
| VAE2 | when op1 = 100, CRm = 0111, op2 = 001 |
| ALLE1 | when op1 = 100, CRm = 0111, op2 = 100 |
| VALE2 | when op1 = 100, CRm = 0111, op2 = 101 |
| VMALLS12E1 | when op1 = 100, CRm = 0111, op2 = 110 |
| ALLE3IS | when op1 = 110, CRm = 0011, op2 = 000 |
| VAE3IS | when op1 = 110, CRm = 0011, op2 = 001 |
| VALE3IS | when op1 = 110, CRm = 0011, op2 = 101 |
| ALLE3 | when op1 = 110, CRm = 0111, op2 = 000 |
| VAE3 | when op1 = 110, CRm = 0111, op2 = 001 |
| VALE3 | when op1 = 110, CRm = 0111, op2 = 101 |

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

## Operation

The description of SYS gives the operational pseudocode for this instruction.

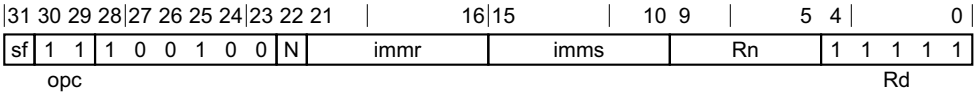

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.210 TST (immediate)

Test bits (immediate) , setting the condition flags and discarding the result : `Rn AND imm`

This instruction is an alias of the ANDS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode for this instruction.




#### *32-bit variant*

Applies when `sf = 0 && N = 0`.

```
TST <Wn>, #<imm>
```

is equivalent to

```
ANDS WZR, <Wn>, #<imm>
```

and is always the preferred disassembly.

#### *64-bit variant*

Applies when `sf = 1`.

```
TST <Xn>, #<imm>
```

is equivalent to

```
ANDS XZR, <Xn>, #<imm>
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<imm>` | For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".<br>For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr". |

### Operation

The description of ANDS (immediate) gives the operational pseudocode for this instruction.

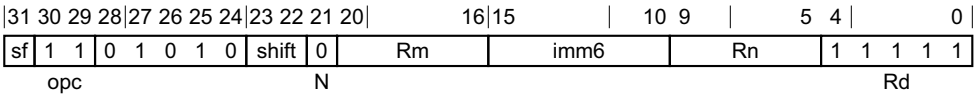

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.211 TST (shifted register)

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ANDS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0`.

```
TST <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

### *64-bit variant*

Applies when `sf = 1`.

```
TST <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

### Assembler symbols

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`
- `ROR` when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

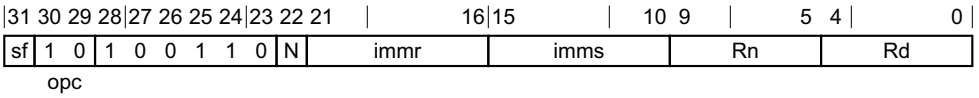

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.212 UBFIZ

Unsigned Bitfield Insert in Zero zeroes the destination register and copies any number of contiguous bits from a source register into any position in the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0 && N = 0`.

`UBFIZ <Wd>, <Wn>, #<lsb>, #<width>`

is equivalent to

`UBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)`

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

### *64-bit variant*

Applies when `sf = 1 && N = 1`.

`UBFIZ <Xd>, <Xn>, #<lsb>, #<width>`

is equivalent to

`UBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)`

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<lsb>` | For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.<br>For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63. |
| `<width>` | For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.<br>For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>. |

### Operation

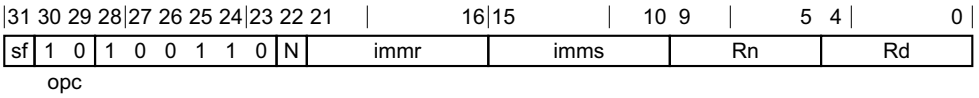
The description of UBFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.213 UBFM

Unsigned Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, with zeros in the upper and lower bits.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH. See the *Alias conditions* on page C6-819 table for details of when each alias is preferred.




### 32-bit variant

Applies when `sf = 0 && N = 0`.

```
UBFM <Wd>, <Wn>, #<immr>, #<imms>
```

### 64-bit variant

Applies when `sf = 1 && N = 1`.

```
UBFM <Xd>, <Xn>, #<immr>, #<imms>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

case opc of
    when '00' inzero = TRUE;  extend = TRUE;    // SBFM
    when '01' inzero = FALSE; extend = FALSE;   // BFM
    when '10' inzero = TRUE;  extend = FALSE;   // UBFM
    when '11' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Alias conditions

| Alias | of variant | is preferred when |
|---|---|---|
| LSL (immediate) | 32-bit | `imms != '011111' && imms + 1 == immr` |
| LSL (immediate) | 64-bit | `imms != '111111' && imms + 1 == immr` |
| LSR (immediate) | 32-bit | `imms == '011111'` |
| LSR (immediate) | 64-bit | `imms == '111111'` |
| UBFIZ | - | `UInt(imms) < UInt(immr)` |
| UBFX | - | `BFXPreferred(sf, opc<1>, imms, immr)` |
| UXTB | - | `immr == '000000' && imms == '000111'` |
| UXTH | - | `immr == '000000' && imms == '001111'` |

### Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<immr> For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

<imms> For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

### Operation

```
bits(datasize) dst = if inzero then Zeros() else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

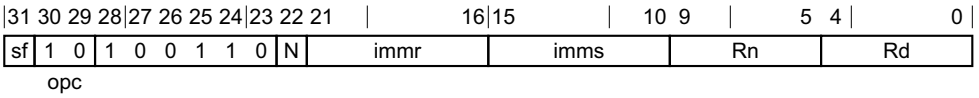

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.214 UBFX

Unsigned Bitfield Extract extracts any number of adjacent bits at any position from a register, zero-extends them to the size of the register, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `sf = 0 && N = 0`.

`UBFX <Wd>, <Wn>, #<lsb>, #<width>`

is equivalent to

`UBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)`

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

### *64-bit variant*

Applies when `sf = 1 && N = 1`.

`UBFX <Xd>, <Xn>, #<lsb>, #<width>`

is equivalent to

`UBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)`

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Wn>` | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xn>` | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| `<lsb>` | For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.<br>For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63. |
| `<width>` | For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.<br>For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>. |

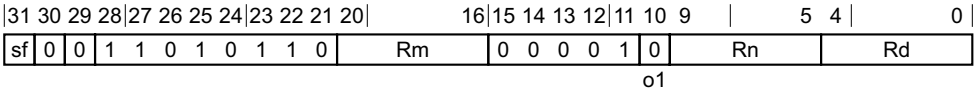
## Operation

The description of UBFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.215 UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.




### 32-bit variant

Applies when `sf = 0`.

```
UDIV <Wd>, <Wn>, <Wm>
```

### 64-bit variant

Applies when `sf = 1`.

```
UDIV <Xd>, <Xn>, <Xm>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean unsigned = (o1 == '0');
```

## Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, unsigned)) / Real(Int(operand2, unsigned)));

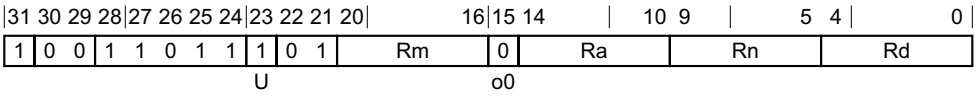
X[d] = result<datasize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.216 UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMULL. See the *Alias conditions* table for details of when each alias is preferred.




### 64-bit variant

```
UMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| UMULL | `Ra == '11111'` |

## Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

if sub_op then
    result = Int(operand3, unsigned) - (Int(operand1, unsigned) * Int(operand2, unsigned));
else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    result = Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```

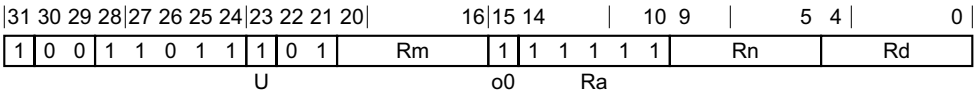

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.217 UMNEGL

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This instruction is an alias of the UMSUBL instruction. This means that:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode for this instruction.




### *64-bit variant*

```
UMNEGL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMSUBL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

### Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

### Operation

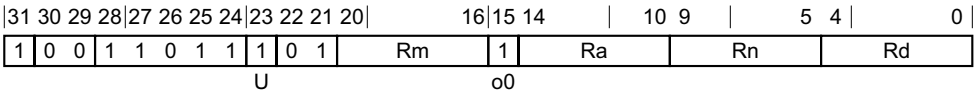
The description of UMSUBL gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.218 UMSUBL

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMNEGL. See the *Alias conditions* table for details of when each alias is preferred.




### 64-bit variant

```
UMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| UMNEGL | `Ra == '11111'` |

## Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

if sub_op then
    result = Int(operand3, unsigned) - (Int(operand1, unsigned) * Int(operand2, unsigned));
else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

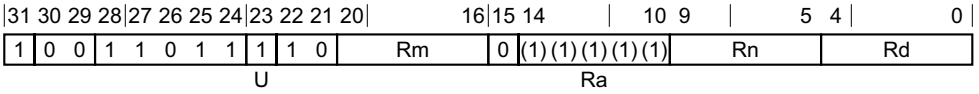
```
        result = Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

    X[d] = result<63:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.219 UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.




### 64-bit variant

```
UMULH <Xd>, <Xn>, <Xm>
```

### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);           // ignored by UMULH/SMULH
integer destsize = 64;
integer datasize = destsize;
boolean unsigned = (U == '1');
```

## Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

integer result;

result = Int(operand1, unsigned) * Int(operand2, unsigned);

X[d] = result<127:64>;
```

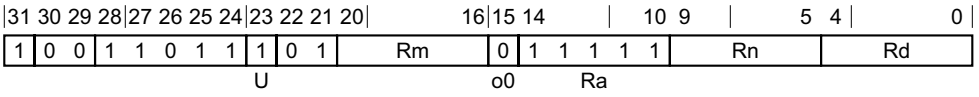

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.220 UMULL

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This instruction is an alias of the UMADDL instruction. This means that:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode for this instruction.




### *64-bit variant*

```
UMULL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMADDL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

### Assembler symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

### Operation

The description of UMADDL gives the operational pseudocode for this instruction.

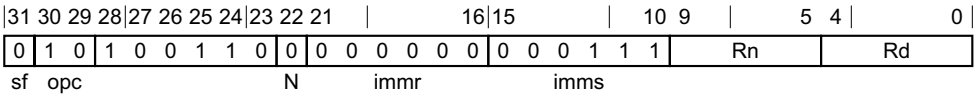

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.221 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.




### *32-bit variant*

`UXTB <Wd>, <Wn>`

is equivalent to

`UBFM <Wd>, <Wn>, #0, #7`

and is always the preferred disassembly.

### Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

### Operation

The description of UBFM gives the operational pseudocode for this instruction.

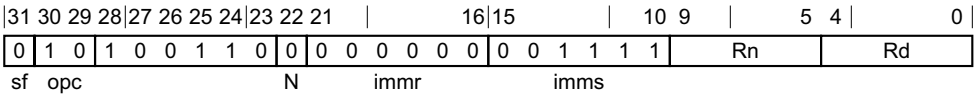

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C6.6.222 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.




### *32-bit variant*

```
UXTH <Wd>, <Wn>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

### Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

### Operation

The description of UBFM gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.223 WFE

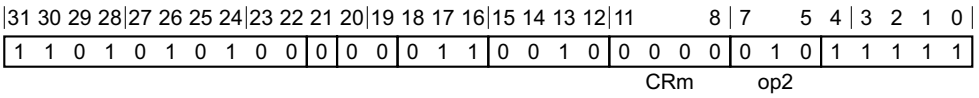
Wait For Event is a hint instruction that permits the PE to enter a low-power state until one of a number of events occurs, including events signaled by executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event* on page D1-1611.

As described in *Wait for Event mechanism and Send event* on page D1-1611, the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-1578.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page D1-1594.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-1603.

This instruction is an alias of the HINT instruction. This means that:

- The encodings in this description are named to match the encodings of HINT.
- The description of HINT gives the operational pseudocode for this instruction.




#### *System variant*

```
WFE
```

is equivalent to

```
HINT #2
```

and is always the preferred disassembly.

### Operation

The description of HINT gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C6.6.224 WFI

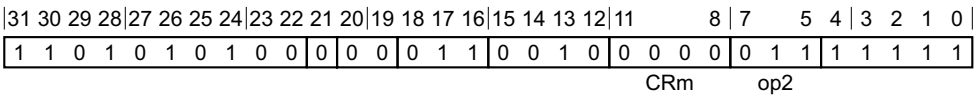
Wait For Interrupt is a hint instruction that permits the PE to enter a low-power state until one of a number of asynchronous event occurs. For more information, see *Wait For Interrupt* on page D1-1614.

As described in *Wait For Interrupt* on page D1-1614, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-1578.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page D1-1594.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-1603.

This instruction is an alias of the HINT instruction. This means that:

- The encodings in this description are named to match the encodings of HINT.
- The description of HINT gives the operational pseudocode for this instruction.




#### *System variant*

`WFI`

is equivalent to

`HINT #3`

and is always the preferred disassembly.

### Operation

The description of HINT gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

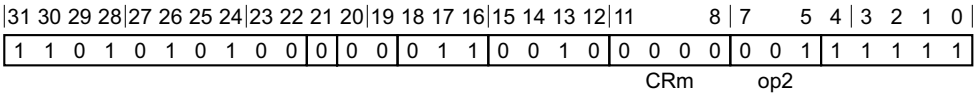
## C6.6.225 YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see *The YIELD instruction* on page B1-64.

This instruction is an alias of the HINT instruction. This means that:

- The encodings in this description are named to match the encodings of HINT.
- The description of HINT gives the operational pseudocode for this instruction.




### *System variant*

`YIELD`

is equivalent to

`HINT #1`

and is always the preferred disassembly.

### Operation

The description of HINT gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter C7
# A64 Advanced SIMD and Floating-point Instruction Descriptions

This chapter describes the A64 SIMD and floating-point instructions.

It contains the following sections:


- *Introduction* on page C7-836.
- *About the SIMD and floating-point instructions* on page C7-837.
- *Alphabetical list of floating-point and Advanced SIMD instructions* on page C7-839.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## C7.1 Introduction

*Alphabetical list of floating-point and Advanced SIMD instructions* on page C7-839 is an alphabetical list of instructions that are part of the following two functional groups:

- Loads and store instructions associated with the SIMD and floating-point registers.
- Data processing instructions with SIMD and floating-point registers.

*A64 instruction index by encoding* on page C4-186 in the A64 Instruction Encodings chapter provides an overview of the instruction encodings as part of an instruction class within a functional group.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.2 About the SIMD and floating-point instructions

This section provides a general description of the SIMD and floating-point instructions. It contains the following subsections:

- *Register size*.
- *Data types*.
- *Condition flags and related instructions* on page C7-838.
- *General capabilities* on page C7-838.

### C7.2.1 Register size

A64 provides a comprehensive set of packed Single Instruction Multiple Data (SIMD) and scalar operations using data held in the 32 entry 128-bit wide SIMD and floating-point register file.

Each SIMD and floating-point register can be used to hold:

- A single scalar value of the floating-point or integer type.
- A 64-bit wide vector containing one or more elements.
- A 128-bit wide vector containing two or more elements.

Where the entire 128-bit wide register is not fully utilized, the vector or scalar quantity is held in the least significant bits of the register, with the most significant bits being cleared to zero on a write, see *Vector formats* on page A1-37.

The following instructions can insert data into individual elements within a SIMD and floating-pointer register without clearing the remaining bits to zero:

- Insert vector element from another vector element or general-purpose register, `INS`.
- Load structure into a single lane, for example `LD3`.
- All second-part narrowing operations, for example `SHRN2`.

### C7.2.2 Data types

The A64 instruction set provides support for arithmetic, conversion, and bitwise operations on:

- Half-precision, single-precision, and double-precision floating points.
- Signed and unsigned integers.
- Polynomials over {0, 1}.

For all AArch64 floating-point operations, including SIMD operations, the rounding mode and exception trap handling are controlled by the FPCR.

**Note**

AArch32 Advanced SIMD operations always use ARM standard floating-point arithmetic independent of the FPCR and FPSCR rounding mode. In addition, floating-point multiply-addition operations in AArch64 are always performed as fused operations, whereas AArch32 provides both fused and chained variants.

In addition to operations that consume and produce values of the same width and type, the A64 instruction set supports SIMD and scalar operations that produce a wider or narrower vector result:

- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the A64 instruction set provides a second-part operation, for example `SHRN2`, that can pack the result of a second operation into the upper part of the same destination register.
- Where a SIMD operation widens a 64-bit vector to a 128-bit vector, the A64 instruction set provides a second-part operation, for example `SMLAL2`, that can extract the source from the upper 64 bits of the source registers.

All SIMD operations that could produce side-effects that are not limited to the destination SIMD and floating-point register, for example a potential update of FPSR.Q or FPSR.IDC, have a dedicated scalar variant to support the use of SIMD with loops requiring specialised head or tail handling, or both.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C7.2.3 Condition flags and related instructions

The A64 instruction set provides support for flag setting and conditional operations on the SIMD and floating-point register file:

- Floating-point FCSEL and FCCMP instructions are equivalent to the integer CSEL and CCMP instructions.
- Floating-point FCMP. FCMPE, FCCMP, and FCCMP set the PSTATE.{N, Z, C, V} flags based on the result of the floating-point comparison.
- Floating-point and integer instructions provide a means of producing either a scalar or a vector mask based on a comparison in a SIMD and floating-point register, for example FCMEQ.

**Note**

FCMP and FCMPE differ from the A32/T32 VCMP and VCMPE instructions, which use the dedicated FPSCR.NZCV field for the result. A64 instructions store the result of an FCMP or FCMPE operation in the PSTATE.{N, Z, C, V} field.

### C7.2.4 General capabilities

A64 SIMD and floating-point instructions provide the following capabilities:

- General arithmetic on vector and scalar floating-point and integer values.
- Dedicated polynomial multiply over {0, 1}.
- Vector and scalar fused multiply-addition of single-precision and double-precision floating-points.
- Load and store of single and pairs of SIMD and floating-point registers.
- Load and store of structures and individual lanes of between one and four SIMD and floating-point registers.
- Direct conversion between 64-bit integers and floating-point values, with explicit rounding.
- Double-rounding free conversion between double-precision and half-precision floating-point values.
- Comprehensive SIMD with widening and narrowing support.
- Vector to scalar reduction returning the minimum or maximum value, or the sum.
- Floating-point to nearest integer in floating-point format.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3 Alphabetical list of floating-point and Advanced SIMD instructions

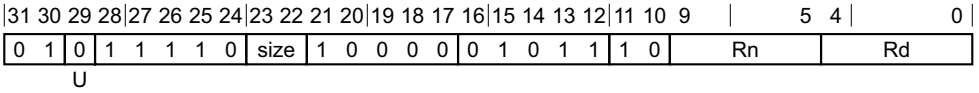
This section lists every section in the floating-point and Advanced SIMD categories of the A64 instruction set. For details of the format used, see *Structure of the A64 assembler language* on page C1-121.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.1 ABS

Absolute value (vector)

### Scalar




#### *Scalar variant*

```
ABS <V><d>, <V><n>
```

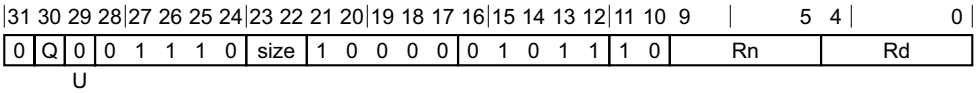
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean neg = (U == '1');
```

### Vector




#### *Vector variant*

```
ABS <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean neg = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

`D` when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

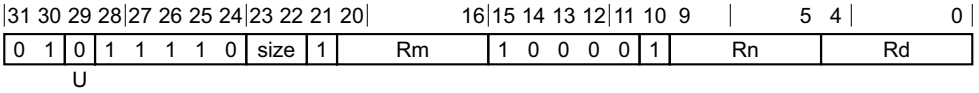
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.2 ADD (vector)

Add (vector)

### Scalar



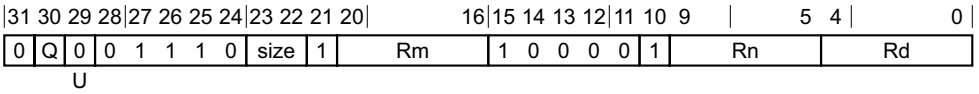

#### *Scalar variant*

```
ADD <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

### Vector




#### *Vector variant*

```
ADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

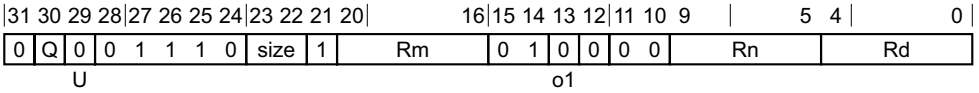
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.3 ADDHN, ADDHN2

Add returning high narrow




### *Three registers, not all the same type variant*

```
ADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize)   result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

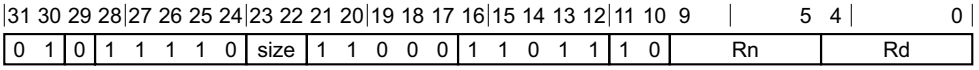
for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.4 ADDP (scalar)

Add pair of elements (scalar)




### *Advanced SIMD variant*

```
ADDP <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then ReservedValue();

integer esize = 8 << UInt(size);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = ReduceOp_ADD;
```

### Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in the "size" field. It can have the following values:

2D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

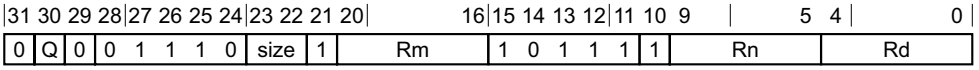
### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.5 ADDP (vector)

Add pairwise (vector)




### *Three registers of the same type variant*

```
ADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

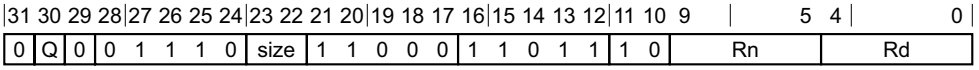
for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.6 ADDV

Add across vector




### *Advanced SIMD variant*

```
ADDV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = ReduceOp_ADD;
```

## Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

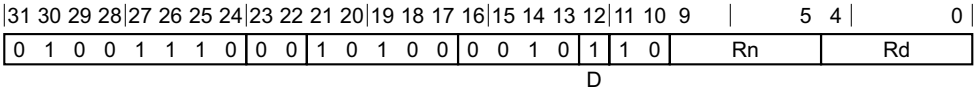
## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.7 AESD

AES single round decryption




### *Advanced SIMD variant*

```
AESD <Vd>.16B, <Vn>.16B
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckCryptoEnabled64();

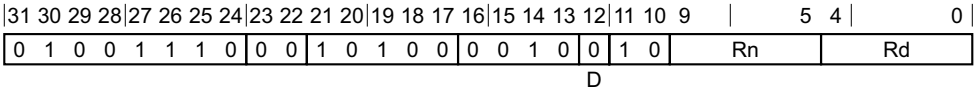
bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
if decrypt then
    result = AESInvSubBytes(AESInvShiftRows(result));
else
    result = AESSubBytes(AESShiftRows(result));

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.8 AESE

AES single round encryption




### *Advanced SIMD variant*

```
AESE <Vd>.16B, <Vn>.16B
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckCryptoEnabled64();

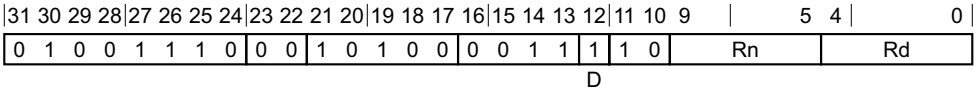
bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
if decrypt then
    result = AESInvSubBytes(AESInvShiftRows(result));
else
    result = AESSubBytes(AESShiftRows(result));

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.9 AESIMC

AES inverse mix columns




### *Advanced SIMD variant*

```
AESIMC <Vd>.16B, <Vn>.16B
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

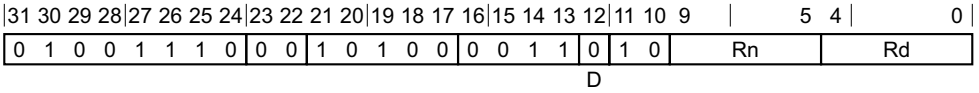
```
CheckCryptoEnabled64();

bits(128) operand = V[n];
bits(128) result;
if decrypt then
    result = AESInvMixColumns(operand);
else
    result = AESMixColumns(operand);
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.10 AESMC

AES mix columns




### *Advanced SIMD variant*

```
AESMC <Vd>.16B, <Vn>.16B
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

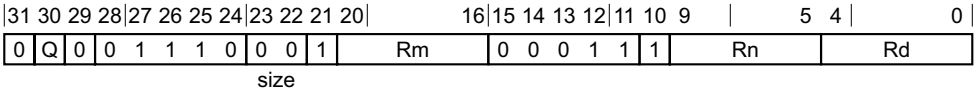
```
CheckCryptoEnabled64();

bits(128) operand = V[n];
bits(128) result;
if decrypt then
    result = AESInvMixColumns(operand);
else
    result = AESMixColumns(operand);
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.11 AND (vector)

Bitwise AND (vector)




### *Three registers of the same type variant*

```
AND <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean invert = (size<0> == '1');
LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

8B when Q = 0

16B when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

if invert then operand2 = NOT(operand2);

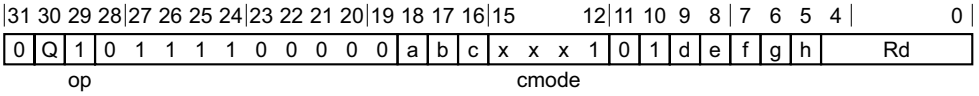
case op of
    when LogicalOp_AND
        result = operand1 AND operand2;
    when LogicalOp_ORR
        result = operand1 OR operand2;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.12 BIC (vector, immediate)

Bitwise bit clear (vector, immediate)




### *16-bit variant*

Applies when `cmode = 10x1`.

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### *32-bit variant*

Applies when `cmode = 0xx1`.

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### *Decode for all variants of this encoding*

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when `Q = 0`
- 8H when `Q = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

2S when Q = 0

4S when Q = 1

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:

0 when cmode<1> = 0

8 when cmode<1> = 1

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:

0 when cmode<2:1> = 00

8 when cmode<2:1> = 01

16 when cmode<2:1> = 10

24 when cmode<2:1> = 11

defaulting to 0 if LSL is omitted.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

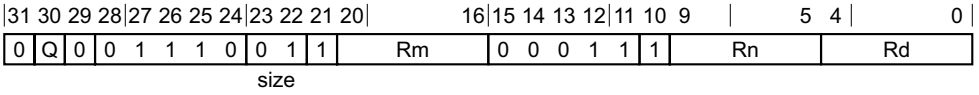
case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.13 BIC (vector, register)

Bitwise bit clear (vector, register)




### *Three registers of the same type variant*

```
BIC <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean invert = (size<0> == '1');
LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

8B when Q = 0

16B when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

if invert then operand2 = NOT(operand2);

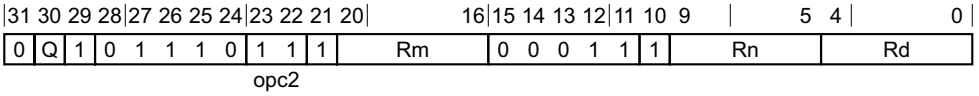
case op of
    when LogicalOp_AND
        result = operand1 AND operand2;
    when LogicalOp_ORR
        result = operand1 OR operand2;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.14 BIF

Bitwise insert if false




### *Three registers of the same type variant*

```
BIF <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

VBitOp op;

case opc2 of
    when '00' op = VBitOp_VEOR;
    when '01' op = VBitOp_VBSL;
    when '10' op = VBitOp_VBIT;
    when '11' op = VBitOp_VBIF;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

8B when Q = 0

16B when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

case op of
    when VBitOp_VEOR
        operand1 = V[m];
        operand2 = Zeros();
        operand3 = Ones();
    when VBitOp_VBSL
        operand1 = V[m];
        operand2 = operand1;
        operand3 = V[d];
    when VBitOp_VBIT
        operand1 = V[d];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

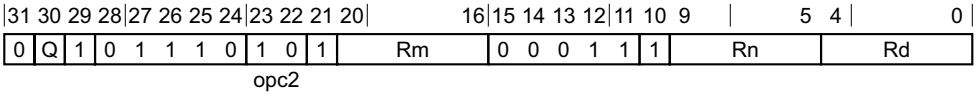
```
            operand2 = operand1;
            operand3 = V[m];
        when VBitOp_VBIF
            operand1 = V[d];
            operand2 = operand1;
            operand3 = NOT(V[m]);

    V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.15 BIT

Bitwise insert if true




### *Three registers of the same type variant*

```
BIT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

VBitOp op;

case opc2 of
    when '00' op = VBitOp_VEOR;
    when '01' op = VBitOp_VBSL;
    when '10' op = VBitOp_VBIT;
    when '11' op = VBitOp_VBIF;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

8B when Q = 0

16B when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

case op of
    when VBitOp_VEOR
        operand1 = V[m];
        operand2 = Zeros();
        operand3 = Ones();
    when VBitOp_VBSL
        operand1 = V[m];
        operand2 = operand1;
        operand3 = V[d];
    when VBitOp_VBIT
        operand1 = V[d];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

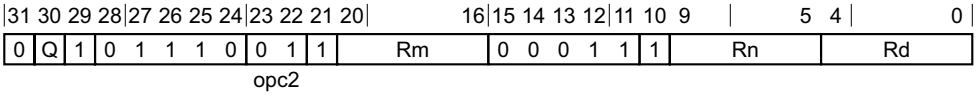
```
            operand2 = operand1;
            operand3 = V[m];
        when VBitOp_VBIF
            operand1 = V[d];
            operand2 = operand1;
            operand3 = NOT(V[m]);

    V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.16 BSL

Bitwise select




### *Three registers of the same type variant*

```
BSL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

VBitOp op;

case opc2 of
    when '00' op = VBitOp_VEOR;
    when '01' op = VBitOp_VBSL;
    when '10' op = VBitOp_VBIT;
    when '11' op = VBitOp_VBIF;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

8B when Q = 0

16B when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

case op of
    when VBitOp_VEOR
        operand1 = V[m];
        operand2 = Zeros();
        operand3 = Ones();
    when VBitOp_VBSL
        operand1 = V[m];
        operand2 = operand1;
        operand3 = V[d];
    when VBitOp_VBIT
        operand1 = V[d];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

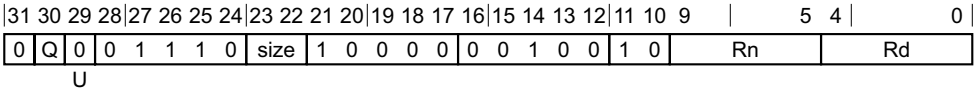
```
            operand2 = operand1;
            operand3 = V[m];
        when VBitOp_VBIF
            operand1 = V[d];
            operand2 = operand1;
            operand3 = NOT(V[m]);

    V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.17 CLS (vector)

Count leading sign bits (vector)




### *Vector variant*

```
CLS <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when size = 00, Q = 0 |
| 16B | when size = 00, Q = 1 |
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

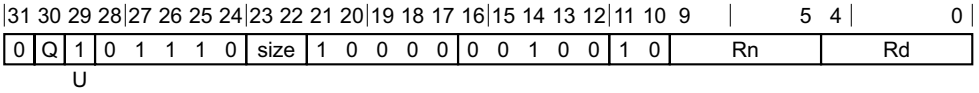
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.18 CLZ (vector)

Count leading zero bits (vector)




### *Vector variant*

```
CLZ <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when size = 00, Q = 0 |
| 16B | when size = 00, Q = 1 |
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

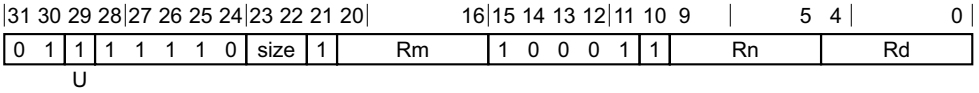
integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.19 CMEQ (register)

Compare bitwise equal (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar



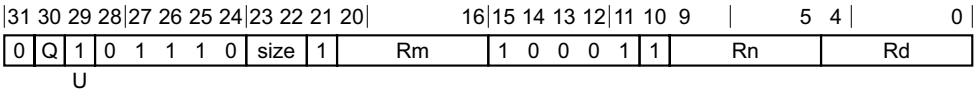

#### *Scalar variant*

```
CMEQ <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

### Vector




#### *Vector variant*

```
CMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

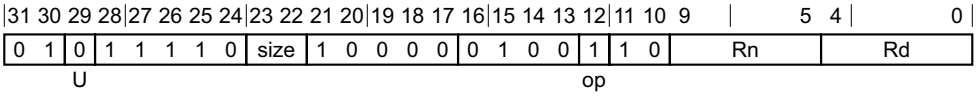
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.20 CMEQ (zero)

Compare bitwise equal to zero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

```
CMEQ <V><d>, <V><n>, #0
```

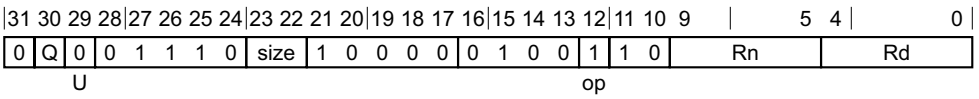
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

### Vector




#### *Vector variant*

```
CMEQ <Vd>.<T>, <Vn>.<T>, #0
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

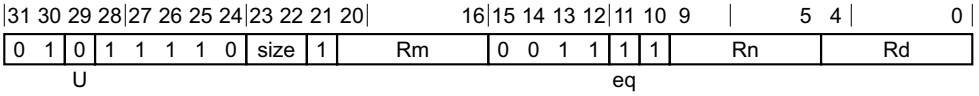
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.21 CMGE (register)

Compare signed greater than or equal (vector)

### Scalar



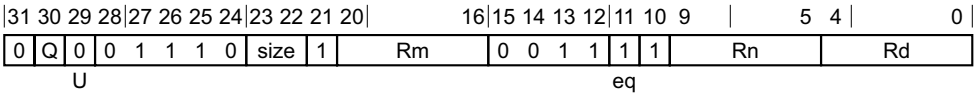

#### *Scalar variant*

```
CMGE <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Vector




#### *Vector variant*

```
CMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

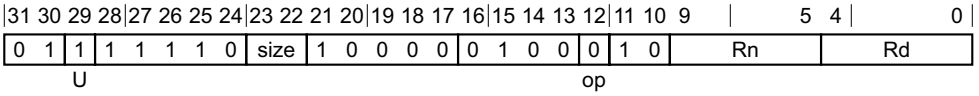
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.22 CMGE (zero)

Compare signed greater than or equal to zero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

```
CMGE <V><d>, <V><n>, #0
```

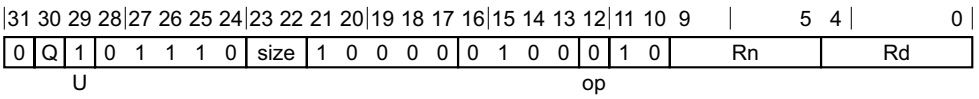
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

### Vector




#### *Vector variant*

```
CMGE <Vd>.<T>, <Vn>.<T>, #0
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

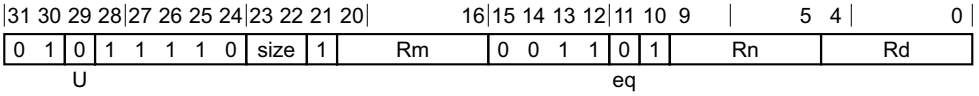
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.23 CMGT (register)

Compare signed greater than (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar



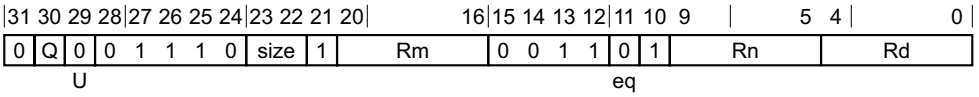

#### *Scalar variant*

```
CMGT <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Vector




#### *Vector variant*

```
CMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

`D` when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

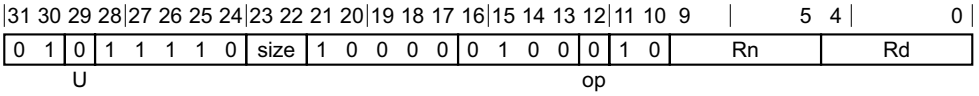
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.24 CMGT (zero)

Compare signed greater than zero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

```
CMGT <V><d>, <V><n>, #0
```

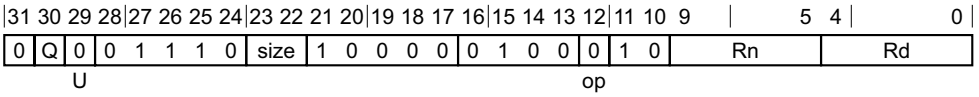
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

### Vector




#### *Vector variant*

```
CMGT <Vd>.<T>, <Vn>.<T>, #0
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

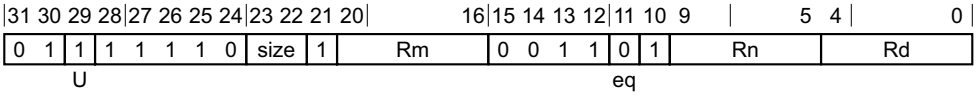
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.25 CMHI (register)

Compare unsigned higher (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar



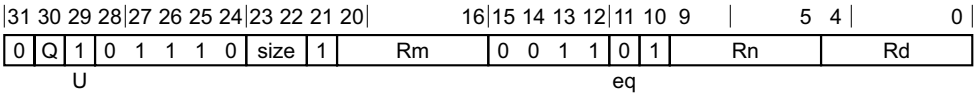

#### *Scalar variant*

```
CMHI <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Vector




#### *Vector variant*

```
CMHI <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

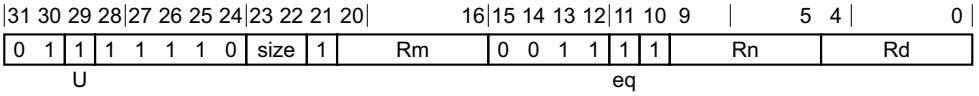
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.26 CMHS (register)

Compare unsigned higher or same (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar



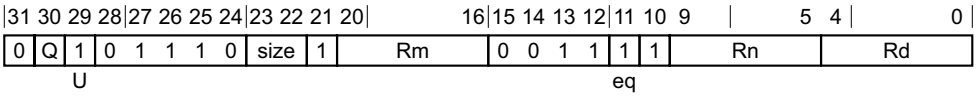

#### *Scalar variant*

```
CMHS <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Vector




#### *Vector variant*

```
CMHS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

`D` when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

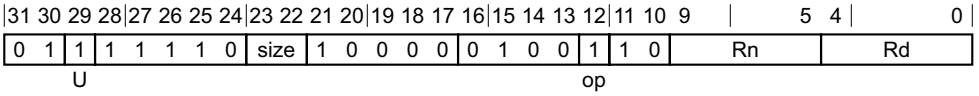
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.27 CMLE (zero)

Compare signed less than or equal to zero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

```
CMLE <V><d>, <V><n>, #0
```

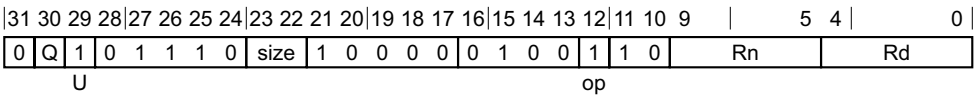
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

### Vector




#### *Vector variant*

```
CMLE <Vd>.<T>, <Vn>.<T>, #0
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

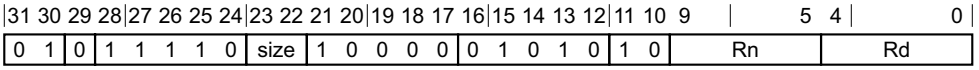
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.28 CMLT (zero)

Compare signed less than zero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

```
CMLT <V><d>, <V><n>, #0
```

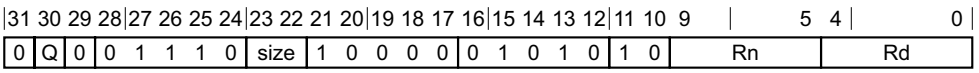
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

### Vector




#### *Vector variant*

```
CMLT <Vd>.<T>, <Vn>.<T>, #0
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

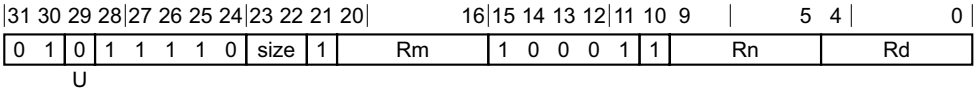
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.29 CMTST

Compare bitwise test bits nonzero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar



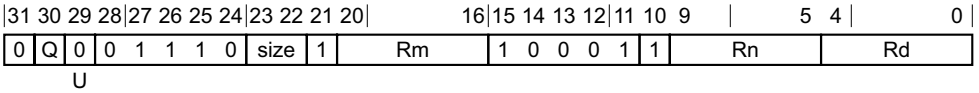

#### *Scalar variant*

```
CMTST <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

### Vector




#### *Vector variant*

```
CMTST <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

`D` when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

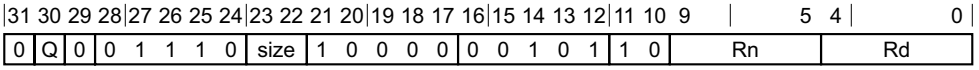
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.30 CNT

Population count per byte




### *Vector variant*

```
CNT <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '00' then ReservedValue();
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

The following encodings are reserved:

- size = 01, Q = x.
- size = 1x, Q = x.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    count = BitCount(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

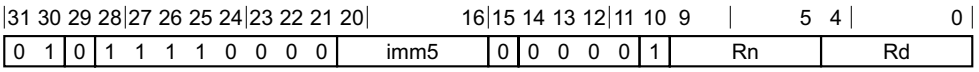

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.31 DUP (element)

Duplicate vector element to vector or scalar

This instruction is used by the alias MOV (scalar). The alias is always the preferred disassembly.

### Scalar




#### *Scalar variant*

```
DUP <V><d>, <Vn>.<T>[<index>]
```

#### *Decode for this encoding*

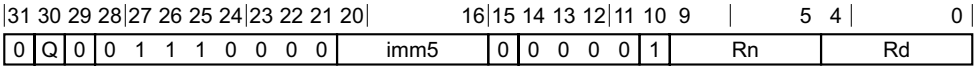
```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UnallocatedEncoding();

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;
```

### Vector




#### *Vector variant*

```
DUP <Vd>.<T>, <Vn>.<Ts>[<index>]
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UnallocatedEncoding();

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then ReservedValue();
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<T> For the scalar variant: is the element width specifier, encoded in the "imm5" field. It can have the following values:

B when imm5 = xxxx1

H when imm5 = xxx10

S when imm5 = xx100

D when imm5 = x1000

The encoding imm5 = x0000 is reserved.

For the vector variant: is an arrangement specifier, encoded in the "imm5:Q" field. It can have the following values:

8B when imm5 = xxxx1, Q = 0

16B when imm5 = xxxx1, Q = 1

4H when imm5 = xxx10, Q = 0

8H when imm5 = xxx10, Q = 1

2S when imm5 = xx100, Q = 0

4S when imm5 = xx100, Q = 1

2D when imm5 = x1000, Q = 1

The following encodings are reserved:

- imm5 = x0000, Q = x.
- imm5 = x1000, Q = 0.

<Ts> Is an element size specifier, encoded in the "imm5" field. It can have the following values:

B when imm5 = xxxx1

H when imm5 = xxx10

S when imm5 = xx100

D when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<V> Is the destination width specifier, encoded in the "imm5" field. It can have the following values:

B when imm5 = xxxx1

H when imm5 = xxx10

S when imm5 = xx100

D when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index> Is the element index encoded in the "imm5" field. It can have the following values:

imm5<4:1> when imm5 = xxxx1

imm5<4:2> when imm5 = xxx10

imm5<4:3> when imm5 = xx100

imm5<4> when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

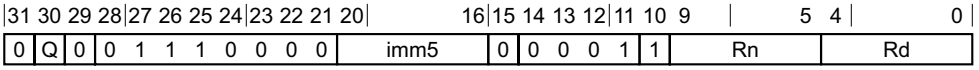
```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(datasize) result;
bits(esize) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.32 DUP (general)

Duplicate general-purpose register to vector




### *Advanced SIMD variant*

```
DUP <Vd>.<T>, <R><n>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UnallocatedEncoding();

// imm5<4:size+1> is IGNORED

if size == 3 && Q == '0' then ReservedValue();
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "imm5:Q" field. It can have the following values:

8B when `imm5 = xxxx1, Q = 0`

16B when `imm5 = xxxx1, Q = 1`

4H when `imm5 = xxx10, Q = 0`

8H when `imm5 = xxx10, Q = 1`

2S when `imm5 = xx100, Q = 0`

4S when `imm5 = xx100, Q = 1`

2D when `imm5 = x1000, Q = 1`

The following encodings are reserved:

- `imm5 = x0000, Q = x`.
- `imm5 = x1000, Q = 0`.

<R> Is the width specifier for the general-purpose source register, encoded in the "imm5" field. It can have the following values:

W when `imm5 = xxxx1`

W when `imm5 = xxx10`

W when `imm5 = xx100`

X when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

Unspecified bits in "imm5" are ignored but should be set to zero by an assembler.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

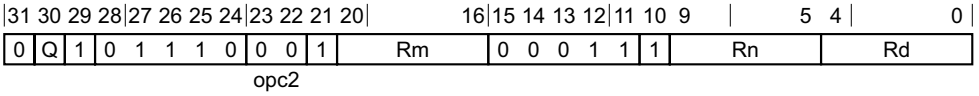
```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.33 EOR (vector)

Bitwise exclusive OR (vector)




### *Three registers of the same type variant*

```
EOR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

VBitOp op;

case opc2 of
    when '00' op = VBitOp_VEOR;
    when '01' op = VBitOp_VBSL;
    when '10' op = VBitOp_VBIT;
    when '11' op = VBitOp_VBIF;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

8B when Q = 0

16B when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

case op of
    when VBitOp_VEOR
        operand1 = V[m];
        operand2 = Zeros();
        operand3 = Ones();
    when VBitOp_VBSL
        operand1 = V[m];
        operand2 = operand1;
        operand3 = V[d];
    when VBitOp_VBIT
        operand1 = V[d];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

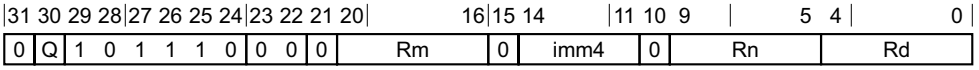
```
            operand2 = operand1;
            operand3 = V[m];
        when VBitOp_VBIF
            operand1 = V[d];
            operand2 = operand1;
            operand3 = NOT(V[m]);

    V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.34 EXT

Extract vector from pair of vectors




### *Advanced SIMD variant*

```
EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if Q == '0' && imm4<3> == '1' then UnallocatedEncoding();

integer datasize = if Q == '1' then 128 else 64;
integer position = UInt(imm4) << 3;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 8B when Q = 0
- 16B when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the lowest numbered byte element to be extracted, encoded in the "Q:imm4" field. It can have the following values:

- imm4<2:0> when Q = 0, imm4<3> = 0
- imm4 when Q = 1, imm4<3> = x

The encoding Q = 0, imm4<3> = 1 is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m];
bits(datasize) lo = V[n];
bits(datasize*2) concat = hi : lo;

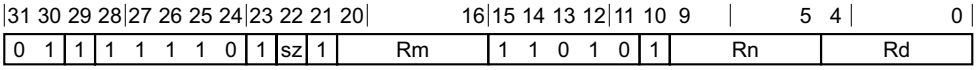
V[d] = concat<position+datasize-1:position>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.35 FABD

Floating-point absolute difference (vector)

### Scalar



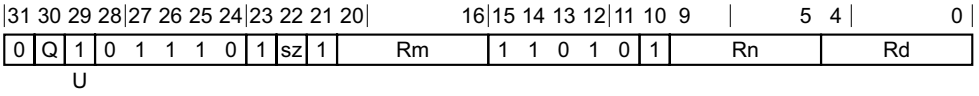

#### *Scalar variant*

```
FABD <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

### Vector




#### *Vector variant*

```
FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) diff;

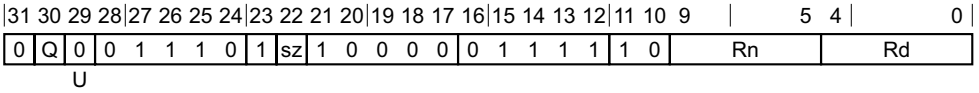
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.36 FABS (vector)

Floating-point absolute value (vector)




### *Vector variant*

```
FABS <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean neg = (U == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when `sz = 0, Q = 0`

4S when `sz = 0, Q = 1`

2D when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

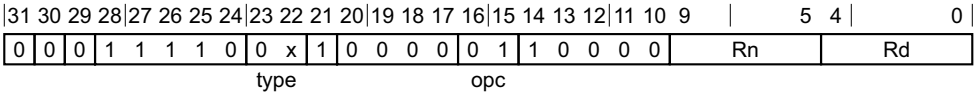
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.37 FABS (scalar)

Floating-point absolute value (scalar) : `Vd = abs(Vn)`




### *Single-precision variant*

Applies when `type = 00`.

```
FABS <Sd>, <Sn>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FABS <Dd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

FPUnaryOp fpop;
case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_NEG;
    when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
    when FPUnaryOp_MOV  result = operand;
    when FPUnaryOp_ABS  result = FPAbs(operand);
    when FPUnaryOp_NEG  result = FPNeg(operand);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when FPUnaryOp_SQRT result = FPSqrt(operand, FPCR);

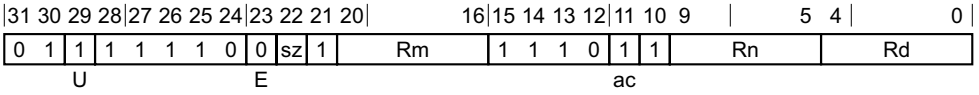
    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.38 FACGE

Floating-point absolute compare greater than or equal (vector)

### Scalar




#### *Scalar variant*

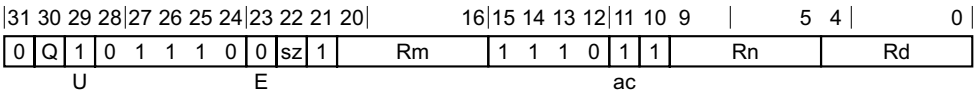
```
FACGE <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise  UnallocatedEncoding();
```

### Vector




#### *Vector variant*

```
FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise  UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

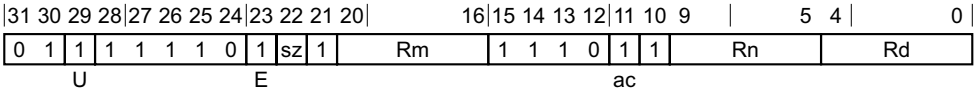
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.39 FACGT

Floating-point absolute compare greater than (vector)

### Scalar




#### *Scalar variant*

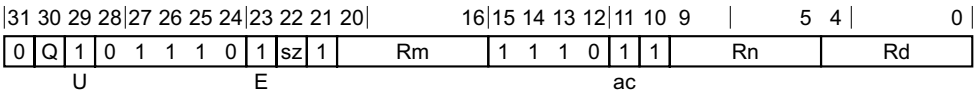
```
FACGT <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise  UnallocatedEncoding();
```

### Vector




#### *Vector variant*

```
FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise  UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

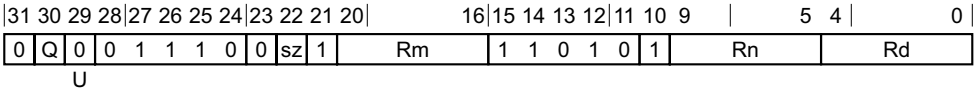
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.40 FADD (vector)

Floating-point add (vector)




### *Three registers of the same type variant*

```
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

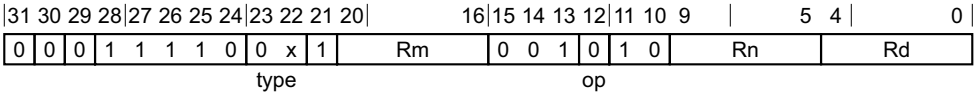
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.41 FADD (scalar)

Floating-point add (scalar) : `Vd = Vn + Vm`




### *Single-precision variant*

Applies when `type = 00`.

```
FADD <Sd>, <Sn>, <Sm>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FADD <Dd>, <Dn>, <Dm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean sub_op = (op == '1');
```

## Assembler symbols

| | |
|---|---|
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Dn>` | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Dm>` | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Sn>` | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Sm>` | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

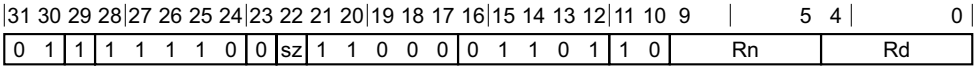
if sub_op then
    result = FPSub(operand1, operand2, FPCR);
else
    result = FPAdd(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.42 FADDP (scalar)

Floating-point add pair of elements (scalar)




### *Advanced SIMD variant*

```
FADDP <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = ReduceOp_FADD;
```

### Assembler symbols

<V> Is the destination width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- 2S when sz = 0
- 2D when sz = 1

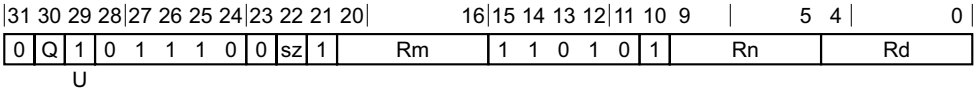
### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.43 FADDP (vector)

Floating-point add pairwise (vector)




### *Three registers of the same type variant*

```
FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

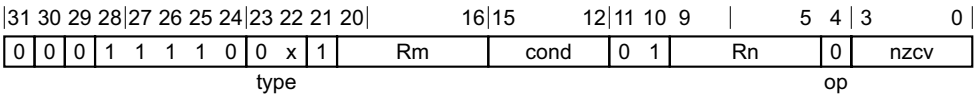
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.44 FCCMP

Floating-point conditional quiet compare (scalar) , setting condition flags to result of comparison or an immediate value : `flags = if cond then compareQuiet(Vn,Vm) else #nzcv`




### *Single-precision variant*

Applies when `type = 00`.

```
FCCMP <Sn>, <Sm>, #<nzcv>, <cond>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FCCMP <Dn>, <Dm>, #<nzcv>, <cond>
```

### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

## Assembler symbols

| | |
|---|---|
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| <nzcv> | Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field. |
| <cond> | Is one of the standard conditions, encoded in the "cond" field in the standard way. |

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];
```

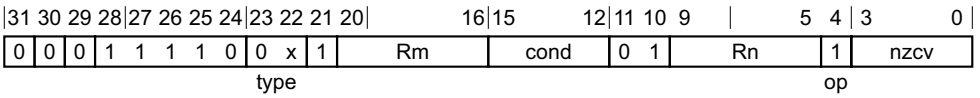

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if ConditionHolds(condition) then
    flags = FPCompare(operand1, operand2, signal_all_nans, FPCR);
PSTATE.<N,Z,C,V> = flags;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.45 FCCMPE

Floating-point conditional signaling compare (scalar) , setting condition flags to result of comparison or an immediate value : `flags = if cond then compareSignaling(Vn,Vm) else #nzcv`




### *Single-precision variant*

Applies when `type = 00`.

```
FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>
```

### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

## Assembler symbols

| | |
|---|---|
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| <nzcv> | Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field. |
| <cond> | Is one of the standard conditions, encoded in the "cond" field in the standard way. |

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

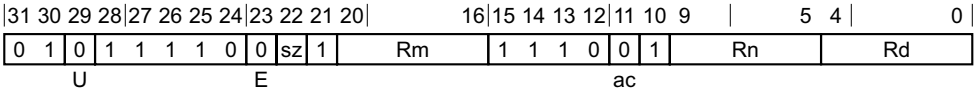
```
if ConditionHolds(condition) then
    flags = FPCompare(operand1, operand2, signal_all_nans, FPCR);
PSTATE.<N,Z,C,V> = flags;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.46 FCMEQ (register)

Floating-point compare equal (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

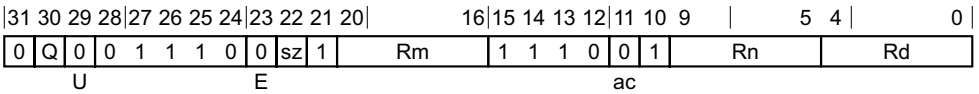
```
FCMEQ <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise  UnallocatedEncoding();
```

### Vector




#### *Vector variant*

```
FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise  UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

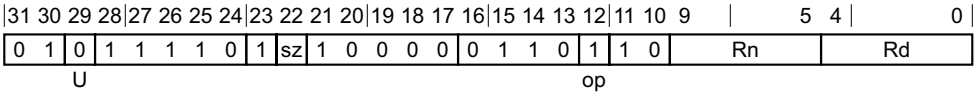
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.47 FCMEQ (zero)

Floating-point compare equal to zero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

```
FCMEQ <V><d>, <V><n>, #0.0
```

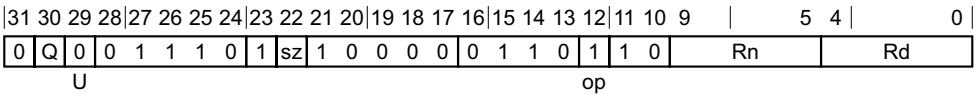
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

### Vector




#### *Vector variant*

```
FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

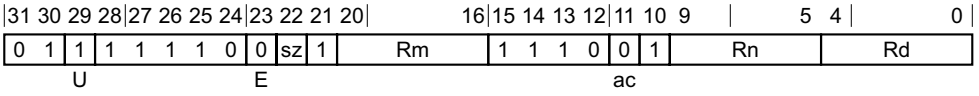
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.48 FCMGE (register)

Floating-point compare greater than or equal (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

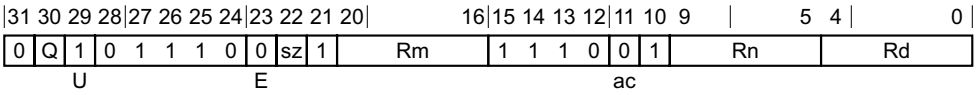
```
FCMGE <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise  UnallocatedEncoding();
```

### Vector




#### *Vector variant*

```
FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when '110' cmp = CompareOp_GT; abs = FALSE;
when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise  UnallocatedEncoding();
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

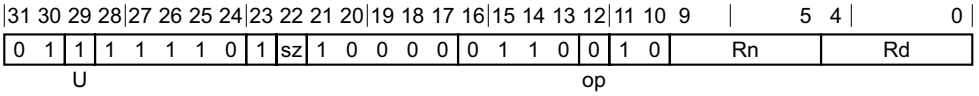
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.49 FCMGE (zero)

Floating-point compare greater than or equal to zero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

```
FCMGE <V><d>, <V><n>, #0.0
```

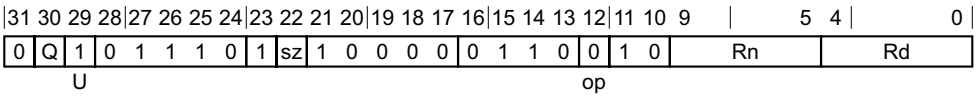
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

### Vector




#### *Vector variant*

```
FCMGE <Vd>.<T>, <Vn>.<T>, #0.0
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

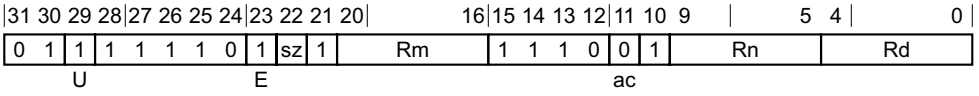
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.50 FCMGT (register)

Floating-point compare greater than (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

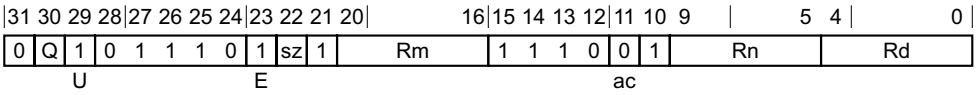
```
FCMGT <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise  UnallocatedEncoding();
```

### Vector




#### *Vector variant*

```
FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when '110' cmp = CompareOp_GT; abs = FALSE;
when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise  UnallocatedEncoding();
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

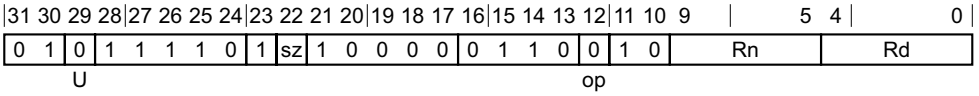
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.51 FCMGT (zero)

Floating-point compare greater than zero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

```
FCMGT <V><d>, <V><n>, #0.0
```

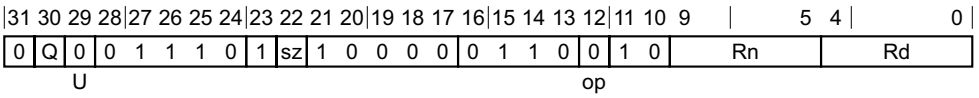
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

### Vector




#### *Vector variant*

```
FCMGT <Vd>.<T>, <Vn>.<T>, #0.0
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

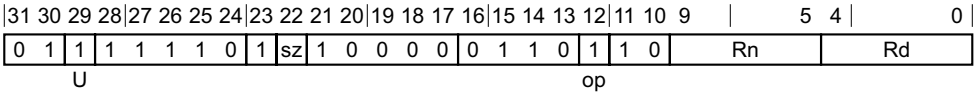
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.52 FCMLE (zero)

Floating-point compare less than or equal to zero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

```
FCMLE <V><d>, <V><n>, #0.0
```

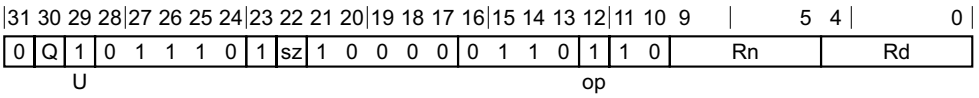
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

### Vector




#### *Vector variant*

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

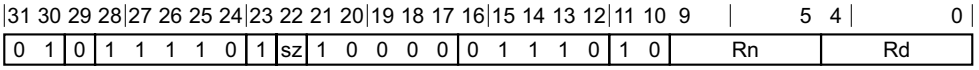
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.53 FCMLT (zero)

Floating-point compare less than zero (vector) , setting destination vector element to all ones if the condition holds, else zero

### Scalar




#### *Scalar variant*

```
FCMLT <V><d>, <V><n>, #0.0
```

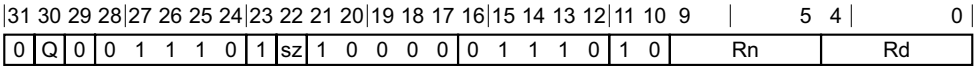
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

### Vector




#### *Vector variant*

```
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

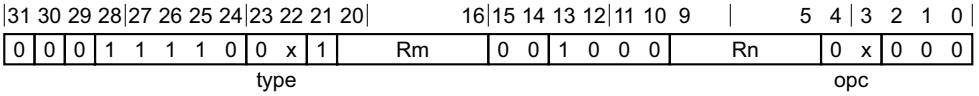
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.54 FCMP

Floating-point quiet compare (scalar) : `flags = compareQuiet(Vn, Vm) // with register`




### *Single-precision variant*

Applies when `type = 00 && opc = 00`.

```
FCMP <Sn>, <Sm>
```

### *Single-precision, zero variant*

Applies when `type = 00 && Rm = (00000) && opc = 01`.

```
FCMP <Sn>, #0.0
```

### *Double-precision variant*

Applies when `type = 01 && opc = 00`.

```
FCMP <Dn>, <Dm>
```

### *Double-precision, zero variant*

Applies when `type = 01 && Rm = (00000) && opc = 01`.

```
FCMP <Dn>, #0.0
```

### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer m = UInt(Rm);   // ignored when opc<0> == '1'

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

## Assembler symbols

<Dn> For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sn> For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

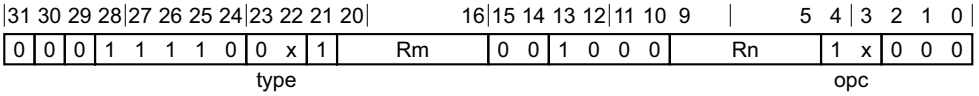
operand2 = if cmp_with_zero then FPZero('0') else V[m];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.55 FCMPE

Floating-point signaling compare (scalar) : `flags = compareSignaling(Vn, Vm) // with register`




### *Single-precision variant*

Applies when `type = 00 && opc = 10`.

```
FCMPE <Sn>, <Sm>
```

### *Single-precision, zero variant*

Applies when `type = 00 && Rm = (00000) && opc = 11`.

```
FCMPE <Sn>, #0.0
```

### *Double-precision variant*

Applies when `type = 01 && opc = 10`.

```
FCMPE <Dn>, <Dm>
```

### *Double-precision, zero variant*

Applies when `type = 01 && Rm = (00000) && opc = 11`.

```
FCMPE <Dn>, #0.0
```

### *Decode for all variants of this encoding*

```
integer n = UInt(Rn);
integer m = UInt(Rm);   // ignored when opc<0> == '1'

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

## Assembler symbols

<Dn> For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sn> For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

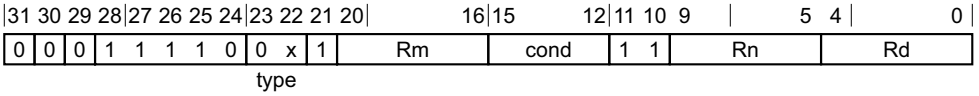
operand2 = if cmp_with_zero then FPZero('0') else V[m];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.56 FCSEL

Floating-point conditional select (scalar) : `Vd = if cond then Vn else Vm`




### *Single-precision variant*

Applies when `type = 00`.

```
FCSEL <Sd>, <Sn>, <Sm>, <cond>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FCSEL <Dd>, <Dn>, <Dm>, <cond>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

bits(4) condition = cond;
```

## Assembler symbols

| | |
|---|---|
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Dn>` | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Dm>` | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Sn>` | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Sm>` | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| `<cond>` | Is one of the standard conditions, encoded in the "cond" field in the standard way. |

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;

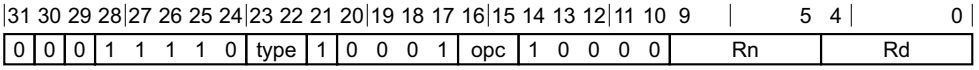
result = if ConditionHolds(condition) then V[n] else V[m];

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.57 FCVT

Floating-point convert precision (scalar) : `Vd = convertFormat(Vn)`




### *Half-precision to single-precision variant*

Applies when `type = 11 && opc = 00`.

```
FCVT <Sd>, <Hn>
```

### *Half-precision to double-precision variant*

Applies when `type = 11 && opc = 01`.

```
FCVT <Dd>, <Hn>
```

### *Single-precision to half-precision variant*

Applies when `type = 00 && opc = 11`.

```
FCVT <Hd>, <Sn>
```

### *Single-precision to double-precision variant*

Applies when `type = 00 && opc = 01`.

```
FCVT <Dd>, <Sn>
```

### *Double-precision to half-precision variant*

Applies when `type = 01 && opc = 11`.

```
FCVT <Hd>, <Dn>
```

### *Double-precision to single-precision variant*

Applies when `type = 01 && opc = 00`.

```
FCVT <Sd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if type == opc then UnallocatedEncoding();

integer srcsize;
case type of
    when '00' srcsize = 32;
    when '01' srcsize = 64;
    when '10' UnallocatedEncoding();
    when '11' srcsize = 16;
integer dstsize;
case opc of
    when '00' dstsize = 32;
    when '01' dstsize = 64;
    when '10' UnallocatedEncoding();
    when '11' dstsize = 16;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();

bits(dstsize) result;
bits(srcsize) operand = V[n];

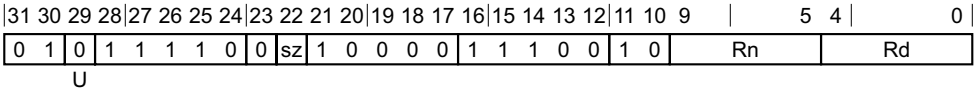
result = FPConvert(operand, FPCR);
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.58 FCVTAS (vector)

Floating-point convert to signed integer, rounding to nearest with ties to away (vector)

### Scalar




#### *Scalar variant*

```
FCVTAS <V><d>, <V><n>
```

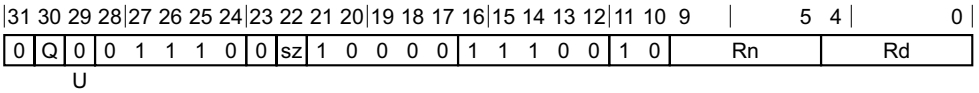
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
FCVTAS <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

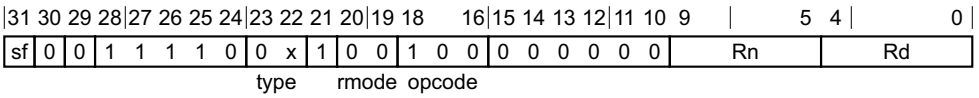
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.59 FCVTAS (scalar)

Floating-point convert to signed integer, rounding to nearest with ties to away (scalar) : `Rd = signed_convertToIntegerExactTiesToAway(Vn)`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTAS <Wd>, <Sn>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTAS <Xd>, <Sn>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTAS <Wd>, <Dn>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTAS <Xd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

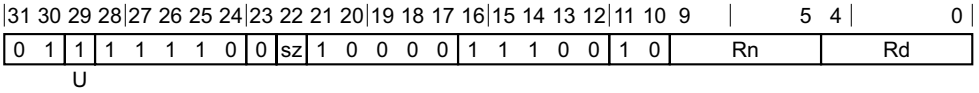
case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.60 FCVTAU (vector)

Floating-point convert to unsigned integer, rounding to nearest with ties to away (vector)

### Scalar




#### *Scalar variant*

```
FCVTAU <V><d>, <V><n>
```

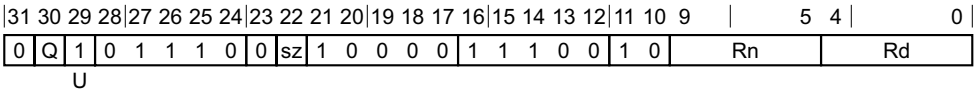
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
FCVTAU <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

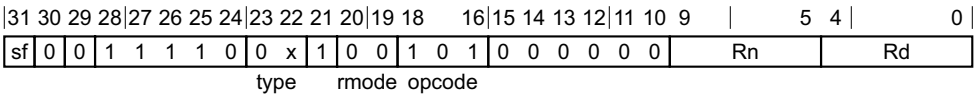
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.61 FCVTAU (scalar)

Floating-point convert to unsigned integer, rounding to nearest with ties to away (scalar) : `Rd = unsigned_convertToIntegerExactTiesToAway(Vn)`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTAU <Wd>, <Sn>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTAU <Xd>, <Sn>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTAU <Wd>, <Dn>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTAU <Xd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

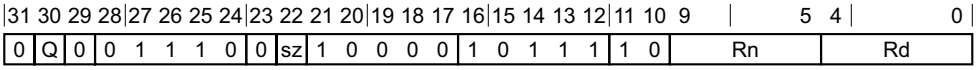
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.62 FCVTL, FCVTL2

Floating-point convert to higher precision long (vector)




### *Vector variant*

```
FCVTL{2} <Vd>.<Ta>, <Vn>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "sz" field. It can have the following values:

- 4S when sz = 0
- 2D when sz = 1

<Vn> — Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 4H when sz = 0, Q = 0
- 8H when sz = 0, Q = 1
- 2S when sz = 1, Q = 0
- 4S when sz = 1, Q = 1

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;

for e = 0 to elements-1
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR);

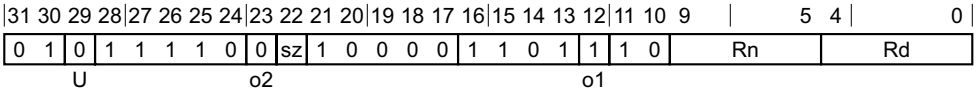
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.63 FCVTMS (vector)

Floating-point convert to signed integer, rounding toward minus infinity (vector)

### Scalar




#### *Scalar variant*

```
FCVTMS <V><d>, <V><n>
```

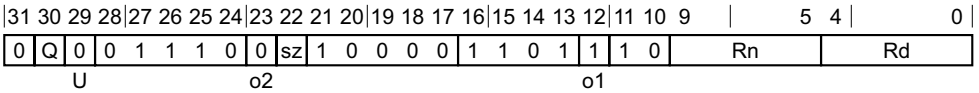
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
FCVTMS <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

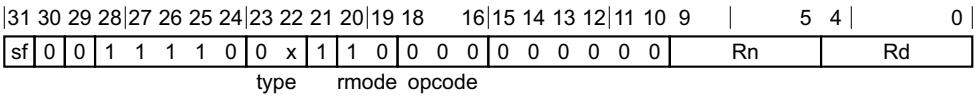
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.64 FCVTMS (scalar)

Floating-point convert to signed integer, rounding toward minus infinity (scalar) : `Rd = signed_convertToIntegerExactTowardNegative(Vn)`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTMS <Wd>, <Sn>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTMS <Xd>, <Sn>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTMS <Wd>, <Dn>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTMS <Xd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

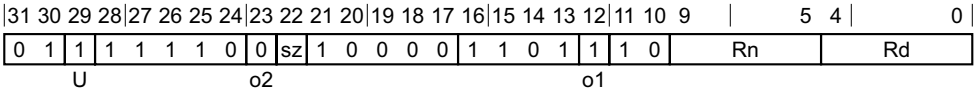
case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.65 FCVTMU (vector)

Floating-point convert to unsigned integer, rounding toward minus infinity (vector)

### Scalar




#### *Scalar variant*

```
FCVTMU <V><d>, <V><n>
```

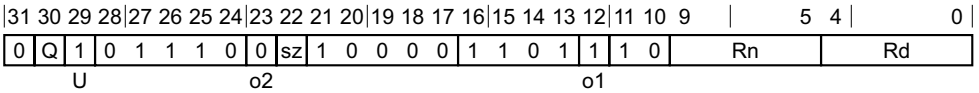
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
FCVTMU <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

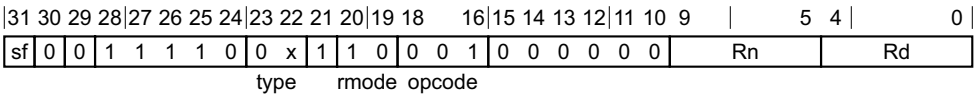
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.66 FCVTMU (scalar)

Floating-point convert to unsigned integer, rounding toward minus infinity (scalar) : `Rd = unsigned_convertToIntegerExactTowardNegative(Vn)`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTMU <Wd>, <Sn>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTMU <Xd>, <Sn>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTMU <Wd>, <Dn>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTMU <Xd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

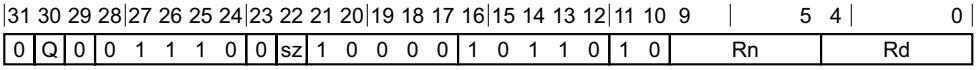
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.67 FCVTN, FCVTN2

Floating-point convert to lower precision narrow (vector)




### *Vector variant*

```
FCVTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> — Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 4H when sz = 0, Q = 0
- 8H when sz = 0, Q = 1
- 2S when sz = 1, Q = 0
- 4S when sz = 1, Q = 1

<Vn> — Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> — Is an arrangement specifier, encoded in the "sz" field. It can have the following values:

- 4S when sz = 0
- 2D when sz = 1

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR);

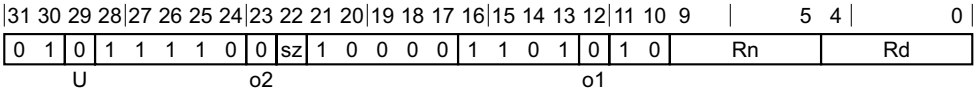
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.68 FCVTNS (vector)

Floating-point convert to signed integer, rounding to nearest with ties to even (vector)

### Scalar




#### *Scalar variant*

```
FCVTNS <V><d>, <V><n>
```

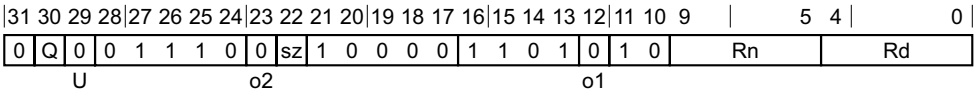
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
FCVTNS <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

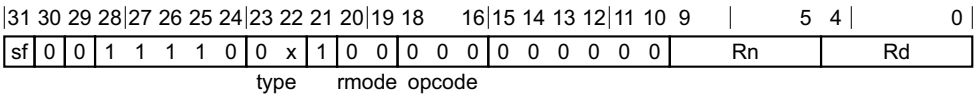
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.69 FCVTNS (scalar)

Floating-point convert to signed integer, rounding to nearest with ties to even (scalar) : `Rd = signed_convertToIntegerExactTiesToEven(Vn)`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTNS <Wd>, <Sn>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTNS <Xd>, <Sn>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTNS <Wd>, <Dn>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTNS <Xd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

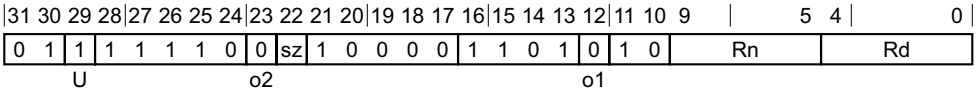
case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.70 FCVTNU (vector)

Floating-point convert to unsigned integer, rounding to nearest with ties to even (vector)

### Scalar




#### *Scalar variant*

```
FCVTNU <V><d>, <V><n>
```

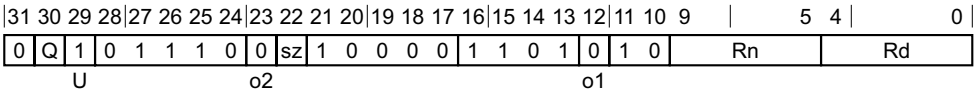
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
FCVTNU <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

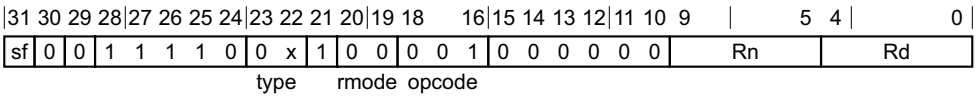
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.71 FCVTNU (scalar)

Floating-point convert to unsigned integer, rounding to nearest with ties to even (scalar) : `Rd = unsigned_convertToIntegerExactTiesToEven(Vn)`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTNU <Wd>, <Sn>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTNU <Xd>, <Sn>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTNU <Wd>, <Dn>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTNU <Xd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

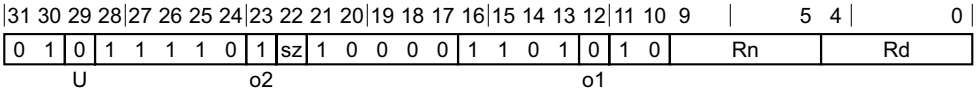
case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.72 FCVTPS (vector)

Floating-point convert to signed integer, rounding toward positive infinity (vector)

### Scalar




#### *Scalar variant*

```
FCVTPS <V><d>, <V><n>
```

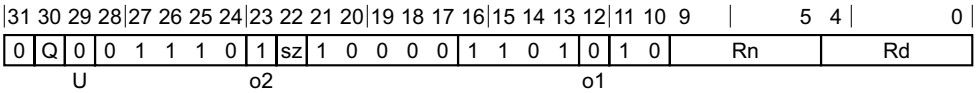
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
FCVTPS <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

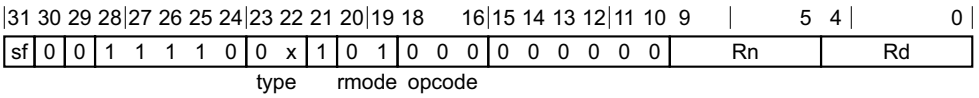
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.73 FCVTPS (scalar)

Floating-point convert to signed integer, rounding toward positive infinity (scalar) : `Rd = signed_convertToIntegerExactTowardPositive(Vn)`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTPS <Wd>, <Sn>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTPS <Xd>, <Sn>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTPS <Wd>, <Dn>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTPS <Xd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

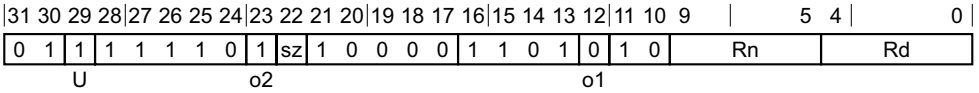
case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.74 FCVTPU (vector)

Floating-point convert to unsigned integer, rounding toward positive infinity (vector)

### Scalar




#### *Scalar variant*

```
FCVTPU <V><d>, <V><n>
```

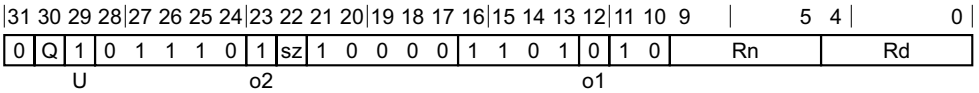
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
FCVTPU <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

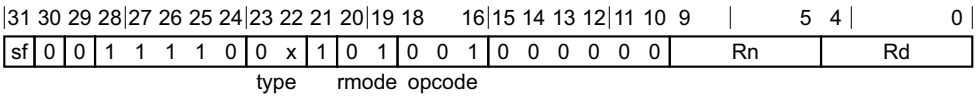
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.75 FCVTPU (scalar)

Floating-point convert to unsigned integer, rounding toward positive infinity (scalar) : `Rd = unsigned_convertToIntegerExactTowardPositive(Vn)`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTPU <Wd>, <Sn>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTPU <Xd>, <Sn>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTPU <Wd>, <Dn>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTPU <Xd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

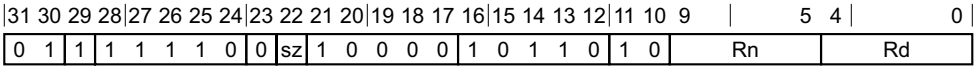
case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.76 FCVTXN, FCVTXN2

Floating-point convert to lower precision narrow, rounding to odd (vector)

### Scalar




#### *Scalar variant*

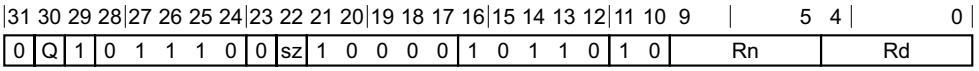
```
FCVTXN <Vb><d>, <Va><n>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then ReservedValue();
integer esize = 32;
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

### Vector




#### *Vector variant*

```
FCVTXN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then ReservedValue();
integer esize = 32;
integer datasize = 64;
integer elements = 2;
integer part = UInt(Q);
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 1, Q = 0


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4S when sz = 1, Q = 1

The encoding sz = 0, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "sz" field. It can have the following values:

2D when sz = 1

The encoding sz = 0 is reserved.

<Vb> Is the destination width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 1

The encoding sz = 0 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in the "sz" field. It can have the following values:

D when sz = 1

The encoding sz = 0 is reserved.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR, FPRounding_ODD);

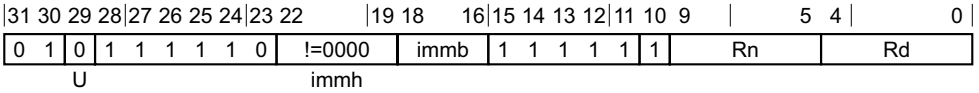
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.77 FCVTZS (vector, fixed-point)

Floating-point convert to signed fixed-point, rounding toward zero (vector)

### Scalar




#### *Scalar variant*

```
FCVTZS <V><d>, <V><n>, #<fbits>
```

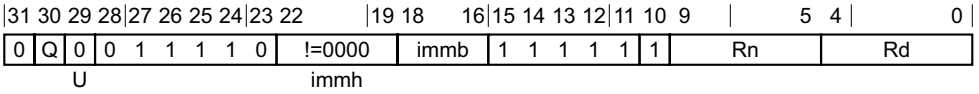
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '00xx' then ReservedValue();
integer esize = 32 << UInt(immh<3>);
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Vector




#### *Vector variant*

```
FCVTZS <Vd>.<T>, <Vn>.<T>, #<fbits>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh == '00xx' then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = 32 << UInt(immh<3>);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

S when `immh = 01xx`

D when `immh = 1xxx`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `immh = 00xx` is reserved.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

`2D` when `immh = 1xxx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`, `Q = x`.

The following encodings are reserved:

- `immh = 0001`, `Q = x`.
- `immh = 001x`, `Q = x`.
- `immh = 1xxx`, `Q = 0`.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<fbits>` For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immh:immb" field. It can have the following values:

`(64-UInt(immh:immb))` when `immh = 01xx`

`(128-UInt(immh:immb))` when `immh = 1xxx`

The encoding `immh = 00xx` is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immh:immb" field. It can have the following values:

`(64-UInt(immh:immb))` when `immh = 01xx`

`(128-UInt(immh:immb))` when `immh = 1xxx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The following encodings are reserved:

- `immh = 0001`.
- `immh = 001x`.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);

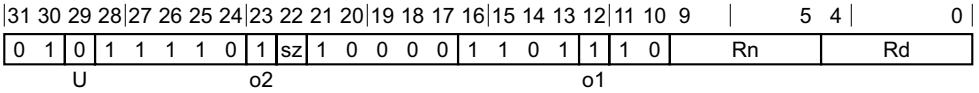
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.78 FCVTZS (vector, integer)

Floating-point convert to signed integer, rounding toward zero (vector)

### Scalar




#### *Scalar variant*

```
FCVTZS <V><d>, <V><n>
```

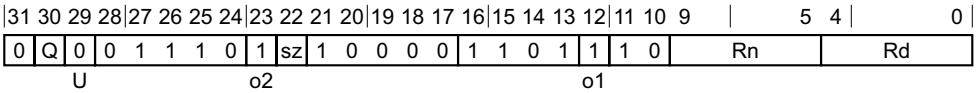
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
FCVTZS <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

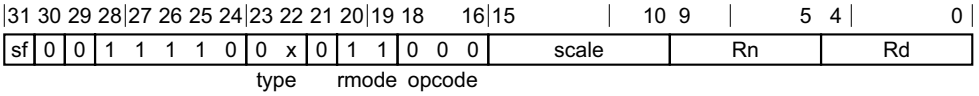
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.79 FCVTZS (scalar, fixed-point)

Floating-point convert to signed fixed-point, rounding toward zero (scalar) : `Rd = signed_convertToIntegerExactTowardZero(Vn*(2^fbits))`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTZS <Wd>, <Sn>, #<fbits>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTZS <Xd>, <Sn>, #<fbits>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTZS <Wd>, <Dn>, #<fbits>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTZS <Xd>, <Dn>, #<fbits>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '1x' UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
    when '00 11'        // FCVTZ
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    otherwise
        UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <fbits> | For the double-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".<br>For the double-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale". |

### Operation

```
CheckFPAdvSIMDEnabled64();

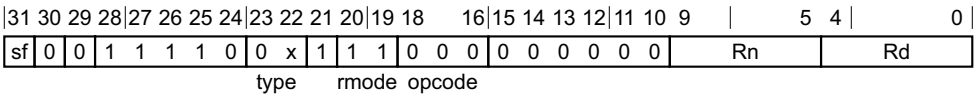
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
        V[d] = fltval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.80 FCVTZS (scalar, integer)

Floating-point convert to signed integer, rounding toward zero (scalar) : `Rd = signed_convertToIntegerExactTowardZero(Vn)`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTZS <Wd>, <Sn>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTZS <Xd>, <Sn>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTZS <Wd>, <Dn>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTZS <Xd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

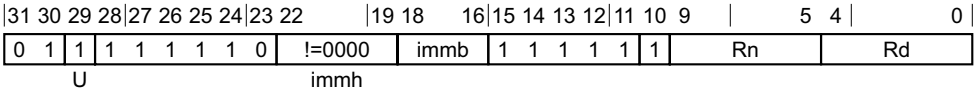
case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.81 FCVTZU (vector, fixed-point)

Floating-point convert to unsigned fixed-point, rounding toward zero (vector)

### Scalar




#### *Scalar variant*

```
FCVTZU <V><d>, <V><n>, #<fbits>
```

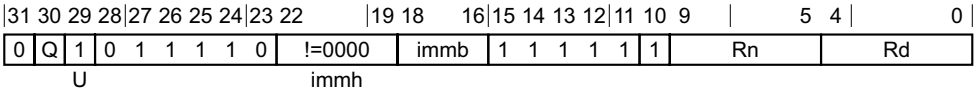
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '00xx' then ReservedValue();
integer esize = 32 << UInt(immh<3>);
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Vector




#### *Vector variant*

```
FCVTZU <Vd>.<T>, <Vn>.<T>, #<fbits>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh == '00xx' then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = 32 << UInt(immh<3>);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

S when immh = 01xx

D when immh = 1xxx


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding immh = 00xx is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The following encodings are reserved:

- immh = 0001, Q = x.
- immh = 001x, Q = x.
- immh = 1xxx, Q = 0.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immh:immb" field. It can have the following values:

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 00xx is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immh:immb" field. It can have the following values:

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

The following encodings are reserved:

- immh = 0001.
- immh = 001x.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);

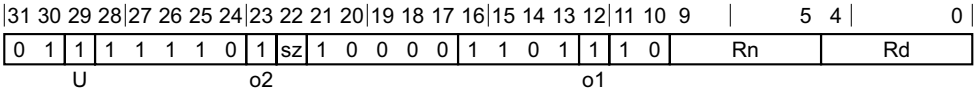
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.82 FCVTZU (vector, integer)

Floating-point convert to unsigned integer, rounding toward zero (vector)

### Scalar




#### *Scalar variant*

```
FCVTZU <V><d>, <V><n>
```

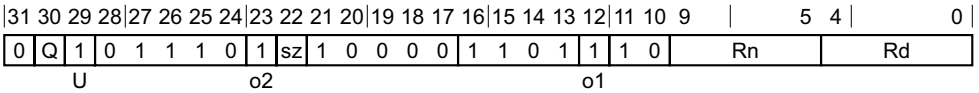
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
FCVTZU <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

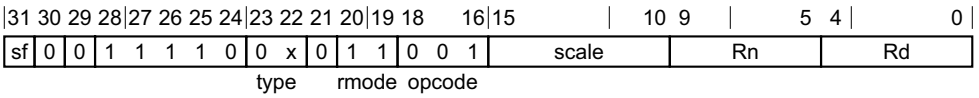
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.83 FCVTZU (scalar, fixed-point)

Floating-point convert to unsigned fixed-point, rounding toward zero (scalar) : `Rd = unsigned_convertToIntegerExactTowardZero(Vn*(2^fbits))`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTZU <Wd>, <Sn>, #<fbits>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTZU <Xd>, <Sn>, #<fbits>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTZU <Wd>, <Dn>, #<fbits>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTZU <Xd>, <Dn>, #<fbits>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '1x' UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
    when '00 11'        // FCVTZ
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    otherwise
        UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the double-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".

For the double-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

### Operation

```
CheckFPAdvSIMDEnabled64();

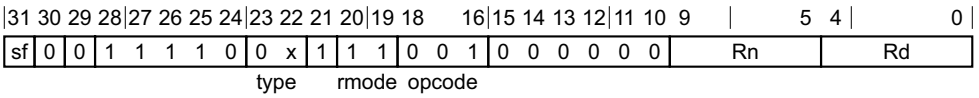
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
        V[d] = fltval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.84 FCVTZU (scalar, integer)

Floating-point convert to unsigned integer, rounding toward zero (scalar) : `Rd = unsigned_convertToIntegerExactTowardZero(Vn)`




### *Single-precision to 32-bit variant*

Applies when `sf = 0 && type = 00`.

```
FCVTZU <Wd>, <Sn>
```

### *Single-precision to 64-bit variant*

Applies when `sf = 1 && type = 00`.

```
FCVTZU <Xd>, <Sn>
```

### *Double-precision to 32-bit variant*

Applies when `sf = 0 && type = 01`.

```
FCVTZU <Wd>, <Dn>
```

### *Double-precision to 64-bit variant*

Applies when `sf = 1 && type = 01`.

```
FCVTZU <Xd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

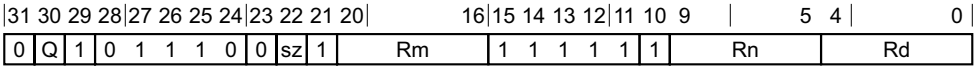
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.85 FDIV (vector)

Floating-point divide (vector)




### *Three registers of the same type variant*

```
FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

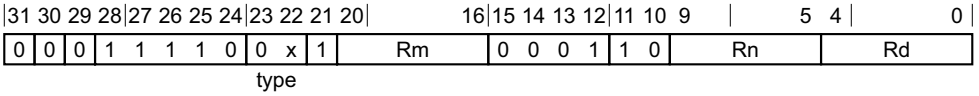
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.86 FDIV (scalar)

Floating-point divide (scalar) : `Vd = Vn / Vm`




### *Single-precision variant*

Applies when `type = 00`.

```
FDIV <Sd>, <Sn>, <Sm>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FDIV <Dd>, <Dn>, <Dm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();
```

## Assembler symbols

| | |
|---|---|
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Dn>` | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Dm>` | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Sn>` | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Sm>` | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

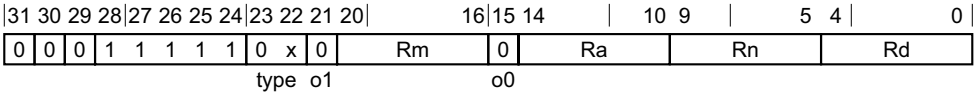
result = FPDiv(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.87 FMADD

Floating-point fused multiply-add (scalar) : `Vd = Va + Vn*Vm`




### *Single-precision variant*

Applies when `type = 00`.

```
FMADD <Sd>, <Sn>, <Sm>, <Sa>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FMADD <Dd>, <Dn>, <Dm>, <Da>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean opa_neg = (o1 == '1');
boolean op1_neg = (o0 != o1);
```

### Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Da>` Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Sa>` Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

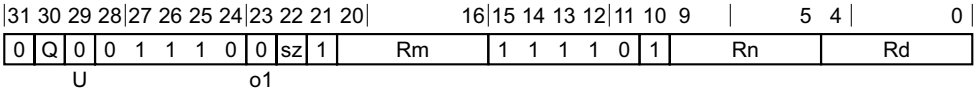
if opa_neg then operanda = FPNeg(operanda);
if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.88 FMAX (vector)

Floating-point maximum (vector)




### *Three registers of the same type variant*

```
FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

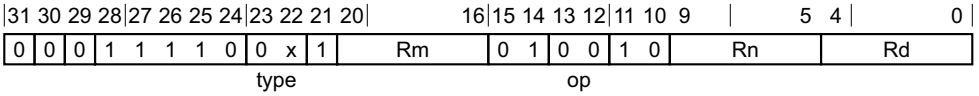
```
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.89 FMAX (scalar)

Floating-point maximum (scalar) : `Vd = max(Vn, Vm)`




### *Single-precision variant*

Applies when `type = 00`.

```
FMAX <Sd>, <Sn>, <Sm>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FMAX <Dd>, <Dn>, <Dm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

FPMaxMinOp operation;
case op of
    when '00' operation = FPMaxMinOp_MAX;
    when '01' operation = FPMaxMinOp_MIN;
    when '10' operation = FPMaxMinOp_MAXNUM;
    when '11' operation = FPMaxMinOp_MINNUM;
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

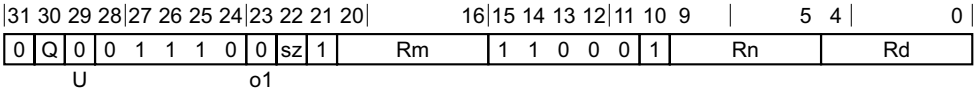
```
    when FPMaxMinOp_MAX    result = FPMax(operand1, operand2, FPCR);
    when FPMaxMinOp_MIN    result = FPMin(operand1, operand2, FPCR);
    when FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
    when FPMaxMinOp_MINNUM result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.90 FMAXNM (vector)

Floating-point maximum number (vector)




### *Three registers of the same type variant*

```
FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

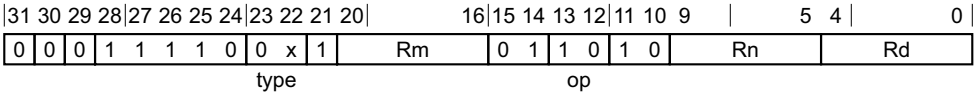
```
        else
            Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.91 FMAXNM (scalar)

Floating-point maximum number (scalar) : `Vd = maxNum(Vn, Vm)`




### *Single-precision variant*

Applies when `type = 00`.

```
FMAXNM <Sd>, <Sn>, <Sm>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FMAXNM <Dd>, <Dn>, <Dm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

FPMaxMinOp operation;
case op of
    when '00' operation = FPMaxMinOp_MAX;
    when '01' operation = FPMaxMinOp_MIN;
    when '10' operation = FPMaxMinOp_MAXNUM;
    when '11' operation = FPMaxMinOp_MINNUM;
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

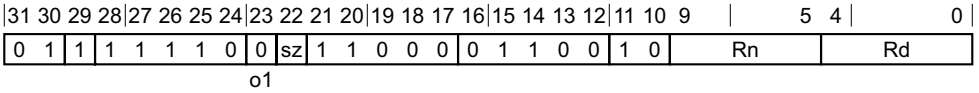
```
    when FPMaxMinOp_MAX    result = FPMax(operand1, operand2, FPCR);
    when FPMaxMinOp_MIN    result = FPMin(operand1, operand2, FPCR);
    when FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
    when FPMaxMinOp_MINNUM result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.92 FMAXNMP (scalar)

Floating-point maximum number of pair of elements (scalar)




### *Advanced SIMD variant*

```
FMAXNMP <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

### Assembler symbols

<V> Is the destination width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- 2S when sz = 0
- 2D when sz = 1

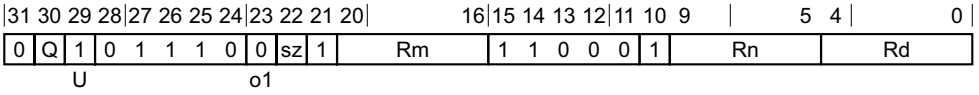
### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.93 FMAXNMP (vector)

Floating-point maximum number pairwise (vector)




### *Three registers of the same type variant*

```
FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

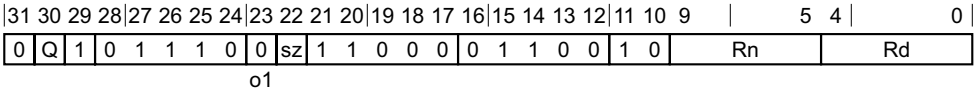
```
        else
            Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.94 FMAXNMV

Floating-point maximum number across vector




### *Advanced SIMD variant*

```
FMAXNMV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then ReservedValue();   // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

## Assembler symbols

<V> Is the destination width specifier, encoded in the "sz" field. It can have the following values:

S when `sz = 0`

The encoding `sz = 1` is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

4S when `Q = 1, sz = 0`

The following encodings are reserved:

- `Q = 0, sz = x`.
- `Q = 1, sz = 1`.

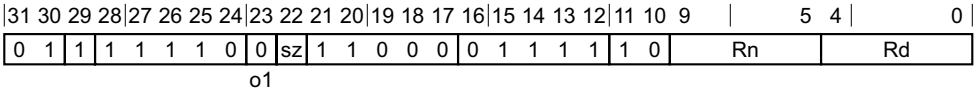
## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.95 FMAXP (scalar)

Floating-point maximum of pair of elements (scalar)




### *Advanced SIMD variant*

```
FMAXP <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

## Assembler symbols

<V> Is the destination width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- 2S when sz = 0
- 2D when sz = 1

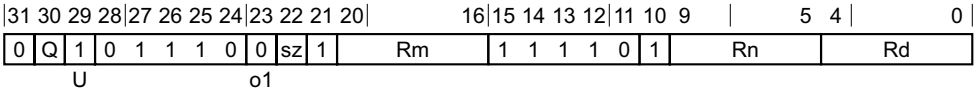
## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.96 FMAXP (vector)

Floating-point maximum pairwise (vector)




### *Three registers of the same type variant*

```
FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

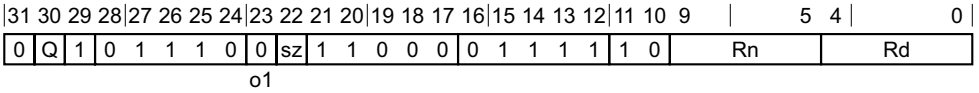
```
        else
            Elem[result, e, esize] = FPMax(element1, element2, FPCR);

    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.97 FMAXV

Floating-point maximum across vector




### *Advanced SIMD variant*

```
FMAXV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then ReservedValue();

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

### Assembler symbols

<V> Is the destination width specifier, encoded in the "sz" field. It can have the following values:

S when `sz = 0`

The encoding `sz = 1` is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

4S when `Q = 1, sz = 0`

The following encodings are reserved:

- `Q = 0, sz = x`.
- `Q = 1, sz = 1`.

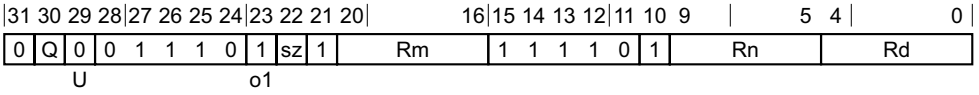
### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.98 FMIN (vector)

Floating-point minimum (vector)




### *Three registers of the same type variant*

```
FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

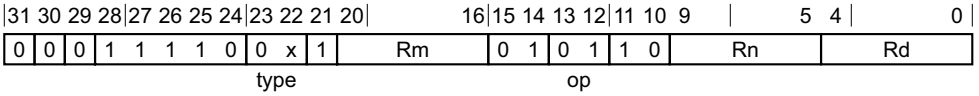
```
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.99 FMIN (scalar)

Floating-point minimum (scalar) : `Vd = min(Vn, Vm)`




### *Single-precision variant*

Applies when `type = 00`.

```
FMIN <Sd>, <Sn>, <Sm>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FMIN <Dd>, <Dn>, <Dm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

FPMaxMinOp operation;
case op of
    when '00' operation = FPMaxMinOp_MAX;
    when '01' operation = FPMaxMinOp_MIN;
    when '10' operation = FPMaxMinOp_MAXNUM;
    when '11' operation = FPMaxMinOp_MINNUM;
```

## Assembler symbols

| | |
|---|---|
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Dn>` | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Dm>` | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Sn>` | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Sm>` | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

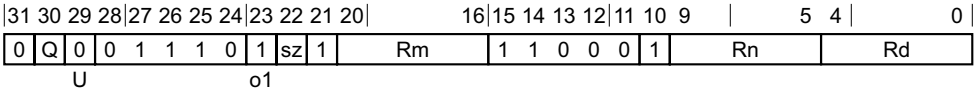
```
    when FPMaxMinOp_MAX    result = FPMax(operand1, operand2, FPCR);
    when FPMaxMinOp_MIN    result = FPMin(operand1, operand2, FPCR);
    when FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
    when FPMaxMinOp_MINNUM result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.100 FMINNM (vector)

Floating-point minimum number (vector)




### *Three registers of the same type variant*

```
FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

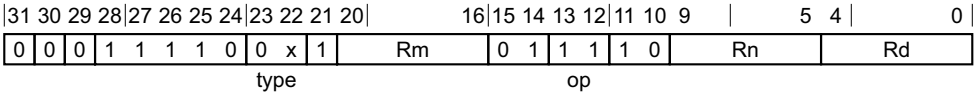
```
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.101 FMINNM (scalar)

Floating-point minimum number (scalar) : `Vd = minNum(Vn, Vm)`




### *Single-precision variant*

Applies when `type = 00`.

```
FMINNM <Sd>, <Sn>, <Sm>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FMINNM <Dd>, <Dn>, <Dm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

FPMaxMinOp operation;
case op of
    when '00' operation = FPMaxMinOp_MAX;
    when '01' operation = FPMaxMinOp_MIN;
    when '10' operation = FPMaxMinOp_MAXNUM;
    when '11' operation = FPMaxMinOp_MINNUM;
```

## Assembler symbols

| | |
|---|---|
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Dn>` | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Dm>` | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Sn>` | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Sm>` | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

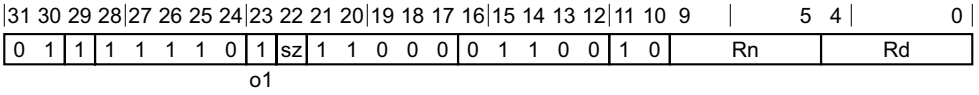
```
    when FPMaxMinOp_MAX    result = FPMax(operand1, operand2, FPCR);
    when FPMaxMinOp_MIN    result = FPMin(operand1, operand2, FPCR);
    when FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
    when FPMaxMinOp_MINNUM result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.102 FMINNMP (scalar)

Floating-point minimum number of pair of elements (scalar)




### *Advanced SIMD variant*

```
FMINNMP <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

### Assembler symbols

<V> Is the destination width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- 2S when sz = 0
- 2D when sz = 1

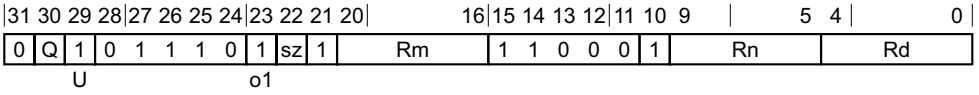
### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.103 FMINNMP (vector)

Floating-point minimum number pairwise (vector)




### *Three registers of the same type variant*

```
FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

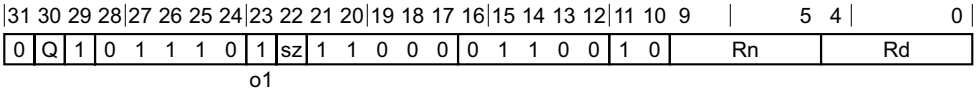
```
        else
            Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.104 FMINNMV

Floating-point minimum number across vector




### *Advanced SIMD variant*

```
FMINNMV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then ReservedValue();   // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

### Assembler symbols

<V> Is the destination width specifier, encoded in the "sz" field. It can have the following values:

S when `sz = 0`

The encoding `sz = 1` is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

4S when `Q = 1, sz = 0`

The following encodings are reserved:

- `Q = 0, sz = x`.
- `Q = 1, sz = 1`.

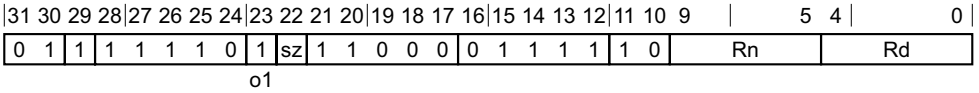
### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.105 FMINP (scalar)

Floating-point minimum of pair of elements (scalar)




### *Advanced SIMD variant*

```
FMINP <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

## Assembler symbols

<V> Is the destination width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- 2S when sz = 0
- 2D when sz = 1

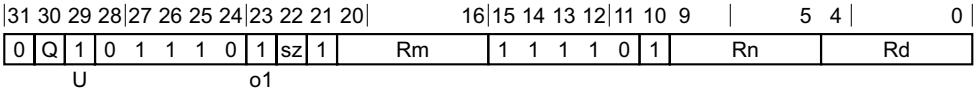
## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.106 FMINP (vector)

Floating-point minimum pairwise (vector)




### *Three registers of the same type variant*

```
FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

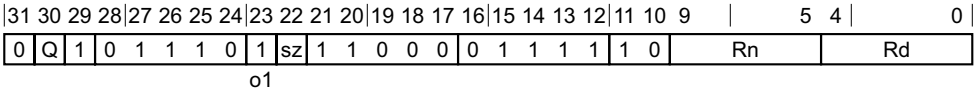
```
        else
            Elem[result, e, esize] = FPMax(element1, element2, FPCR);

    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.107 FMINV

Floating-point minimum across vector




### *Advanced SIMD variant*

```
FMINV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then ReservedValue();

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

### Assembler symbols

<V> Is the destination width specifier, encoded in the "sz" field. It can have the following values:

S when `sz = 0`

The encoding `sz = 1` is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

4S when `Q = 1, sz = 0`

The following encodings are reserved:

- `Q = 0, sz = x`.
- `Q = 1, sz = 1`.

### Operation

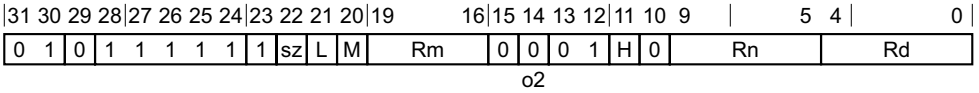
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.108 FMLA (by element)

Floating-point fused multiply-add to accumulator (by element)

### Scalar




#### *Scalar variant*

```
FMLA <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

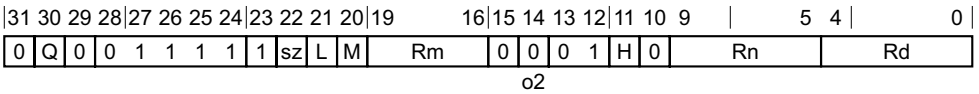
#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

### Vector




#### *Vector variant*

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

2S when Q = 0, sz = 0

4S when Q = 1, sz = 0

2D when Q = 1, sz = 1

The encoding Q = 0, sz = 1 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the scalar variant: is the name of the SIMD&FP source register, encoded in the "M:Rm" fields.

For the vector variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> For the scalar variant: is the element width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

For the vector variant: is an element size specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<index> Is the element index, encoded in the "sz:L:H" field. It can have the following values:

H:L when sz = 0, L = x

H when sz = 1, L = 0

The encoding sz = 1, L = 1 is reserved.

### Operation for all encodings

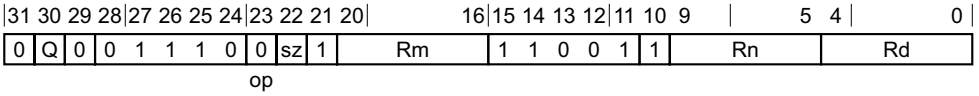
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.109 FMLA (vector)

Floating-point fused multiply-add to accumulator (vector)




### *Three registers of the same type variant*

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

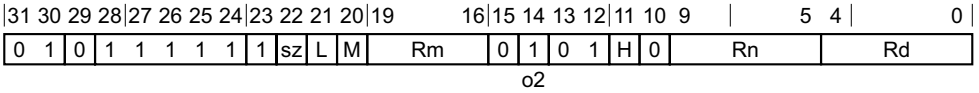
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.110 FMLS (by element)

Floating-point fused multiply-subtract from accumulator (by element)

### Scalar




#### *Scalar variant*

```
FMLS <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

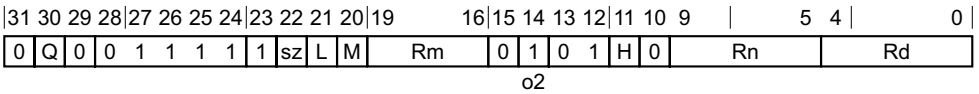
#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

### Vector




#### *Vector variant*

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

2S when Q = 0, sz = 0

4S when Q = 1, sz = 0

2D when Q = 1, sz = 1

The encoding Q = 0, sz = 1 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the scalar variant: is the name of the SIMD&FP source register, encoded in the "M:Rm" fields.

For the vector variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> For the scalar variant: is the element width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

For the vector variant: is an element size specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<index> Is the element index, encoded in the "sz:L:H" field. It can have the following values:

H:L when sz = 0, L = x

H when sz = 1, L = 0

The encoding sz = 1, L = 1 is reserved.

### Operation for all encodings

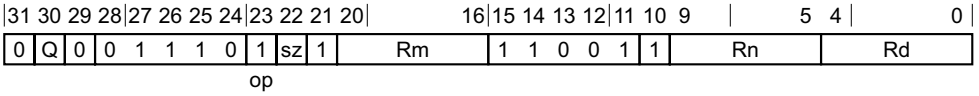
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.111 FMLS (vector)

Floating-point fused multiply-subtract from accumulator (vector)




### *Three registers of the same type variant*

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

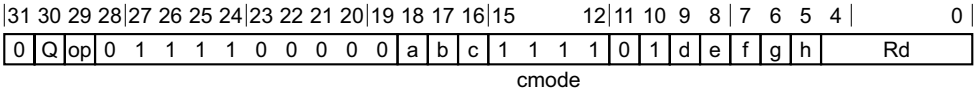
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.112 FMOV (vector, immediate)

Floating-point move immediate (vector)




### *Single-precision variant*

Applies when `op = 0`.

```
FMOV <Vd>.<T>, #<imm>
```

### *Double-precision variant*

Applies when `Q = 1 && op = 1`.

```
FMOV <Vd>.2D, #<imm>
```

### *Decode for all variants of this encoding*

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `2S` when `Q = 0`
- `4S` when `Q = 1`

<imm> Is a floating-point constant with sign, 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h".


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

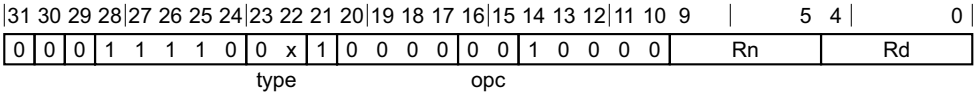
case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.113 FMOV (register)

Floating-point move register without conversion : `Vd = Vn`




### *Single-precision variant*

Applies when `type = 00`.

```
FMOV <Sd>, <Sn>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FMOV <Dd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

FPUnaryOp fpop;
case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_NEG;
    when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
    when FPUnaryOp_MOV  result = operand;
    when FPUnaryOp_ABS  result = FPAbs(operand);
    when FPUnaryOp_NEG  result = FPNeg(operand);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

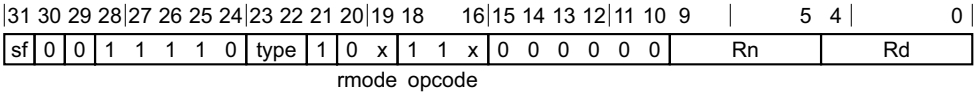
```
        when FPUnaryOp_SQRT result = FPSqrt(operand, FPCR);

    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.114 FMOV (general)

Floating-point move to or from general-purpose register without conversion




### 32-bit to single-precision variant

Applies when `sf = 0 && type = 00 && rmode = 00 && opcode = 111`.

```
FMOV <Sd>, <Wn>
```

### Single-precision to 32-bit variant

Applies when `sf = 0 && type = 00 && rmode = 00 && opcode = 110`.

```
FMOV <Wd>, <Sn>
```

### 64-bit to double-precision variant

Applies when `sf = 1 && type = 01 && rmode = 00 && opcode = 111`.

```
FMOV <Dd>, <Xn>
```

### 64-bit to top half of 128-bit variant

Applies when `sf = 1 && type = 10 && rmode = 01 && opcode = 111`.

```
FMOV <Vd>.D[1], <Xn>
```

### Double-precision to 64-bit variant

Applies when `sf = 1 && type = 01 && rmode = 00 && opcode = 110`.

```
FMOV <Xd>, <Dn>
```

### Top half of 128-bit to 64-bit variant

Applies when `sf = 1 && type = 10 && rmode = 01 && opcode = 110`.

```
FMOV <Xd>, <Vn>.D[1]
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.115 FMOV (scalar, immediate)

Floating-point move immediate (scalar) : `Vd=#imm`




### *Single-precision variant*

Applies when `type = 00`.

```
FMOV <Sd>, #<imm>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FMOV <Dd>, #<imm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

bits(datasize) imm = VFPExpandImm(imm8);
```

## Assembler symbols

| | |
|---|---|
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<imm>` | Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. |

## Operation

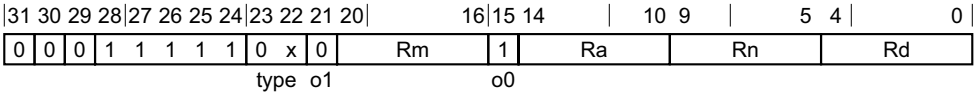
```
CheckFPAdvSIMDEnabled64();

V[d] = imm;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.116 FMSUB

Floating-point fused multiply-subtract (scalar) : `Vd = Va + (-Vn)*Vm`




### *Single-precision variant*

Applies when `type = 00`.

```
FMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FMSUB <Dd>, <Dn>, <Dm>, <Da>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean opa_neg = (o1 == '1');
boolean op1_neg = (o0 != o1);
```

### Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Da>` Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Sa>` Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if opa_neg then operanda = FPNeg(operanda);
if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

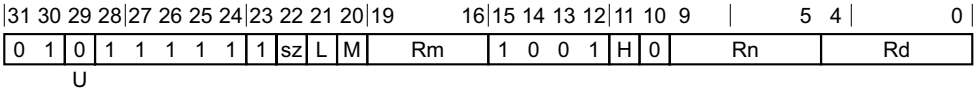
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.117 FMUL (by element)

Floating-point multiply (by element)

### Scalar




#### *Scalar variant*

```
FMUL <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

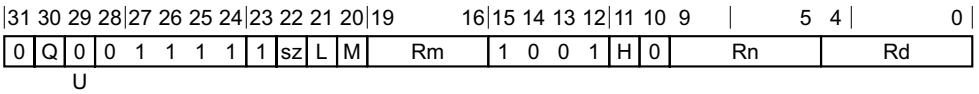
#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

### Vector




#### *Vector variant*

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

2S when Q = 0, sz = 0

4S when Q = 1, sz = 0

2D when Q = 1, sz = 1

The encoding Q = 0, sz = 1 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the scalar variant: is the name of the SIMD&FP source register, encoded in the "M:Rm" fields.

For the vector variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> For the scalar variant: is the element width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

For the vector variant: is an element size specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<index> Is the element index, encoded in the "sz:L:H" field. It can have the following values:

H:L when sz = 0, L = x

H when sz = 1, L = 0

The encoding sz = 1, L = 1 is reserved.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

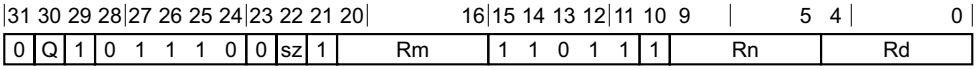
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.118 FMUL (vector)

Floating-point multiply (vector)




### *Three registers of the same type variant*

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

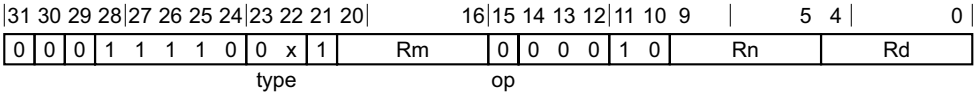
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.119 FMUL (scalar)

Floating-point multiply (scalar) : `Vd = Vn * Vm`




### *Single-precision variant*

Applies when `type = 00`.

```
FMUL <Sd>, <Sn>, <Sm>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FMUL <Dd>, <Dn>, <Dm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean negated = (op == '1');
```

## Assembler symbols

| | |
|---|---|
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Dn>` | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Dm>` | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Sn>` | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Sm>` | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);

if negated then result = FPNeg(result);

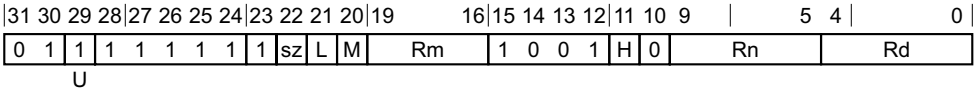
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.120 FMULX (by element)

Floating-point multiply extended (by element)

### Scalar




#### *Scalar variant*

```
FMULX <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

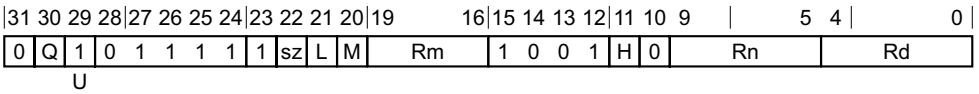
#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

### Vector




#### *Vector variant*

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

2S when Q = 0, sz = 0

4S when Q = 1, sz = 0

2D when Q = 1, sz = 1

The encoding Q = 0, sz = 1 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the scalar variant: is the name of the SIMD&FP source register, encoded in the "M:Rm" fields.

For the vector variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> For the scalar variant: is the element width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

For the vector variant: is an element size specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<index> Is the element index, encoded in the "sz:L:H" field. It can have the following values:

H:L when sz = 0, L = x

H when sz = 1, L = 0

The encoding sz = 1, L = 1 is reserved.

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMul(element1, element2, FPCR);

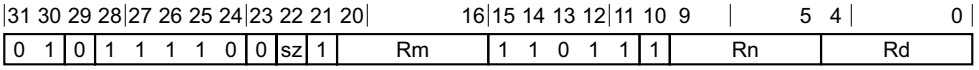
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.121 FMULX

Floating-point multiply extended

### Scalar



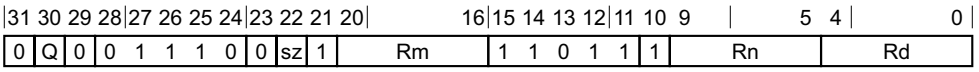

#### *Scalar variant*

```
FMULX <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector




#### *Vector variant*

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

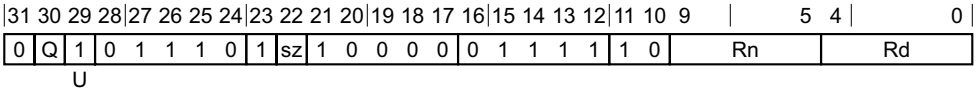
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.122 FNEG (vector)

Floating-point negate (vector)




### *Vector variant*

```
FNEG <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean neg = (U == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

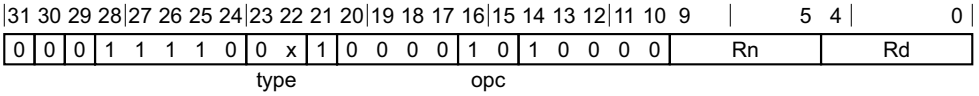
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.123 FNEG (scalar)

Floating-point negate (scalar) : `Vd = -Vn`




### *Single-precision variant*

Applies when `type = 00`.

```
FNEG <Sd>, <Sn>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FNEG <Dd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

FPUnaryOp fpop;
case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_NEG;
    when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
    when FPUnaryOp_MOV  result = operand;
    when FPUnaryOp_ABS  result = FPAbs(operand);
    when FPUnaryOp_NEG  result = FPNeg(operand);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

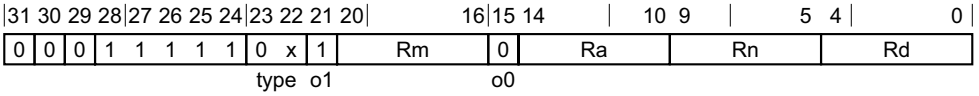
```
            when FPUnaryOp_SQRT result = FPSqrt(operand, FPCR);

    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.124 FNMADD

Floating-point negated fused multiply-add (scalar) : `Vd = (-Va) + (-Vn)*Vm`




### *Single-precision variant*

Applies when `type = 00`.

```
FNMADD <Sd>, <Sn>, <Sm>, <Sa>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FNMADD <Dd>, <Dn>, <Dm>, <Da>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean opa_neg = (o1 == '1');
boolean op1_neg = (o0 != o1);
```

### Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Da>` Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Sa>` Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

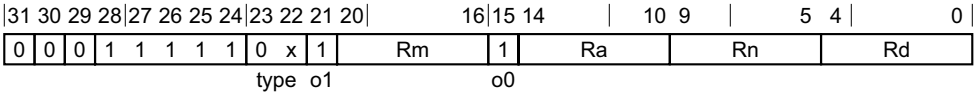
if opa_neg then operanda = FPNeg(operanda);
if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.125 FNMSUB

Floating-point negated fused multiply-subtract (scalar) : `Vd = (-Va) + Vn*Vm`




### *Single-precision variant*

Applies when `type = 00`.

```
FNMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FNMSUB <Dd>, <Dn>, <Dm>, <Da>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean opa_neg = (o1 == '1');
boolean op1_neg = (o0 != o1);
```

## Assembler symbols

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Da>` Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Sn>` Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

`<Sm>` Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

`<Sa>` Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

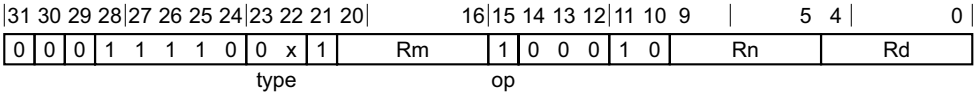
if opa_neg then operanda = FPNeg(operanda);
if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.126 FNMUL

Floating-point multiply-negate (scalar) : `Vd = -(Vn * Vm)`




### *Single-precision variant*

Applies when `type = 00`.

```
FNMUL <Sd>, <Sn>, <Sm>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FNMUL <Dd>, <Dn>, <Dm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean negated = (op == '1');
```

## Assembler symbols

| | |
|---|---|
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Dn>` | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Dm>` | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Sn>` | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Sm>` | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);

if negated then result = FPNeg(result);

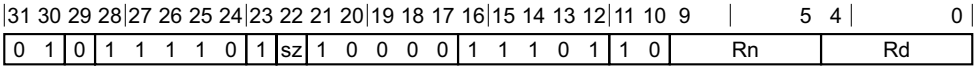
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.127 FRECPE

Floating-point reciprocal estimate

### Scalar




#### *Scalar variant*

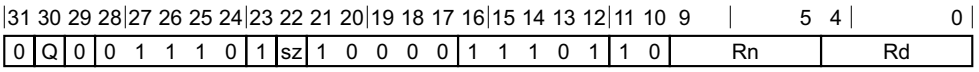
```
FRECPE <V><d>, <V><n>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector




#### *Vector variant*

```
FRECPE <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when `sz = 0`

D when `sz = 1`

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when `sz = 0, Q = 0`

4S when `sz = 0, Q = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR);

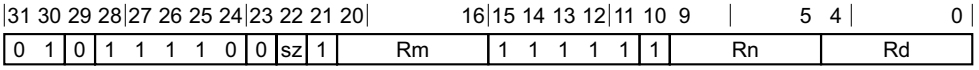
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.128 FRECPS

Floating-point reciprocal step

### Scalar



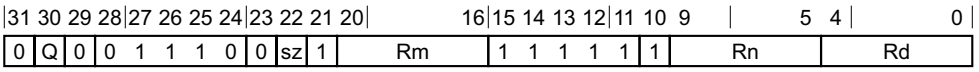

#### *Scalar variant*

```
FRECPS <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector




#### *Vector variant*

```
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

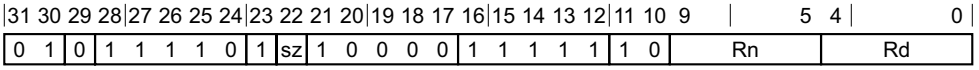
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.129 FRECPX

Floating-point reciprocal exponent (scalar)




### *Scalar variant*

```
FRECPX <V><d>, <V><n>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

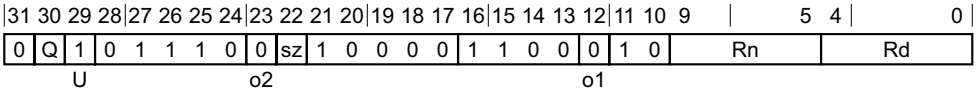
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecpX(element, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.130 FRINTA (vector)

Floating-point round to integral, to nearest with ties to away (vector)




### *Vector variant*

```
FRINTA <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when `sz = 0, Q = 0`
- 4S when `sz = 0, Q = 1`
- 2D when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

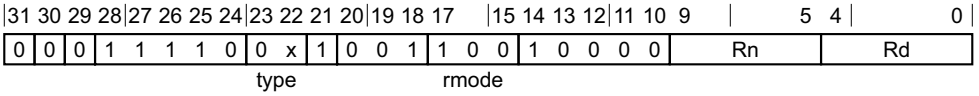
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.131 FRINTA (scalar)

Floating-point round to integral, to nearest with ties to away (scalar) : `Vd = roundToIntegralTiesToAway(Vn)`




### *Single-precision variant*

Applies when `type = 00`.

```
FRINTA <Sd>, <Sn>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FRINTA <Dd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

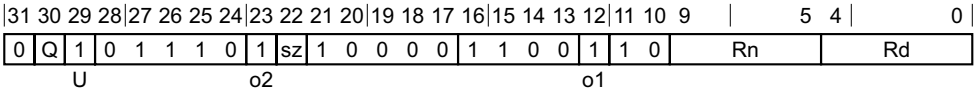
result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.132 FRINTI (vector)

Floating-point round to integral, using current rounding mode (vector)




### *Vector variant*

```
FRINTI <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when `sz = 0, Q = 0`

4S when `sz = 0, Q = 1`

2D when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

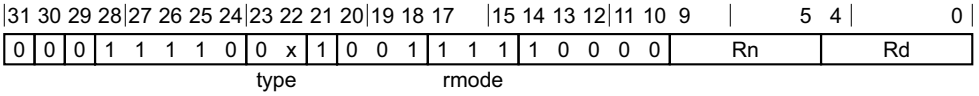
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.133 FRINTI (scalar)

Floating-point round to integral, using current rounding mode (scalar) : `Vd = roundToIntegral(Vn)`




### *Single-precision variant*

Applies when `type = 00`.

```
FRINTI <Sd>, <Sn>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FRINTI <Dd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

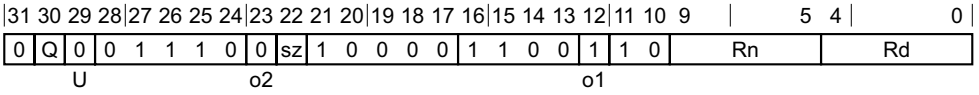
result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.134 FRINTM (vector)

Floating-point round to integral, toward minus infinity (vector)




### *Vector variant*

```
FRINTM <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when `sz = 0, Q = 0`
- 4S when `sz = 0, Q = 1`
- 2D when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

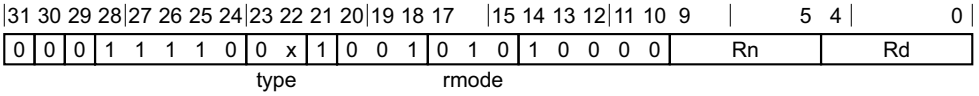
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.135 FRINTM (scalar)

Floating-point round to integral, toward minus infinity (scalar) : `Vd = roundToIntegralTowardNegative(Vn)`




### *Single-precision variant*

Applies when `type = 00`.

```
FRINTM <Sd>, <Sn>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FRINTM <Dd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

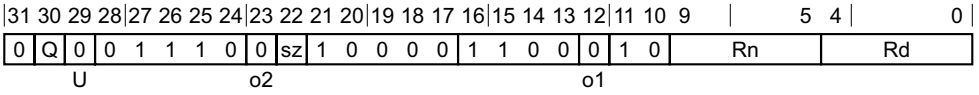
result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.136 FRINTN (vector)

Floating-point round to integral, to nearest with ties to even (vector)




### *Vector variant*

```
FRINTN <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when `sz = 0, Q = 0`
- 4S when `sz = 0, Q = 1`
- 2D when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

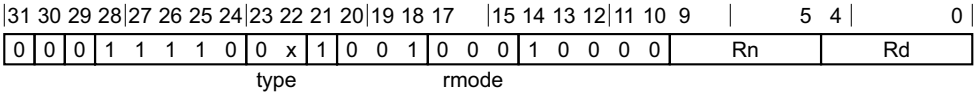
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.137 FRINTN (scalar)

Floating-point round to integral, to nearest with ties to even (scalar) : `Vd = roundToIntegralTiesToEven(Vn)`




### *Single-precision variant*

Applies when `type = 00`.

```
FRINTN <Sd>, <Sn>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FRINTN <Dd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

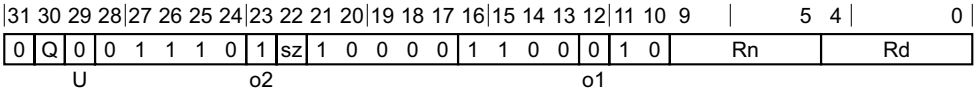
result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.138 FRINTP (vector)

Floating-point round to integral, toward positive infinity (vector)




### *Vector variant*

```
FRINTP <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when `sz = 0, Q = 0`
- 4S when `sz = 0, Q = 1`
- 2D when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

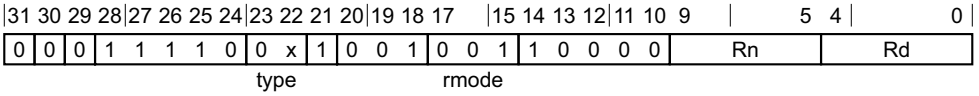
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C7.3.139 FRINTP (scalar)

Floating-point round to integral, toward positive infinity (scalar) : `Vd = roundToIntegralTowardPositive(Vn)`




#### *Single-precision variant*

Applies when `type = 00`.

```
FRINTP <Sd>, <Sn>
```

#### *Double-precision variant*

Applies when `type = 01`.

```
FRINTP <Dd>, <Dn>
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

### Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

### Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

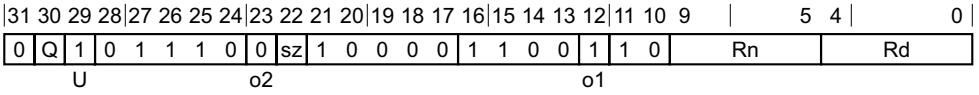
result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.140 FRINTX (vector)

Floating-point round to integral exact, using current rounding mode (vector)




### *Vector variant*

```
FRINTX <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when `sz = 0, Q = 0`

4S when `sz = 0, Q = 1`

2D when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

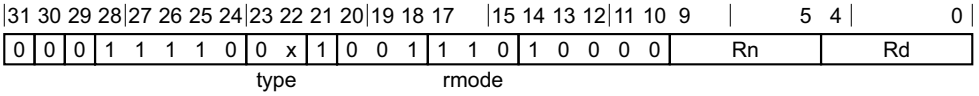
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C7.3.141 FRINTX (scalar)

Floating-point round to integral exact, using current rounding mode (scalar) : `Vd = roundToIntegralExact(Vn)`




#### *Single-precision variant*

Applies when `type = 00`.

```
FRINTX <Sd>, <Sn>
```

#### *Double-precision variant*

Applies when `type = 01`.

```
FRINTX <Dd>, <Dn>
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

### Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

### Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

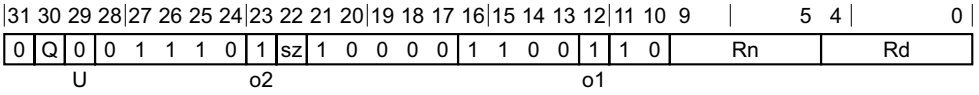
result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.142 FRINTZ (vector)

Floating-point round to integral, toward zero (vector)




### *Vector variant*

```
FRINTZ <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when `sz = 0, Q = 0`
- 4S when `sz = 0, Q = 1`
- 2D when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

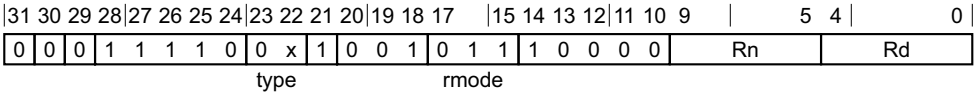
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C7.3.143 FRINTZ (scalar)

Floating-point round to integral, toward zero (scalar) : `Vd = roundToIntegralTowardZero(Vn)`




#### *Single-precision variant*

Applies when `type = 00`.

```
FRINTZ <Sd>, <Sn>
```

#### *Double-precision variant*

Applies when `type = 01`.

```
FRINTZ <Dd>, <Dn>
```

#### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

### Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

### Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

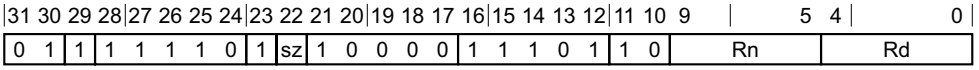
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.144 FRSQRTE

Floating-point reciprocal square root estimate

### Scalar




#### *Scalar variant*

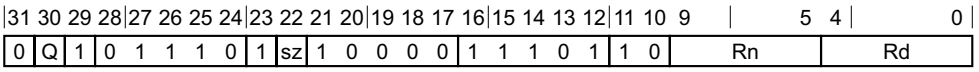
```
FRSQRTE <V><d>, <V><n>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector




#### *Vector variant*

```
FRSQRTE <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0

4S when sz = 0, Q = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR);

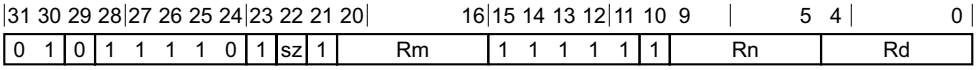
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.145 FRSQRTS

Floating-point reciprocal square root step

### Scalar



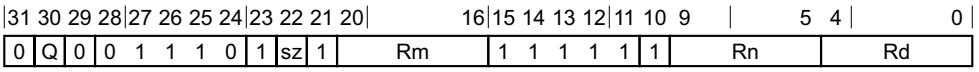

#### *Scalar variant*

```
FRSQRTS <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector




#### *Vector variant*

```
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

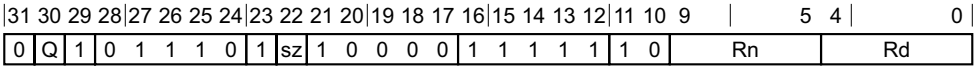
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.146 FSQRT (vector)

Floating-point square root (vector)




### *Vector variant*

```
FSQRT <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

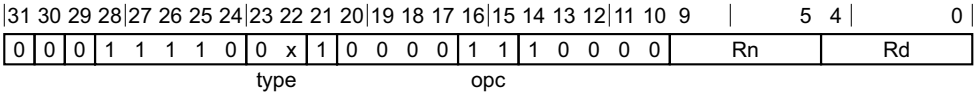
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPSqrt(element, FPCR);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.147 FSQRT (scalar)

Floating-point square root (scalar) : `Vd = sqrt(Vn)`




### *Single-precision variant*

Applies when `type = 00`.

```
FSQRT <Sd>, <Sn>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FSQRT <Dd>, <Dn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

FPUnaryOp fpop;
case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_NEG;
    when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
    when FPUnaryOp_MOV  result = operand;
    when FPUnaryOp_ABS  result = FPAbs(operand);
    when FPUnaryOp_NEG  result = FPNeg(operand);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

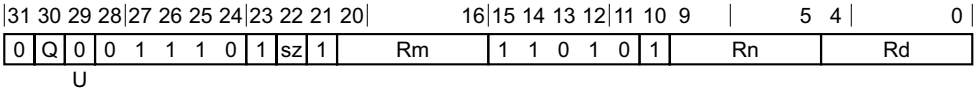
```
        when FPUnaryOp_SQRT result = FPSqrt(operand, FPCR);

    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.148 FSUB (vector)

Floating-point subtract (vector)




### *Three registers of the same type variant*

```
FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when `sz = 0, Q = 0`
- 4S when `sz = 0, Q = 1`
- 2D when `sz = 1, Q = 1`

The encoding `sz = 1, Q = 0` is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) diff;

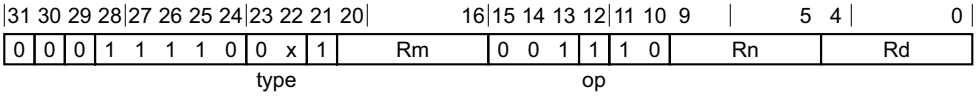
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.149 FSUB (scalar)

Floating-point subtract (scalar) : `Vd = Vn - Vm`




### *Single-precision variant*

Applies when `type = 00`.

```
FSUB <Sd>, <Sn>, <Sm>
```

### *Double-precision variant*

Applies when `type = 01`.

```
FSUB <Dd>, <Dn>, <Dm>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UnallocatedEncoding();

boolean sub_op = (op == '1');
```

## Assembler symbols

| | |
|---|---|
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Dn>` | Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Dm>` | Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| `<Sn>` | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<Sm>` | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if sub_op then
    result = FPSub(operand1, operand2, FPCR);
else
    result = FPAdd(operand1, operand2, FPCR);

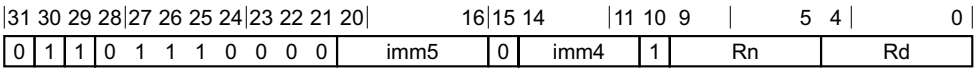
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.150 INS (element)

Insert vector element from another vector element

This instruction is used by the alias MOV (element). The alias is always the preferred disassembly.




### *Advanced SIMD variant*

```
INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UnallocatedEncoding();

integer dst_index = UInt(imm5<4:size+1>);
integer src_index = UInt(imm4<3:size>);
integer idxdsize = if imm4<3> == '1' then 128 else 64;
// imm4<size-1:0> is IGNORED

integer esize = 8 << size;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in the "imm5" field. It can have the following values:

| | |
|---|---|
| B | when imm5 = xxxx1 |
| H | when imm5 = xxx10 |
| S | when imm5 = xx100 |
| D | when imm5 = x1000 |

The encoding imm5 = x0000 is reserved.

<index1> Is the destination element index encoded in the "imm5" field. It can have the following values:

| | |
|---|---|
| imm5<4:1> | when imm5 = xxxx1 |
| imm5<4:2> | when imm5 = xxx10 |
| imm5<4:3> | when imm5 = xx100 |
| imm5<4> | when imm5 = x1000 |

The encoding imm5 = x0000 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index2> Is the source element index encoded in the "imm5:imm4" field. It can have the following values:

| | |
|---|---|
| imm4<3:0> | when imm5 = xxxx1 |
| imm4<3:1> | when imm5 = xxx10 |
| imm4<3:2> | when imm5 = xx100 |
| imm4<3> | when imm5 = x1000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `imm5 = x0000` is reserved.

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(128) result;

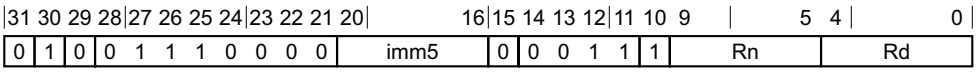
result = V[d];
Elem[result, dst_index, esize] = Elem[operand, src_index, esize];
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.151 INS (general)

Insert vector element from general-purpose register

This instruction is used by the alias MOV (from general). The alias is always the preferred disassembly.




### *Advanced SIMD variant*

```
INS <Vd>.<Ts>[<index>], <R><n>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);

if size > 3 then UnallocatedEncoding();
integer index = UInt(imm5<4:size+1>);

integer esize = 8 << size;
integer datasize = 128;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in the "imm5" field. It can have the following values:

| | |
|---|---|
| B | when imm5 = xxxx1 |
| H | when imm5 = xxx10 |
| S | when imm5 = xx100 |
| D | when imm5 = x1000 |

The encoding imm5 = x0000 is reserved.

<index> Is the element index encoded in the "imm5" field. It can have the following values:

| | |
|---|---|
| imm5<4:1> | when imm5 = xxxx1 |
| imm5<4:2> | when imm5 = xxx10 |
| imm5<4:3> | when imm5 = xx100 |
| imm5<4> | when imm5 = x1000 |

The encoding imm5 = x0000 is reserved.

<R> Is the width specifier for the general-purpose source register, encoded in the "imm5" field. It can have the following values:

| | |
|---|---|
| W | when imm5 = xxxx1 |
| W | when imm5 = xxx10 |
| W | when imm5 = xx100 |
| X | when imm5 = x1000 |

The encoding imm5 = x0000 is reserved.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(datasize) result;

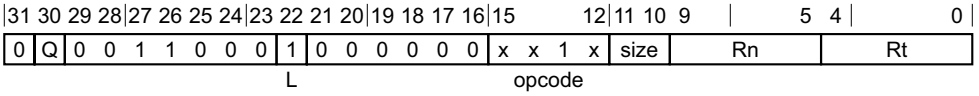
result = V[d];
Elem[result, index, esize] = element;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.152 LD1 (multiple structures)

Load multiple 1-element structures to one, two, three or four registers

### No offset




#### *One register variant*

Applies when `opcode = 0111`.

```
LD1 { <Vt>.<T> }, [<Xn|SP>]
```

#### *Two registers variant*

Applies when `opcode = 1010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

#### *Three registers variant*

Applies when `opcode = 0110`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

#### *Four registers variant*

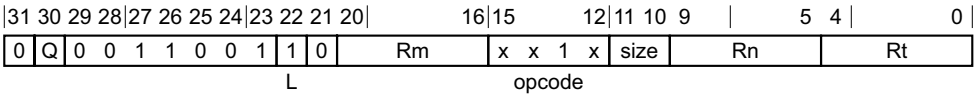
Applies when `opcode = 0010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *One register, immediate offset variant*

Applies when `Rm = 11111 && opcode = 0111`.

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

#### *One register, register offset variant*

Applies when `Rm != 11111 && opcode = 0111`.

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

#### *Two registers, immediate offset variant*

Applies when `Rm = 11111 && opcode = 1010`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

#### *Two registers, register offset variant*

Applies when `Rm != 11111 && opcode = 1010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

#### *Three registers, immediate offset variant*

Applies when `Rm = 11111 && opcode = 0110`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

#### *Three registers, register offset variant*

Applies when `Rm != 11111 && opcode = 0110`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

#### *Four registers, immediate offset variant*

Applies when `Rm = 11111 && opcode = 0010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

#### *Four registers, register offset variant*

Applies when `Rm != 11111 && opcode = 0010`.

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| `8B` | when `size = 00, Q = 0` |
| `16B` | when `size = 00, Q = 1` |
| `4H` | when `size = 01, Q = 0` |
| `8H` | when `size = 01, Q = 1` |
| `2S` | when `size = 10, Q = 0` |
| `4S` | when `size = 10, Q = 1` |
| `1D` | when `size = 11, Q = 0` |
| `2D` | when `size = 11, Q = 1` |

`<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

`<Vt3>` Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

`<Vt4>` Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#8 when Q = 0

#16 when Q = 1

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#16 when Q = 0

#32 when Q = 1

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#24 when Q = 0

#48 when Q = 1

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#32 when Q = 0

#64 when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

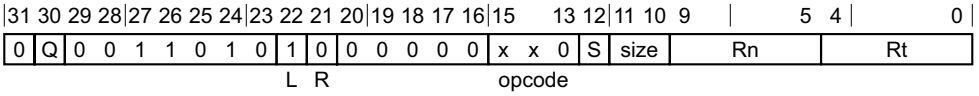
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.153 LD1 (single structure)

Load single 1-element structure to one lane of one register

### No offset




#### *8-bit variant*

Applies when `opcode = 000`.

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>]
```

#### *16-bit variant*

Applies when `opcode = 010 && size = x0`.

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>]
```

#### *32-bit variant*

Applies when `opcode = 100 && size = 00`.

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>]
```

#### *64-bit variant*

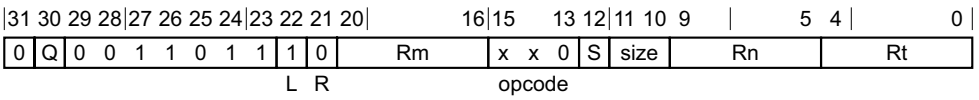
Applies when `opcode = 100 && S = 0 && size = 01`.

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>]
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *8-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 000`.

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

#### *8-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 000`.

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

#### *16-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 010 && size = x0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

#### *16-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 010 && size = x0`.

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

#### *32-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 100 && size = 00`.

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

#### *32-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 100 && size = 00`.

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

#### *64-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 100 && S = 0 && size = 01`.

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

#### *64-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 100 && S = 0 && size = 01`.

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

| | |
|---|---|
| `<Vt>` | Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<index>` | For the 8-bit variant: is the element index, encoded in "Q:S:size".<br>For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".<br>For the 32-bit variant: is the element index, encoded in "Q:S".<br>For the 64-bit variant: is the element index, encoded in "Q". |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field. |

### Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

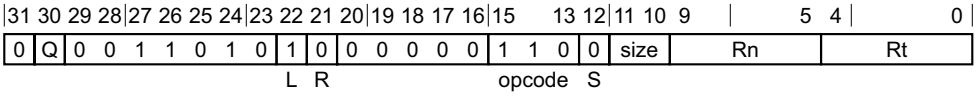
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.154 LD1R

Load single 1-element structure and replicate to all lanes (of one register)

### No offset



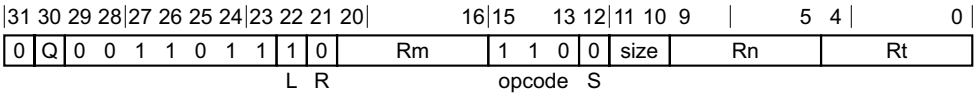

#### *No offset variant*

```
LD1R { <Vt>.<T> }, [<Xn|SP>]
```

#### *Decode for this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *Immediate offset variant*

Applies when `Rm = 11111`.

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <imm>
```

#### *Register offset variant*

Applies when `Rm != 11111`.

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when `size = 00, Q = 0` |
| 16B | when `size = 00, Q = 1` |
| 4H | when `size = 01, Q = 0` |
| 8H | when `size = 01, Q = 1` |
| 2S | when `size = 10, Q = 0` |
| 4S | when `size = 10, Q = 1` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1D when size = 11, Q = 0

2D when size = 11, Q = 1

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:

#1 when size = 00

#2 when size = 01

#4 when size = 10

#8 when size = 11

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

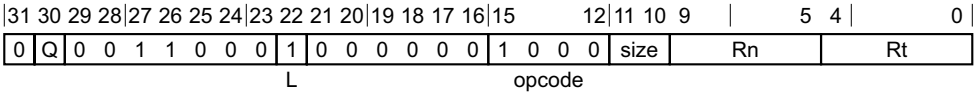
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.155 LD2 (multiple structures)

Load multiple 2-element structures to two registers

### No offset



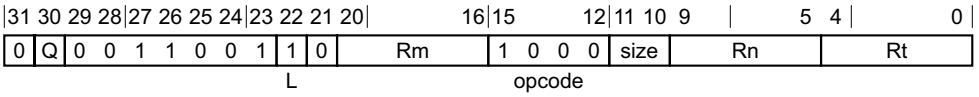

#### *No offset variant*

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

#### *Decode for this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *Immediate offset variant*

Applies when `Rm = 11111`.

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

#### *Register offset variant*

Applies when `Rm != 11111`.

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when `size = 00, Q = 0` |
| 16B | when `size = 00, Q = 1` |
| 4H | when `size = 01, Q = 0` |
| 8H | when `size = 01, Q = 1` |
| 2S | when `size = 10, Q = 0` |
| 4S | when `size = 10, Q = 1` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#16 when Q = 0

#32 when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            tt = (tt + 1) MOD 32;

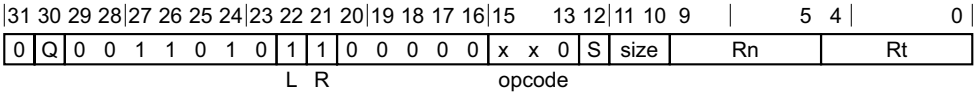
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.156 LD2 (single structure)

Load single 2-element structure to one lane of two registers

### No offset




#### *8-bit variant*

Applies when `opcode = 000`.

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

#### *16-bit variant*

Applies when `opcode = 010 && size = x0`.

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

#### *32-bit variant*

Applies when `opcode = 100 && size = 00`.

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

#### *64-bit variant*

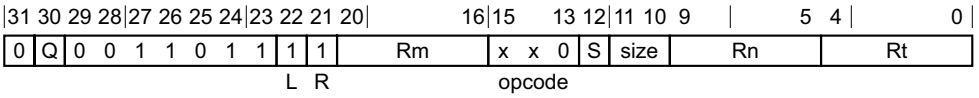
Applies when `opcode = 100 && S = 0 && size = 01`.

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *8-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 000`.

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

#### *8-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 000`.

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

#### *16-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 010 && size = x0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

#### *16-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 010 && size = x0`.

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

#### *32-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 100 && size = 00`.

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

#### *32-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 100 && size = 00`.

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

#### *64-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 100 && S = 0 && size = 01`.

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

#### *64-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 100 && S = 0 && size = 01`.

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

| | |
|---|---|
| `<Vt>` | Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Vt2>` | Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32. |
| `<index>` | For the 8-bit variant: is the element index, encoded in "Q:S:size".<br>For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".<br>For the 32-bit variant: is the element index, encoded in "Q:S".<br>For the 64-bit variant: is the element index, encoded in "Q". |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field. |

### Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

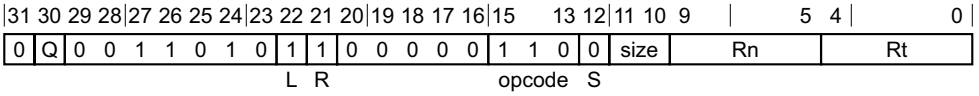
```
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.157 LD2R

Load single 2-element structure and replicate to all lanes of two registers

### No offset



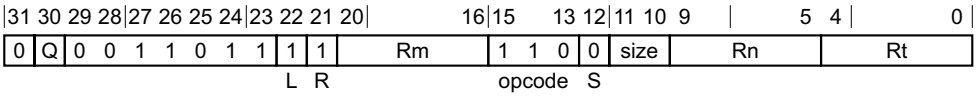

#### *No offset variant*

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

#### *Decode for this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *Immediate offset variant*

Applies when `Rm = 11111`.

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

#### *Register offset variant*

Applies when `Rm != 11111`.

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when `size = 00, Q = 0` |
| 16B | when `size = 00, Q = 1` |
| 4H | when `size = 01, Q = 0` |
| 8H | when `size = 01, Q = 1` |
| 2S | when `size = 10, Q = 0` |
| 4S | when `size = 10, Q = 1` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1D when size = 11, Q = 0

2D when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:

#2 when size = 00

#4 when size = 01

#8 when size = 10

#16 when size = 11

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

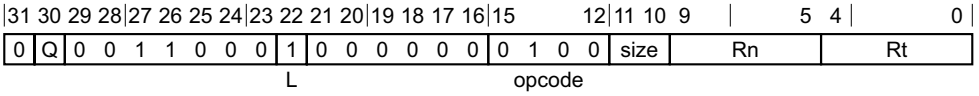
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.158 LD3 (multiple structures)

Load multiple 3-element structures to three registers

### No offset



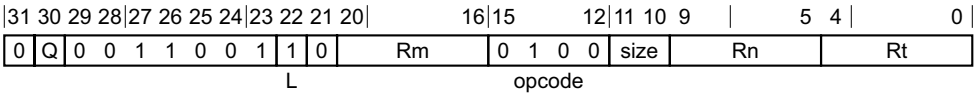

#### *No offset variant*

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

#### *Decode for this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *Immediate offset variant*

Applies when `Rm = 11111`.

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

#### *Register offset variant*

Applies when `Rm != 11111`.

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| `8B` | when `size = 00, Q = 0` |
| `16B` | when `size = 00, Q = 1` |
| `4H` | when `size = 01, Q = 0` |
| `8H` | when `size = 01, Q = 1` |
| `2S` | when `size = 10, Q = 0` |
| `4S` | when `size = 10, Q = 1` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#24 when Q = 0

#48 when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                    Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

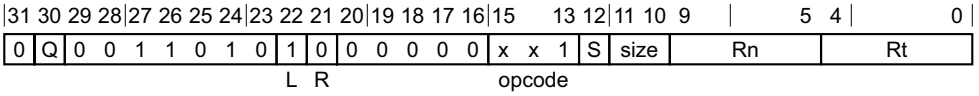
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.159 LD3 (single structure)

Load single 3-element structure to one lane of three registers)

### No offset




#### *8-bit variant*

Applies when `opcode = 001`.

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

#### *16-bit variant*

Applies when `opcode = 011 && size = x0`.

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

#### *32-bit variant*

Applies when `opcode = 101 && size = 00`.

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

#### *64-bit variant*

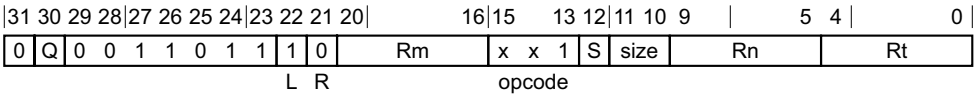
Applies when `opcode = 101 && S = 0 && size = 01`.

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *8-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 001`.

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

#### *8-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 001`.

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

#### *16-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 011 && size = x0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

#### *16-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 011 && size = x0`.

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

#### *32-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 101 && size = 00`.

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

#### *32-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 101 && size = 00`.

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

#### *64-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 101 && S = 0 && size = 01`.

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

#### *64-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 101 && S = 0 && size = 01`.

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

| | |
|---|---|
| `<Vt>` | Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Vt2>` | Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32. |
| `<Vt3>` | Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32. |
| `<index>` | For the 8-bit variant: is the element index, encoded in "Q:S:size".<br>For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".<br>For the 32-bit variant: is the element index, encoded in "Q:S".<br>For the 64-bit variant: is the element index, encoded in "Q". |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field. |

### Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

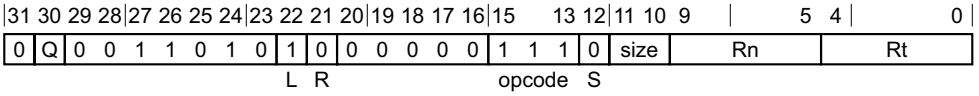
```
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.160 LD3R

Load single 3-element structure and replicate to all lanes of three registers

### No offset



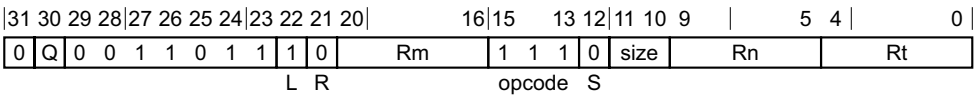

#### *No offset variant*

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

#### *Decode for this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *Immediate offset variant*

Applies when `Rm = 11111`.

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

#### *Register offset variant*

Applies when `Rm != 11111`.

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| `8B` | when `size = 00, Q = 0` |
| `16B` | when `size = 00, Q = 1` |
| `4H` | when `size = 01, Q = 0` |
| `8H` | when `size = 01, Q = 1` |
| `2S` | when `size = 10, Q = 0` |
| `4S` | when `size = 10, Q = 1` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1D when size = 11, Q = 0

2D when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:

#3 when size = 00

#6 when size = 01

#12 when size = 10

#24 when size = 11

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

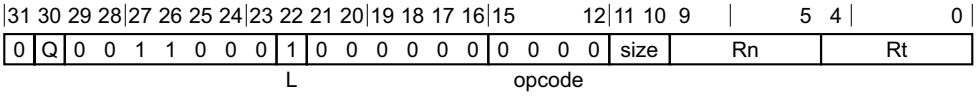
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.161 LD4 (multiple structures)

Load multiple 4-element structures to four registers

### No offset



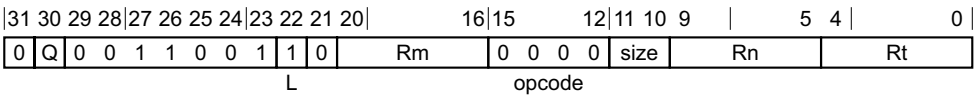

#### *No offset variant*

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

#### *Decode for this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *Immediate offset variant*

Applies when `Rm = 11111`.

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

#### *Register offset variant*

Applies when `Rm != 11111`.

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| `8B` | when `size = 00, Q = 0` |
| `16B` | when `size = 00, Q = 1` |
| `4H` | when `size = 01, Q = 0` |
| `8H` | when `size = 01, Q = 1` |
| `2S` | when `size = 10, Q = 0` |
| `4S` | when `size = 10, Q = 1` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#32 when Q = 0

#64 when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                    V[tt] = rval;
                else // memop == MemOp_STORE
                    Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

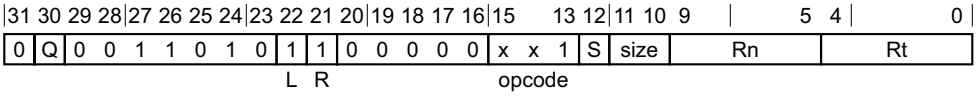
    if wback then
        if m != 31 then
            offs = X[m];
        if n == 31 then
            SP[] = address + offs;
        else
            X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.162 LD4 (single structure)

Load single 4-element structure to one lane of four registers

### No offset




#### *8-bit variant*

Applies when `opcode = 001`.

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

#### *16-bit variant*

Applies when `opcode = 011 && size = x0`.

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

#### *32-bit variant*

Applies when `opcode = 101 && size = 00`.

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

#### *64-bit variant*

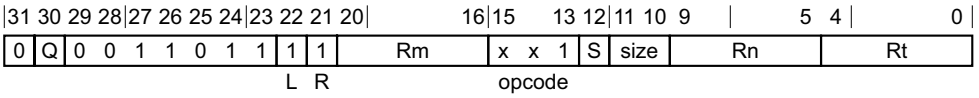
Applies when `opcode = 101 && S = 0 && size = 01`.

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *8-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 001`.

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

#### *8-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 001`.

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

#### *16-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 011 && size = x0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

#### *16-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 011 && size = x0`.

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

#### *32-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 101 && size = 00`.

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

#### *32-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 101 && size = 00`.

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

#### *64-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 101 && S = 0 && size = 01`.

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

#### *64-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 101 && S = 0 && size = 01`.

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

| | |
|---|---|
| `<Vt>` | Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Vt2>` | Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32. |
| `<Vt3>` | Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32. |
| `<Vt4>` | Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32. |
| `<index>` | For the 8-bit variant: is the element index, encoded in "Q:S:size".<br>For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".<br>For the 32-bit variant: is the element index, encoded in "Q:S".<br>For the 64-bit variant: is the element index, encoded in "Q". |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field. |

### Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

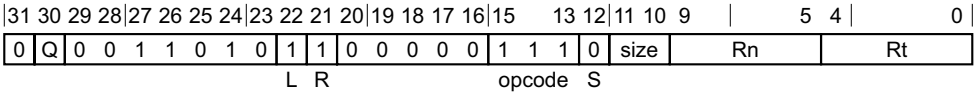
```
    offs = X[m];
if n == 31 then
    SP[] = address + offs;
else
    X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.163 LD4R

Load single 4-element structure and replicate to all lanes of four registers

### No offset



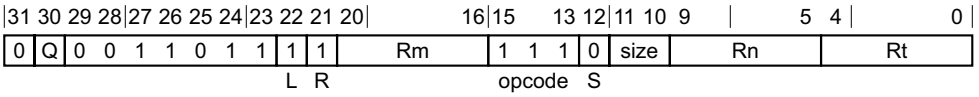

#### *No offset variant*

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

#### *Decode for this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *Immediate offset variant*

Applies when `Rm = 11111`.

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

#### *Register offset variant*

Applies when `Rm != 11111`.

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when `size = 00, Q = 0` |
| 16B | when `size = 00, Q = 1` |
| 4H | when `size = 01, Q = 0` |
| 8H | when `size = 01, Q = 1` |
| 2S | when `size = 10, Q = 0` |
| 4S | when `size = 10, Q = 1` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1D when size = 11, Q = 0

2D when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:

#4 when size = 00

#8 when size = 01

#16 when size = 10

#32 when size = 11

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    address = SP[];
else
    address = X[n];

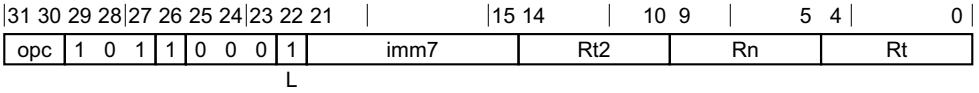
offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.164 LDNP (SIMD&FP)

Load pair of SIMD&FP registers, with non-temporal hint




### *32-bit variant*

Applies when `opc = 00`.

```
LDNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### *64-bit variant*

Applies when `opc = 01`.

```
LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### *128-bit variant*

Applies when `opc = 10`.

```
LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

### *Decode for all variants of this encoding*

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

## Assembler symbols

| | |
|---|---|
| `<Dt1>` | Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Dt2>` | Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| `<Qt1>` | Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Qt2>` | Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| `<St1>` | Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<St2>` | Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<imm>` | For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.<br>For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.<br>For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VECSTREAM;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t]  = data1;
        V[t2] = data2;

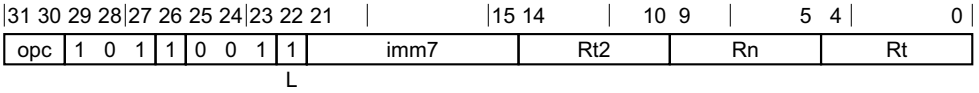
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.165 LDP (SIMD&FP)

Load pair of SIMD&FP registers

### Post-index




#### *32-bit variant*

Applies when opc = 00.

```
LDP <St1>, <St2>, [<Xn|SP>], #<imm>
```

#### *64-bit variant*

Applies when opc = 01.

```
LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

#### *128-bit variant*

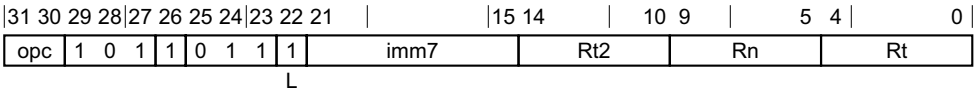
Applies when opc = 10.

```
LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

#### *Decode for all variants of this encoding*

```
boolean wback  = TRUE;
boolean postindex = TRUE;
```

### Pre-index




#### *32-bit variant*

Applies when opc = 00.

```
LDP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

#### *64-bit variant*

Applies when opc = 01.

```
LDP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

#### *128-bit variant*

Applies when opc = 10.

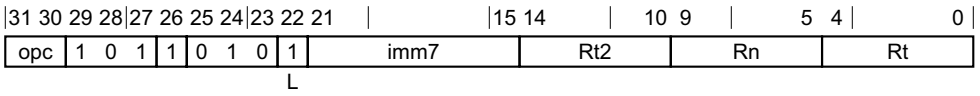
```
LDP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

#### *Decode for all variants of this encoding*

```
boolean wback  = TRUE;
boolean postindex = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Signed offset




#### *32-bit variant*

Applies when `opc = 00`.

```
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

#### *64-bit variant*

Applies when `opc = 01`.

```
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

#### *128-bit variant*

Applies when `opc = 10`.

```
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

#### *Decode for all variants of this encoding*

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

### Assembler symbols

| | |
|---|---|
| `<Dt1>` | Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Dt2>` | Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| `<Qt1>` | Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Qt2>` | Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| `<St1>` | Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<St2>` | Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<imm>` | For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.<br>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.<br>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.<br>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.<br>For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.<br>For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t]  = data1;
        V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.166 LDR (immediate, SIMD&FP)

Load SIMD&FP register (immediate offset)

### Post-index




#### *8-bit variant*

Applies when `size = 00 && opc = 01`.

```
LDR <Bt>, [<Xn|SP>], #<simm>
```

#### *16-bit variant*

Applies when `size = 01 && opc = 01`.

```
LDR <Ht>, [<Xn|SP>], #<simm>
```

#### *32-bit variant*

Applies when `size = 10 && opc = 01`.

```
LDR <St>, [<Xn|SP>], #<simm>
```

#### *64-bit variant*

Applies when `size = 11 && opc = 01`.

```
LDR <Dt>, [<Xn|SP>], #<simm>
```

#### *128-bit variant*

Applies when `size = 00 && opc = 11`.

```
LDR <Qt>, [<Xn|SP>], #<simm>
```

#### *Decode for all variants of this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index




#### *8-bit variant*

Applies when `size = 00 && opc = 01`.

```
LDR <Bt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### 16-bit variant

Applies when size = 01 && opc = 01.

```
LDR <Ht>, [<Xn|SP>, #<simm>]!
```

#### 32-bit variant

Applies when size = 10 && opc = 01.

```
LDR <St>, [<Xn|SP>, #<simm>]!
```

#### 64-bit variant

Applies when size = 11 && opc = 01.

```
LDR <Dt>, [<Xn|SP>, #<simm>]!
```

#### 128-bit variant

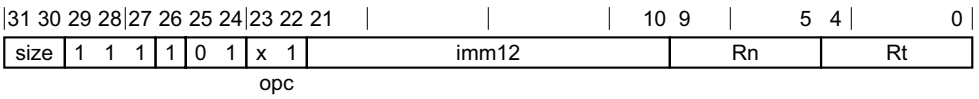
Applies when size = 00 && opc = 11.

```
LDR <Qt>, [<Xn|SP>, #<simm>]!
```

#### Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset




#### 8-bit variant

Applies when size = 00 && opc = 01.

```
LDR <Bt>, [<Xn|SP>{, #<pimm>}]
```

#### 16-bit variant

Applies when size = 01 && opc = 01.

```
LDR <Ht>, [<Xn|SP>{, #<pimm>}]
```

#### 32-bit variant

Applies when size = 10 && opc = 01.

```
LDR <St>, [<Xn|SP>{, #<pimm>}]
```

#### 64-bit variant

Applies when size = 11 && opc = 01.

```
LDR <Dt>, [<Xn|SP>{, #<pimm>}]
```

#### 128-bit variant

Applies when size = 00 && opc = 11.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LDR <Qt>, [<Xn|SP>{, #<pimm>}]
```

### Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

## Assembler symbols

| | |
|---|---|
| <Bt> | Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Dt> | Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Ht> | Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Qt> | Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <St> | Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| <pimm> | For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.<br>For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.<br>For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.<br>For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            data = V[t];
            Mem[address, datasize DIV 8, acctype] = data;

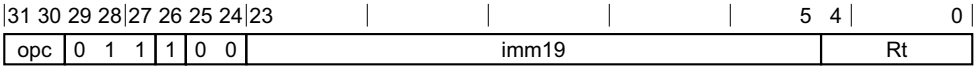
        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            V[t] = data;

    if wback then
        if postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.167 LDR (literal, SIMD&FP)

Load SIMD&FP register (PC-relative literal)




### *32-bit variant*

Applies when `opc = 00`.

```
LDR <St>, <label>
```

### *64-bit variant*

Applies when `opc = 01`.

```
LDR <Dt>, <label>
```

### *128-bit variant*

Applies when `opc = 10`.

```
LDR <Qt>, <label>
```

### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 16;
    when '11'
        UnallocatedEncoding();

offset = SignExtend(imm19:'00', 64);
```

## Assembler symbols

| | |
|---|---|
| `<Dt>` | Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field. |
| `<Qt>` | Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field. |
| `<St>` | Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field. |
| `<label>` | Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4. |

## Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

CheckFPAdvSIMDEnabled64();
```

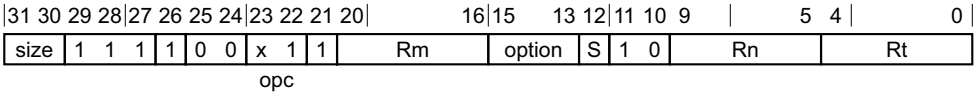

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
data = Mem[address, size, AccType_VEC];
V[t] = data;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.168 LDR (register, SIMD&FP)

Load SIMD&FP register (register offset)




### *8-bit variant*

Applies when `size = 00 && opc = 01 && option != 011`.

```
LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### *8-bit variant*

Applies when `size = 00 && opc = 01 && option = 011`.

```
LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### *16-bit variant*

Applies when `size = 01 && opc = 01`.

```
LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *32-bit variant*

Applies when `size = 10 && opc = 01`.

```
LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *64-bit variant*

Applies when `size = 11 && opc = 01`.

```
LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *128-bit variant*

Applies when `size = 00 && opc = 11`.

```
LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### *Decode for all variants of this encoding*

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler symbols

| | |
|---|---|
| `<Bt>` | Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Dt>` | Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Ht>` | Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qt>` Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<St>` Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` For the 8-bit variant: is the index extend specifier, encoded in the "option" field. It can have the following values:

`UXTW` when `option = 010`

`SXTW` when `option = 110`

`SXTX` when `option = 111`

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

`UXTW` when `option = 010`

`LSL` when `option = 011`

`SXTW` when `option = 110`

`SXTX` when `option = 111`

`<amount>` For the 8-bit variant: is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#1` when `S = 1`

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#2` when `S = 1`

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#3` when `S = 1`

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#4` when `S = 1`

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.169 LDUR (SIMD&FP)

Load SIMD&FP register (unscaled offset)




### 8-bit variant

Applies when size = 00 && opc = 01.

```
LDUR <Bt>, [<Xn|SP>{, #<simm>}]
```

### 16-bit variant

Applies when size = 01 && opc = 01.

```
LDUR <Ht>, [<Xn|SP>{, #<simm>}]
```

### 32-bit variant

Applies when size = 10 && opc = 01.

```
LDUR <St>, [<Xn|SP>{, #<simm>}]
```

### 64-bit variant

Applies when size = 11 && opc = 01.

```
LDUR <Dt>, [<Xn|SP>{, #<simm>}]
```

### 128-bit variant

Applies when size = 00 && opc = 11.

```
LDUR <Qt>, [<Xn|SP>{, #<simm>}]
```

### Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| <Bt> | Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Dt> | Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Ht> | Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Qt> | Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <St> | Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

### Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

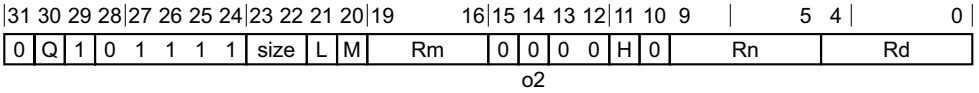
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.170 MLA (by element)

Multiply-add to accumulator (vector, by element)




### *Vector variant*

```
MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation

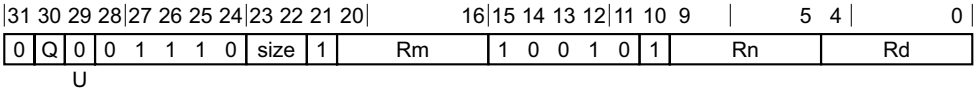
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.171 MLA (vector)

Multiply-add to accumulator (vector)




### *Three registers of the same type variant*

```
MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1) * UInt(element2))<esize-1:0>;
    if sub_op then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

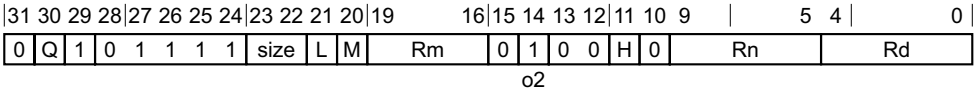
```
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        else
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;

    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.172 MLS (by element)

Multiply-subtract from accumulator (vector, by element)




### *Vector variant*

```
MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation

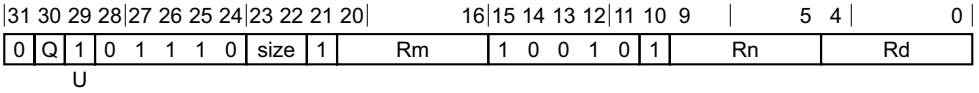
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.173 MLS (vector)

Multiply-subtract from accumulator (vector)




### *Three registers of the same type variant*

```
MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1) * UInt(element2))<esize-1:0>;
    if sub_op then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d] = result;
```

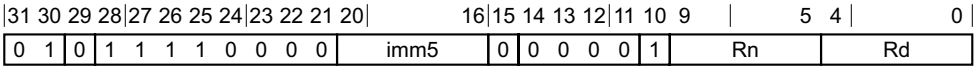

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.174 MOV (scalar)

Move vector element to scalar

This instruction is an alias of the DUP (element) instruction. This means that:

- The encodings in this description are named to match the encodings of DUP (element).
- The description of DUP (element) gives the operational pseudocode for this instruction.




### *Scalar variant*

```
MOV <V><d>, <Vn>.<T>[<index>]
```

is equivalent to

```
DUP  <V><d>, <Vn>.<T>[<index>]
```

and is always the preferred disassembly.

### Assembler symbols

<V> Is the destination width specifier, encoded in the "imm5" field. It can have the following values:

- B when imm5 = xxxx1
- H when imm5 = xxx10
- S when imm5 = xx100
- D when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the element width specifier, encoded in the "imm5" field. It can have the following values:

- B when imm5 = xxxx1
- H when imm5 = xxx10
- S when imm5 = xx100
- D when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<index> Is the element index encoded in the "imm5" field. It can have the following values:

- imm5<4:1> when imm5 = xxxx1
- imm5<4:2> when imm5 = xxx10
- imm5<4:3> when imm5 = xx100
- imm5<4> when imm5 = x1000

The encoding imm5 = x0000 is reserved.

### Operation

The description of DUP (element) gives the operational pseudocode for this instruction.

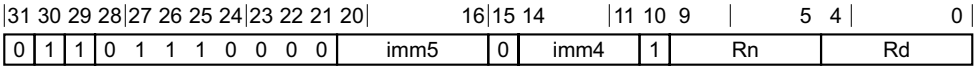

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.175 MOV (element)

Move vector element to another vector element

This instruction is an alias of the INS (element) instruction. This means that:

- The encodings in this description are named to match the encodings of INS (element).
- The description of INS (element) gives the operational pseudocode for this instruction.




### *Advanced SIMD variant*

```
MOV <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

is equivalent to

```
INS  <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

and is always the preferred disassembly.

### Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ts>` Is an element size specifier, encoded in the "imm5" field. It can have the following values:

- `B` when `imm5 = xxxx1`
- `H` when `imm5 = xxx10`
- `S` when `imm5 = xx100`
- `D` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<index1>` Is the destination element index encoded in the "imm5" field. It can have the following values:

- `imm5<4:1>` when `imm5 = xxxx1`
- `imm5<4:2>` when `imm5 = xxx10`
- `imm5<4:3>` when `imm5 = xx100`
- `imm5<4>` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<index2>` Is the source element index encoded in the "imm5:imm4" field. It can have the following values:

- `imm4<3:0>` when `imm5 = xxxx1`
- `imm4<3:1>` when `imm5 = xxx10`
- `imm4<3:2>` when `imm5 = xx100`
- `imm4<3>` when `imm5 = x1000`

The encoding `imm5 = x0000` is reserved.

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

The description of INS (element) gives the operational pseudocode for this instruction.

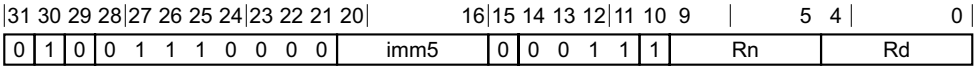

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.176 MOV (from general)

Move general-purpose register to a vector element

This instruction is an alias of the INS (general) instruction. This means that:

- The encodings in this description are named to match the encodings of INS (general).
- The description of INS (general) gives the operational pseudocode for this instruction.




### *Advanced SIMD variant*

```
MOV <Vd>.<Ts>[<index>], <R><n>
```

is equivalent to

```
INS  <Vd>.<Ts>[<index>], <R><n>
```

and is always the preferred disassembly.

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in the "imm5" field. It can have the following values:

- B when imm5 = xxxx1
- H when imm5 = xxx10
- S when imm5 = xx100
- D when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<index> Is the element index encoded in the "imm5" field. It can have the following values:

- imm5<4:1> when imm5 = xxxx1
- imm5<4:2> when imm5 = xxx10
- imm5<4:3> when imm5 = xx100
- imm5<4> when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<R> Is the width specifier for the general-purpose source register, encoded in the "imm5" field. It can have the following values:

- W when imm5 = xxxx1
- W when imm5 = xxx10
- W when imm5 = xx100
- X when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

The description of INS (general) gives the operational pseudocode for this instruction.

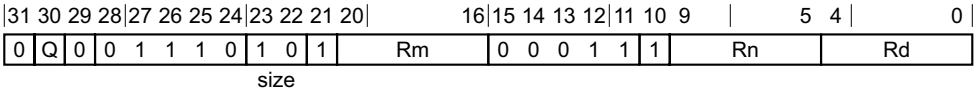

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.177 MOV (vector)

Move vector

This instruction is an alias of the ORR (vector, register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) gives the operational pseudocode for this instruction.




### *Three registers of the same type variant*

`MOV <Vd>.<T>, <Vn>.<T>`

is equivalent to

`ORR  <Vd>.<T>, <Vn>.<T>, <Vn>.<T>`

and is the preferred disassembly when `Rm == Rn`.

### Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`8B` when `Q = 0`

`16B` when `Q = 1`

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

### Operation

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

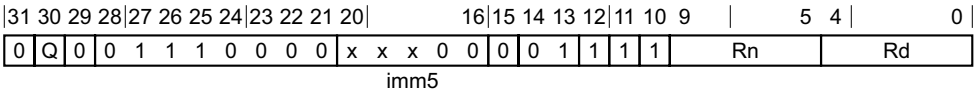

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.178 MOV (to general)

Move vector element to general-purpose register

This instruction is an alias of the UMOV instruction. This means that:

- The encodings in this description are named to match the encodings of UMOV.
- The description of UMOV gives the operational pseudocode for this instruction.




### *32-bit variant*

Applies when `Q = 0 && imm5 = xx100`.

```
MOV <Wd>, <Vn>.S[<index>]
```

is equivalent to

```
UMOV <Wd>, <Vn>.S[<index>]
```

and is always the preferred disassembly.

### *64-bit variant*

Applies when `Q = 1 && imm5 = x1000`.

```
MOV <Xd>, <Vn>.D[<index>]
```

is equivalent to

```
UMOV <Xd>, <Vn>.D[<index>]
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Vn>` | Is the name of the SIMD&FP source register, encoded in the "Rn" field. |
| `<index>` | For the 32-bit variant: is the element index encoded in "imm5<4:3>".<br>For the 64-bit variant: is the element index encoded in "imm5<4>". |

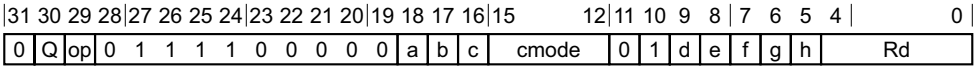
### Operation

The description of UMOV gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.179 MOVI

Move immediate (vector)




### 8-bit variant

Applies when `op = 0 && cmode = 1110`.

```
MOVI <Vd>.<T>, #<imm8>{, LSL #0}
```

### 16-bit shifted immediate variant

Applies when `op = 0 && cmode = 10x0`.

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit shifted immediate variant

Applies when `op = 0 && cmode = 0xx0`.

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit shifting ones variant

Applies when `op = 0 && cmode = 110x`.

```
MOVI <Vd>.<T>, #<imm8>, MSL #<amount>
```

### 64-bit scalar variant

Applies when `Q = 0 && op = 1 && cmode = 1110`.

```
MOVI <Dd>, #<imm>
```

### 64-bit vector variant

Applies when `Q = 1 && op = 1 && cmode = 1110`.

```
MOVI <Vd>.2D, #<imm>
```

### Decode for all variants of this encoding

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

## Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<imm> Is a 64-bit immediate 'aaaaaaaabbbbbbbbccccccccddddddddeeeeeeeeffffffffgggggggghhhhhhhh', encoded in "a:b:c:d:e:f:g:h".

<T> For the 8-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

8B when Q = 0

16B when Q = 1

For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

2S when Q = 0

4S when Q = 1

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:

0 when cmode<1> = 0

8 when cmode<1> = 1

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:

0 when cmode<2:1> = 00

8 when cmode<2:1> = 01

16 when cmode<2:1> = 10

24 when cmode<2:1> = 11

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in the "cmode<0>" field. It can have the following values:

8 when cmode<0> = 0

16 when cmode<0> = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

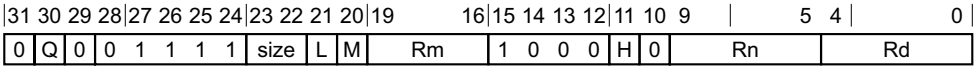
case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.180 MUL (by element)

Multiply (vector, by element)




### *Vector variant*

```
MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

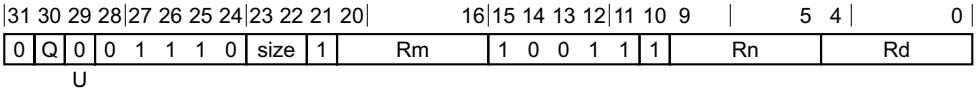
element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.181 MUL (vector)

Multiply (vector)




### *Three registers of the same type variant*

```
MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        product = (UInt(element1) * UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

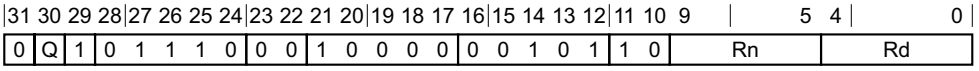

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.182 MVN

Bitwise NOT (vector)

This instruction is an alias of the NOT instruction. This means that:

- The encodings in this description are named to match the encodings of NOT.
- The description of NOT gives the operational pseudocode for this instruction.




### *Vector variant*

`MVN <Vd>.<T>, <Vn>.<T>`

is equivalent to

`NOT  <Vd>.<T>, <Vn>.<T>`

and is always the preferred disassembly.

### Assembler symbols

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

`8B` when `Q = 0`

`16B` when `Q = 1`

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

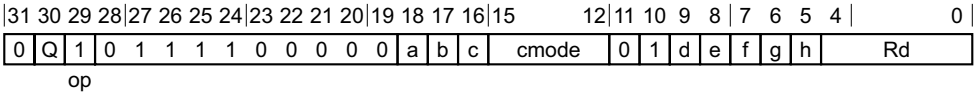
### Operation

The description of NOT gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.183 MVNI

Move inverted immediate (vector)




### *16-bit shifted immediate variant*

Applies when `cmode = 10x0`.

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### *32-bit shifted immediate variant*

Applies when `cmode = 0xx0`.

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### *32-bit shifting ones variant*

Applies when `cmode = 110x`.

```
MVNI <Vd>.<T>, #<imm8>, MSL #<amount>
```

### *Decode for all variants of this encoding*

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0

8H when Q = 1

For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

2S when Q = 0

4S when Q = 1

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:

0 when cmode<1> = 0

8 when cmode<1> = 1

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:

0 when cmode<2:1> = 00

8 when cmode<2:1> = 01

16 when cmode<2:1> = 10

24 when cmode<2:1> = 11

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in the "cmode<0>" field. It can have the following values:

8 when cmode<0> = 0

16 when cmode<0> = 1

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

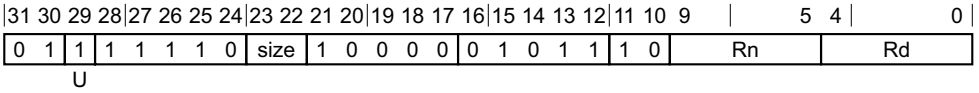
V[rd] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.184 NEG (vector)

Negate (vector)

### Scalar




#### *Scalar variant*

```
NEG <V><d>, <V><n>
```

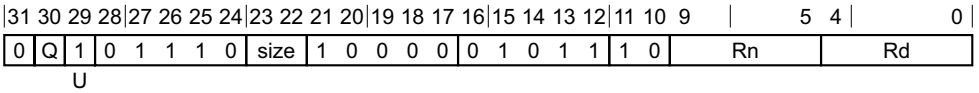
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean neg = (U == '1');
```

### Vector




#### *Vector variant*

```
NEG <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean neg = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

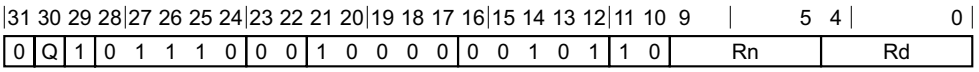
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.185 NOT

Bitwise NOT (vector)

This instruction is used by the alias MVN. The alias is always the preferred disassembly.




### *Vector variant*

```
NOT <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

8B when Q = 0

16B when Q = 1

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

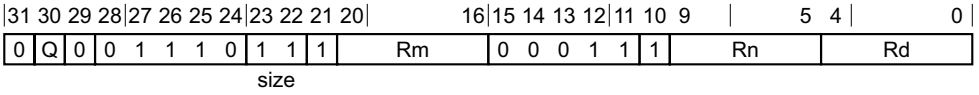
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = NOT(element);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.186 ORN (vector)

Bitwise inclusive OR NOT (vector)




### *Three registers of the same type variant*

```
ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean invert = (size<0> == '1');
LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 8B when Q = 0
- 16B when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

if invert then operand2 = NOT(operand2);

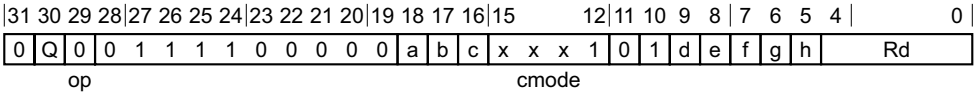
case op of
    when LogicalOp_AND
        result = operand1 AND operand2;
    when LogicalOp_ORR
        result = operand1 OR operand2;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.187 ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate)




### *16-bit variant*

Applies when `cmode = 10x1`.

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### *32-bit variant*

Applies when `cmode = 0xx1`.

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### *Decode for all variants of this encoding*

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when `Q = 0`

8H when `Q = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

2S when Q = 0

4S when Q = 1

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:

0 when cmode<1> = 0

8 when cmode<1> = 1

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:

0 when cmode<2:1> = 00

8 when cmode<2:1> = 01

16 when cmode<2:1> = 10

24 when cmode<2:1> = 11

defaulting to 0 if LSL is omitted.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

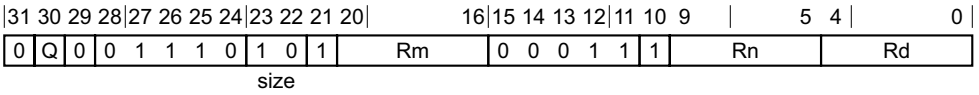
V[rd] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.188 ORR (vector, register)

Bitwise inclusive OR (vector, register)

This instruction is used by the alias MOV (vector). See the *Alias conditions* table for details of when each alias is preferred.




### *Three registers of the same type variant*

```
ORR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean invert = (size<0> == '1');
LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| MOV (vector) | `Rm == Rn` |

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND
        result = operand1 AND operand2;
    when LogicalOp_ORR
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

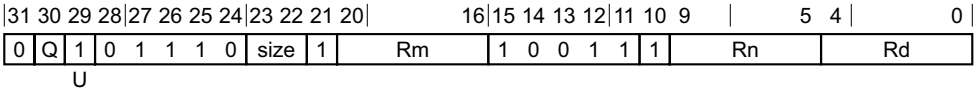
```
            result = operand1 OR operand2;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.189 PMUL

Polynomial multiply




### *Three registers of the same type variant*

```
PMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when `size = 00, Q = 0`

16B when `size = 00, Q = 1`

The following encodings are reserved:

- `size = 01, Q = x`.
- `size = 1x, Q = x`.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1) * UInt(element2))<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

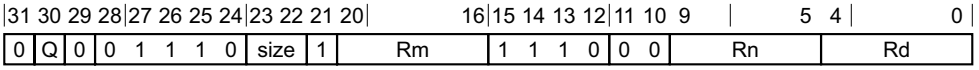
```
            Elem[result, e, esize] = product;

    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.190 PMULL, PMULL2

Polynomial multiply long




### *Three registers, not all the same type variant*

```
PMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '01' || size == '10' then ReservedValue();
if size == '11' && ! HaveCryptoExt() then UnallocatedEncoding();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `1Q` when `size = 11`

The following encodings are reserved:

- `size = 01`.
- `size = 10`.

The '1Q' arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00, Q = 0`
- `16B` when `size = 00, Q = 1`
- `1D` when `size = 11, Q = 0`
- `2D` when `size = 11, Q = 1`

The following encodings are reserved:

- `size = 01, Q = x`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- `size = 10, Q = x`.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
bits(esize) element1;
bits(esize) element2;

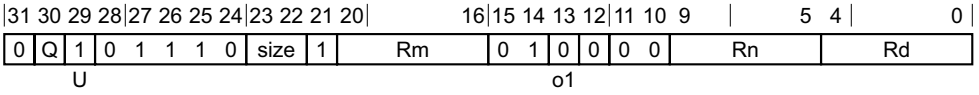
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, 2*esize] = PolynomialMult(element1, element2);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.191 RADDHN, RADDHN2

Rounding add returning high narrow




### *Three registers, not all the same type variant*

```
RADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize)   result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

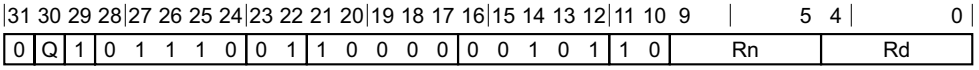
for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.192 RBIT (vector)

Reverse bit order (vector) reverses the bit order in a vector.




### *Vector variant*

```
RBIT <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

8B when `Q = 0`

16B when `Q = 1`

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
bits(esize) rev;

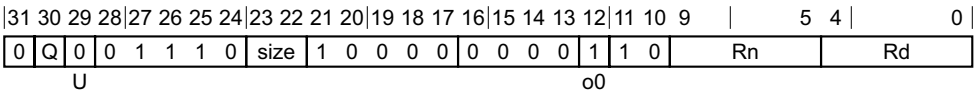
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    for i = 0 to esize-1
        rev<esize-1-i> = element<i>;
    Elem[result, e, esize] = rev;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### C7.3.193 REV16 (vector)

Reverse elements in 16-bit halfwords (vector) reverses the order of 8-bit elements in each halfword of the vector, and places the result in the corresponding destination vector.




#### *Vector variant*

```
REV16 <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op)+UInt(size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when `size = 00, Q = 0`

16B when `size = 00, Q = 1`

The following encodings are reserved:

- `size = 01, Q = x`.
- `size = 1x, Q = x`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

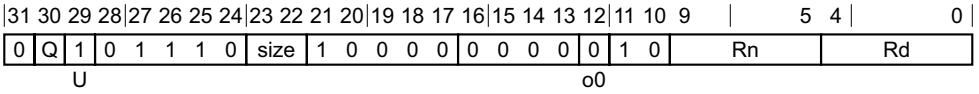
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.194 REV32 (vector)

Reverse elements in 32-bit words (vector) reverses the order of 8-bit, or 16-bit elements in each word of the vector, and places the result in the corresponding destination vector.




### *Vector variant*

```
REV32 <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op)+UInt(size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when `size = 00, Q = 0`
- 16B when `size = 00, Q = 1`
- 4H when `size = 01, Q = 0`
- 8H when `size = 01, Q = 1`

The encoding `size = 1x, Q = x` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

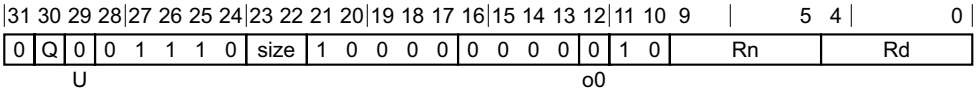
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.195 REV64

Reverse elements in 64-bit doublewords (vector) reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector, and places the result in the corresponding destination vector.




### *Vector variant*

```
REV64 <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op)+UInt(size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

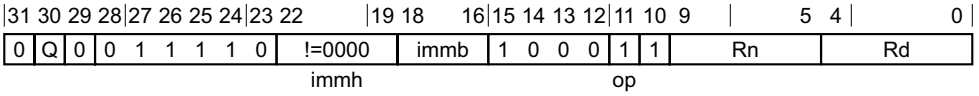
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.196 RSHRN, RSHRN2

Rounding shift right narrow (immediate)




### *Vector variant*

```
RSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- `8B` when `immh = 0001`, `Q = 0`
- `16B` when `immh = 0001`, `Q = 1`
- `4H` when `immh = 001x`, `Q = 0`
- `8H` when `immh = 001x`, `Q = 1`
- `2S` when `immh = 01xx`, `Q = 0`
- `4S` when `immh = 01xx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = x` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

- `8H` when `immh = 0001`
- `4S` when `immh = 001x`
- `2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding immh = 1xxx is reserved.

<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

The encoding immh = 1xxx is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

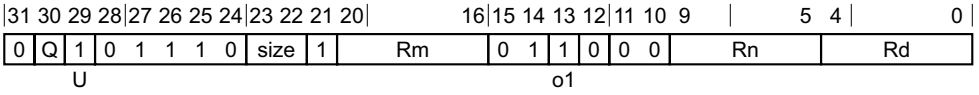
for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.197 RSUBHN, RSUBHN2

Rounding subtract returning high narrow




### *Three registers, not all the same type variant*

```
RSUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize)   result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

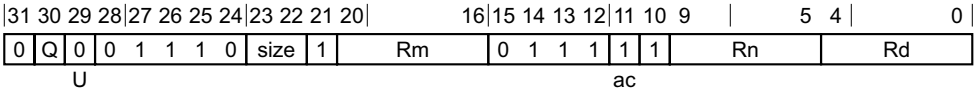
for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.198 SABA

Signed absolute difference and accumulate




### *Three registers of the same type variant*

```
SABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
```

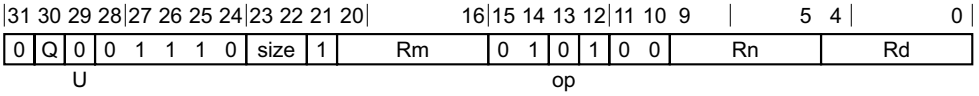

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.199 SABAL, SABAL2

Signed absolute difference and accumulate long




### *Three registers, not all the same type variant*

```
SABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

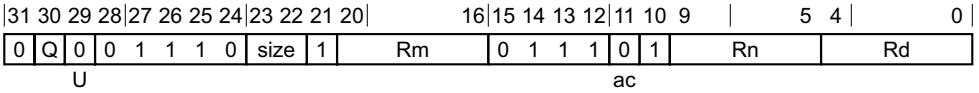
```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.200 SABD

Signed absolute difference




### *Three registers of the same type variant*

```
SABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when size = 00, Q = 0 |
| 16B | when size = 00, Q = 1 |
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
```

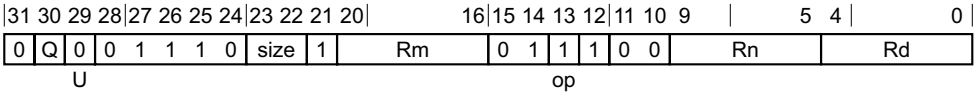

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.201 SABDL, SABDL2

Signed absolute difference long




### *Three registers, not all the same type variant*

```
SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

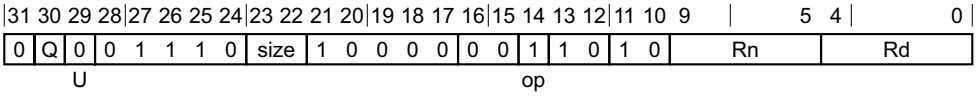
```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.202 SADALP

Signed add and accumulate long pairwise




### *Vector variant*

```
SADALP <Vd>.<Ta>, <Vn>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 4H when size = 00, Q = 0
- 8H when size = 00, Q = 1
- 2S when size = 01, Q = 0
- 4S when size = 01, Q = 1
- 1D when size = 10, Q = 0
- 2D when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(2*esize) sum;
integer op1;
integer op2;

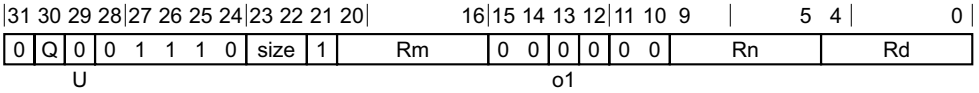
result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.203 SADDL, SADDL2

Signed add long (vector)




### *Three registers, not all the same type variant*

```
SADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

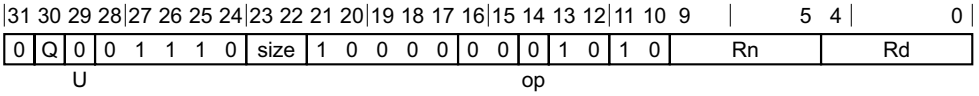
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.204 SADDLP

Signed add long pairwise




### *Vector variant*

```
SADDLP <Vd>.<Ta>, <Vn>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 4H when size = 00, Q = 0
- 8H when size = 00, Q = 1
- 2S when size = 01, Q = 0
- 4S when size = 01, Q = 1
- 1D when size = 10, Q = 0
- 2D when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(2*esize) sum;
integer op1;
integer op2;

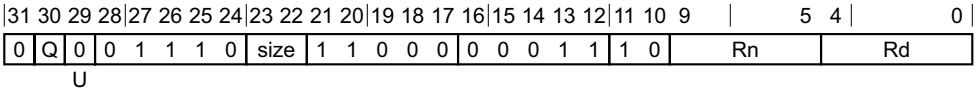
result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.205 SADDLV

Signed add long across vector




### *Advanced SIMD variant*

```
SADDLV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

## Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- H when size = 00
- S when size = 01
- D when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

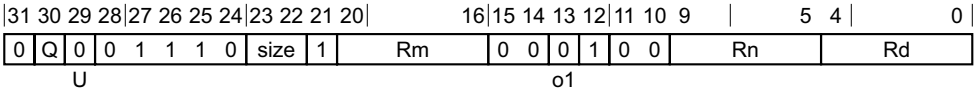
```
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d] = sum<2*esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.206 SADDW, SADDW2

Signed add wide




### *Three registers, not all the same type variant*

```
SADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> — Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `size = 11, Q = x` is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

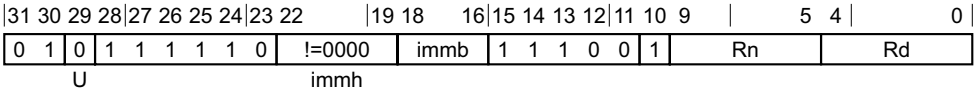
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.207 SCVTF (vector, fixed-point)

Signed fixed-point convert to floating-point (vector)

### Scalar




#### *Scalar variant*

```
SCVTF <V><d>, <V><n>, #<fbits>
```

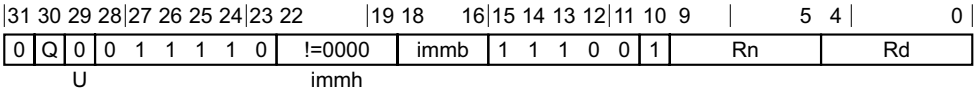
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '00xx' then ReservedValue();
integer esize = 32 << UInt(immh<3>);
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

### Vector




#### *Vector variant*

```
SCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh == '00xx' then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = 32 << UInt(immh<3>);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

S when `immh = 01xx`

D when `immh = 1xxx`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `immh = 00xx` is reserved.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

`2D` when `immh = 1xxx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`, `Q = x`.

The following encodings are reserved:

- `immh = 0001`, `Q = x`.
- `immh = 001x`, `Q = x`.
- `immh = 1xxx`, `Q = 0`.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<fbits>` For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immh:immb" field. It can have the following values:

`(64-UInt(immh:immb))` when `immh = 01xx`

`(128-UInt(immh:immb))` when `immh = 1xxx`

The encoding `immh = 00xx` is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immh:immb" field. It can have the following values:

`(64-UInt(immh:immb))` when `immh = 01xx`

`(128-UInt(immh:immb))` when `immh = 1xxx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The following encodings are reserved:

- `immh = 0001`.
- `immh = 001x`.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding);

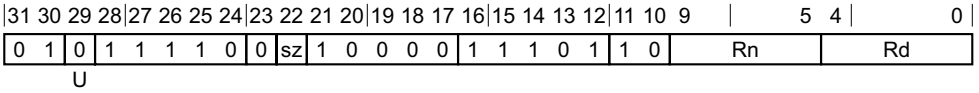
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.208 SCVTF (vector, integer)

Signed integer convert to floating-point (vector)

### Scalar




#### Scalar variant

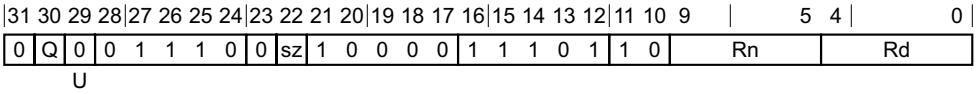
```
SCVTF <V><d>, <V><n>
```

#### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Vector




#### Vector variant

```
SCVTF <Vd>.<T>, <Vn>.<T>
```

#### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when `sz = 0`

D when `sz = 1`

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when `sz = 0, Q = 0`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
FPRounding rounding = FPRoundingMode(FPCR);
bits(esize) element;

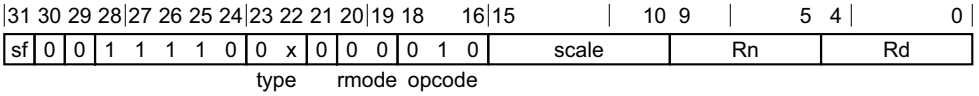
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.209 SCVTF (scalar, fixed-point)

Signed fixed-point convert to floating-point (scalar) : `Vd = signed_convertFromInt(Rn/(2^fbits))`




### 32-bit to single-precision variant

Applies when `sf = 0 && type = 00`.

```
SCVTF <Sd>, <Wn>, #<fbits>
```

### 32-bit to double-precision variant

Applies when `sf = 0 && type = 01`.

```
SCVTF <Dd>, <Wn>, #<fbits>
```

### 64-bit to single-precision variant

Applies when `sf = 1 && type = 00`.

```
SCVTF <Sd>, <Xn>, #<fbits>
```

### 64-bit to double-precision variant

Applies when `sf = 1 && type = 01`.

```
SCVTF <Dd>, <Xn>, #<fbits>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '1x' UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
    when '00 11'        // FCVTZ
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    otherwise
        UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<fbits> For the 32-bit to double-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".

For the 64-bit to double-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

### Operation

```
CheckFPAdvSIMDEnabled64();

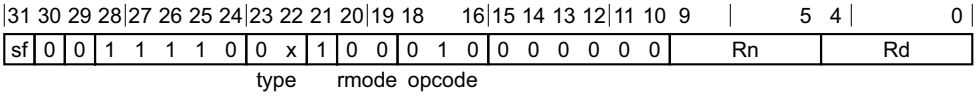
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
        V[d] = fltval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.210 SCVTF (scalar, integer)

Signed integer convert to floating-point (scalar) : `Vd = signed_convertFromInt(Rn)`




### *32-bit to single-precision variant*

Applies when `sf = 0 && type = 00`.

```
SCVTF <Sd>, <Wn>
```

### *32-bit to double-precision variant*

Applies when `sf = 0 && type = 01`.

```
SCVTF <Dd>, <Wn>
```

### *64-bit to single-precision variant*

Applies when `sf = 1 && type = 00`.

```
SCVTF <Sd>, <Xn>
```

### *64-bit to double-precision variant*

Applies when `sf = 1 && type = 01`.

```
SCVTF <Dd>, <Xn>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

| | |
|---|---|
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field. |
| <Wn> | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field. |

## Operation

```
CheckFPAdvSIMDEnabled64();

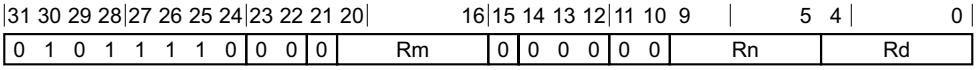
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.211 SHA1C

SHA1 hash update (choose)




### *Advanced SIMD variant*

```
SHA1C <Qd>, <Sn>, <Vm>.4S
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if ! HaveCryptoExt() then UnallocatedEncoding();
```

### Assembler symbols

| | |
|---|---|
| <Qd> | Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field. |
| <Vm> | Is the name of the third SIMD&FP source register, encoded in the "Rm" field. |

### Operation

```
CheckCryptoEnabled64();

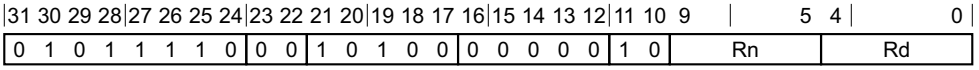
bits(128) X = V[d];
bits(32)  Y = V[n];     // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32)  t;

for e = 0 to 3
    t = SHAchoose(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y : X, 32);
V[d] = X;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.212 SHA1H

SHA1 fixed rotate




### *Advanced SIMD variant*

```
SHA1H <Sd>, <Sn>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
```

### Assembler symbols

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

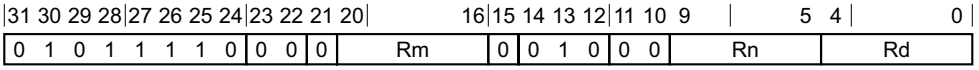
```
CheckCryptoEnabled64();

bits(32) operand = V[n];        // read element [0] only,  [1-3] zeroed
V[d] = ROL(operand, 30);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.213 SHA1M

SHA1 hash update (majority)




### *Advanced SIMD variant*

```
SHA1M <Qd>, <Sn>, <Vm>.4S
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if ! HaveCryptoExt() then UnallocatedEncoding();
```

### Assembler symbols

| | |
|---|---|
| <Qd> | Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field. |
| <Vm> | Is the name of the third SIMD&FP source register, encoded in the "Rm" field. |

### Operation

```
CheckCryptoEnabled64();

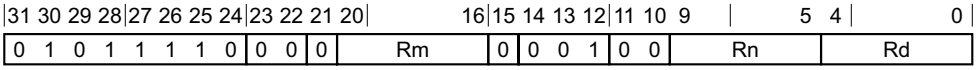
bits(128) X = V[d];
bits(32)  Y = V[n];     // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32)  t;

for e = 0 to 3
    t = SHAmajority(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y : X, 32);
V[d] = X;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.214 SHA1P

SHA1 hash update (parity)




### *Advanced SIMD variant*

```
SHA1P <Qd>, <Sn>, <Vm>.4S
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if ! HaveCryptoExt() then UnallocatedEncoding();
```

### Assembler symbols

| | |
|---|---|
| <Qd> | Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field. |
| <Sn> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field. |
| <Vm> | Is the name of the third SIMD&FP source register, encoded in the "Rm" field. |

### Operation

```
CheckCryptoEnabled64();

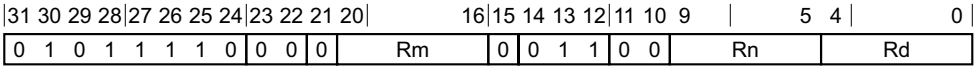
bits(128) X = V[d];
bits(32)  Y = V[n];     // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32)  t;

for e = 0 to 3
    t = SHAparity(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y : X, 32);
V[d] = X;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.215 SHA1SU0

SHA1 schedule update 0




### *Advanced SIMD variant*

```
SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if ! HaveCryptoExt() then UnallocatedEncoding();
```

### Assembler symbols

| | |
|---|---|
| <Vd> | Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field. |
| <Vn> | Is the name of the second SIMD&FP source register, encoded in the "Rn" field. |
| <Vm> | Is the name of the third SIMD&FP source register, encoded in the "Rm" field. |

### Operation

```
CheckCryptoEnabled64();

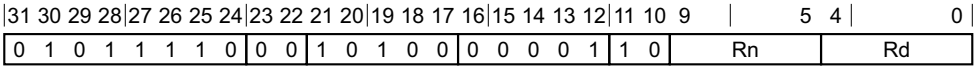
bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;

result = operand2<63:0> : operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.216 SHA1SU1

SHA1 schedule update 1




### *Advanced SIMD variant*

```
SHA1SU1 <Vd>.4S, <Vn>.4S
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

### Operation

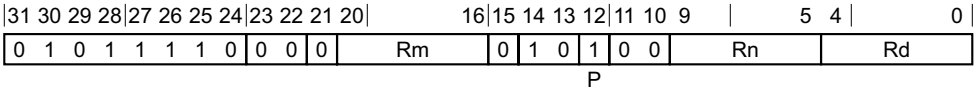
```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand1 EOR LSR(operand2, 32);
result<31:0>   = ROL(T<31:0>,   1);
result<63:32>  = ROL(T<63:32>,  1);
result<95:64>  = ROL(T<95:64>,  1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.217 SHA256H2

SHA256 hash update (part 2)




### *Advanced SIMD variant*

```
SHA256H2 <Qd>, <Qn>, <Vm>.4S
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if ! HaveCryptoExt() then UnallocatedEncoding();
boolean part1 = (P == '0');
```

### Assembler symbols

| | |
|---|---|
| <Qd> | Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field. |
| <Qn> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field. |
| <Vm> | Is the name of the third SIMD&FP source register, encoded in the "Rm" field. |

### Operation

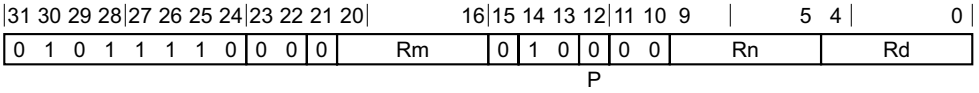
```
CheckCryptoEnabled64();

bits(128) result;
if part1 then
    result = SHA256hash(V[d], V[n], V[m], TRUE);
else
    result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.218 SHA256H

SHA256 hash update (part 1)




### *Advanced SIMD variant*

```
SHA256H <Qd>, <Qn>, <Vm>.4S
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if ! HaveCryptoExt() then UnallocatedEncoding();
boolean part1 = (P == '0');
```

### Assembler symbols

| | |
|---|---|
| <Qd> | Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field. |
| <Qn> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field. |
| <Vm> | Is the name of the third SIMD&FP source register, encoded in the "Rm" field. |

### Operation

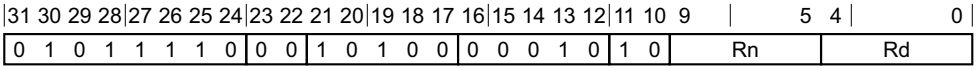
```
CheckCryptoEnabled64();

bits(128) result;
if part1 then
    result = SHA256hash(V[d], V[n], V[m], TRUE);
else
    result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.219 SHA256SU0

SHA256 schedule update 0




### *Advanced SIMD variant*

```
SHA256SU0 <Vd>.4S, <Vn>.4S
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckCryptoEnabled64();

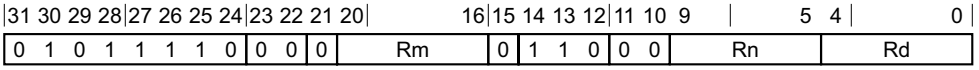
bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand2<31:0> : operand1<127:32>;
bits(32) elt;

for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.220 SHA256SU1

SHA256 schedule update 1




### *Advanced SIMD variant*

```
SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if ! HaveCryptoExt() then UnallocatedEncoding();
```

### Assembler symbols

| | |
|---|---|
| <Vd> | Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field. |
| <Vn> | Is the name of the second SIMD&FP source register, encoded in the "Rn" field. |
| <Vm> | Is the name of the third SIMD&FP source register, encoded in the "Rm" field. |

### Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
bits(128) T0 = operand3<31:0> : operand2<127:32>;
bits(64) T1;
bits(32) elt;

T1 = operand3<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

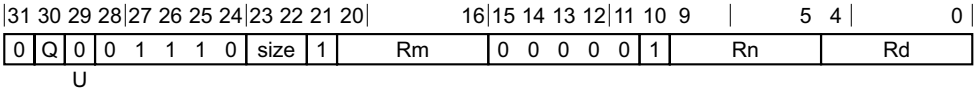
T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e - 2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.221 SHADD

Signed halving add




### *Three registers of the same type variant*

```
SHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;

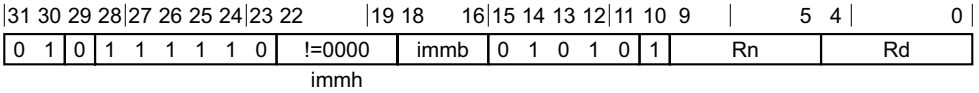
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.222 SHL

Shift left (immediate)

### Scalar




#### *Scalar variant*

```
SHL <V><d>, <V><n>, #<shift>
```

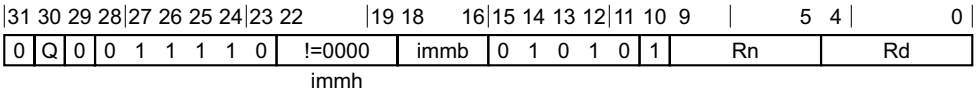
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

### Vector




#### *Vector variant*

```
SHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in the "immh:immb" field. It can have the following values:

(UInt(immh:immb)-64) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

(UInt(immh:immb)-8) when immh = 0001

(UInt(immh:immb)-16) when immh = 001x

(UInt(immh:immb)-32) when immh = 01xx

(UInt(immh:immb)-64) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) result;

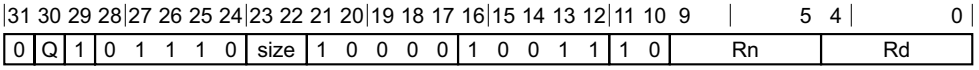
for e = 0 to elements-1
    Elem[result, e, esize] = LSL(Elem[operand, e, esize], shift);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.223 SHLL, SHLL2

Shift left long (by element size)




### *Vector variant*

```
SHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = esize;
boolean unsigned = FALSE; // Or TRUE without change of functionality
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

<Vn> — Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00, Q = 0`
- `16B` when `size = 00, Q = 1`
- `4H` when `size = 01, Q = 0`
- `8H` when `size = 01, Q = 1`
- `2S` when `size = 10, Q = 0`
- `4S` when `size = 10, Q = 1`

The encoding `size = 11, Q = x` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<shift> Is the left shift amount, which must be equal to the source element width in bits, encoded in the "size" field. It can have the following values:

8 when size = 00

16 when size = 01

32 when size = 10

The encoding size = 11 is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;
integer element;

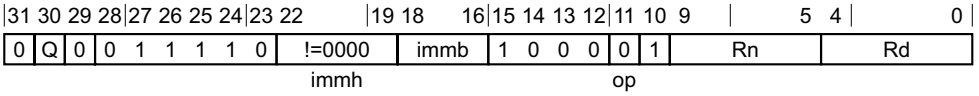
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.224 SHRN, SHRN2

Shift right narrow (immediate)




### *Vector variant*

```
SHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when immh = 0001

4S when immh = 001x

2D when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `immh = 1xxx` is reserved.

`<shift>` Is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

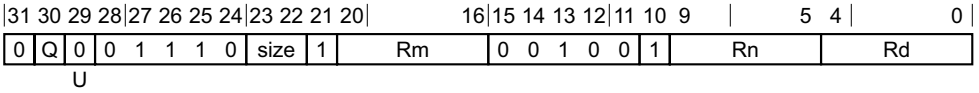
for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.225 SHSUB

Signed halving subtract




### *Three registers of the same type variant*

```
SHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;

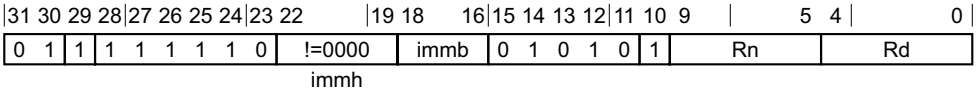
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.226 SLI

Shift left and insert (immediate)

### Scalar




#### *Scalar variant*

```
SLI <V><d>, <V><n>, #<shift>
```

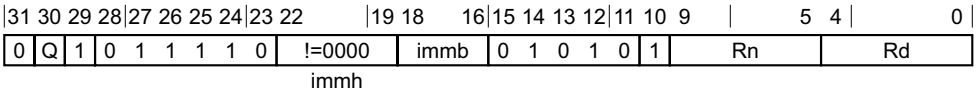
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

### Vector




#### *Vector variant*

```
SLI <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

### Assembler symbols

| | |
|---|---|
| <V> | Is a width specifier, encoded in the "immh" field. It can have the following values:<br>D when `immh = 1xxx`<br>The encoding `immh = 0xxx` is reserved. |
| <d> | Is the number of the SIMD&FP destination register, in the "Rd" field. |
| <n> | Is the number of the first SIMD&FP source register, encoded in the "Rn" field. |
| <Vd> | Is the name of the SIMD&FP destination register, encoded in the "Rd" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in the "immh:immb" field. It can have the following values:

(UInt(immh:immb)-64) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

(UInt(immh:immb)-8) when immh = 0001

(UInt(immh:immb)-16) when immh = 001x

(UInt(immh:immb)-32) when immh = 01xx

(UInt(immh:immb)-64) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

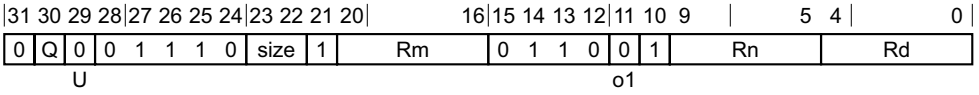
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
bits(esize) mask = LSL(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSL(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.227 SMAX

Signed maximum (vector)




### *Three registers of the same type variant*

```
SMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

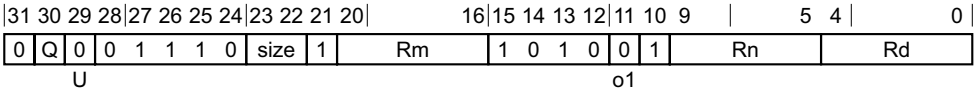
```
        Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.228 SMAXP

Signed maximum pairwise




### *Three registers of the same type variant*

```
SMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when size = 00, Q = 0 |
| 16B | when size = 00, Q = 1 |
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

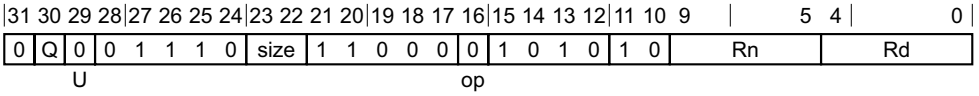
```
        Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.229 SMAXV

Signed maximum across vector




### *Advanced SIMD variant*

```
SMAXV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

## Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

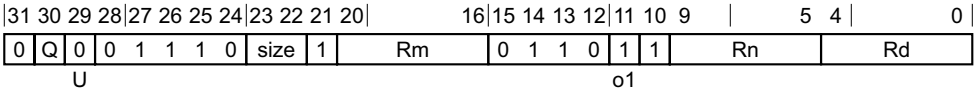
```
maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.230 SMIN

Signed minimum (vector)




### *Three registers of the same type variant*

```
SMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when size = 00, Q = 0 |
| 16B | when size = 00, Q = 1 |
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

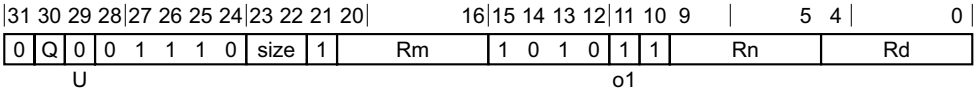
```
        Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.231 SMINP

Signed minimum pairwise




### *Three registers of the same type variant*

```
SMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

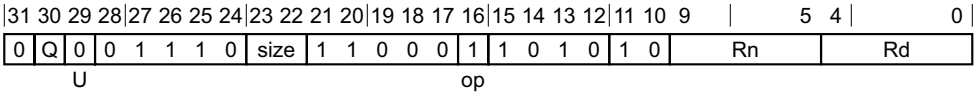
```
        Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.232 SMINV

Signed minimum across vector




### *Advanced SIMD variant*

```
SMINV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

## Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

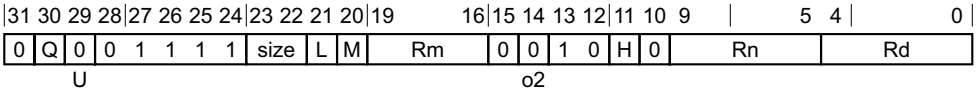
```
maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.233 SMLAL, SMLAL2 (by element)

Signed multiply-add long (vector, by element)




### *Vector variant*

```
SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `4S` when `size = 01`
- `2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `4H` when `size = 01`, `Q = 0`
- `8H` when `size = 01`, `Q = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(idxdsize)   operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

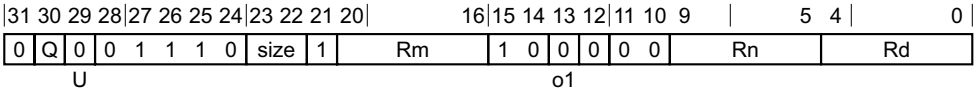
element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.234 SMLAL, SMLAL2 (vector)

Signed multiply-add long (vector)




### *Three registers, not all the same type variant*

```
SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00, Q = 0`
- `16B` when `size = 00, Q = 1`
- `4H` when `size = 01, Q = 0`
- `8H` when `size = 01, Q = 1`
- `2S` when `size = 10, Q = 0`
- `4S` when `size = 10, Q = 1`

The encoding `size = 11, Q = x` is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

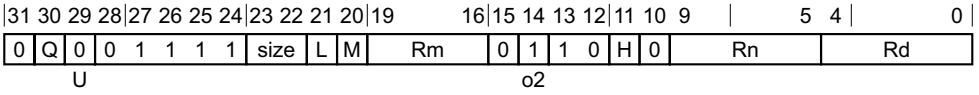
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.235 SMLSL, SMLSL2 (by element)

Signed multiply-subtract long (vector, by element)




### *Vector variant*

```
SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `4S` when `size = 01`
- `2D` when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `4H` when `size = 01`, `Q = 0`
- `8H` when `size = 01`, `Q = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(idxdsize)   operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

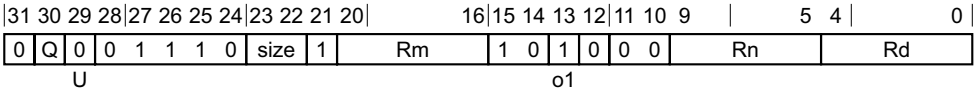
element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.236 SMLSL, SMLSL2 (vector)

Signed multiply-subtract long (vector)




### *Three registers, not all the same type variant*

```
SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> — Is the name of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

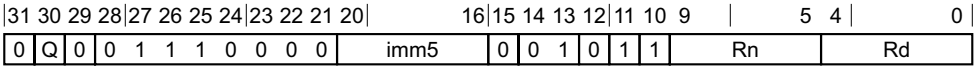
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.237 SMOV

Signed move vector element to general-purpose register




### 32-bit variant

Applies when `Q = 0`.

```
SMOV <Wd>, <Vn>.<Ts>[<index>]
```

### 64-bit variant

Applies when `Q = 1`.

```
SMOV <Xd>, <Vn>.<Ts>[<index>]
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when 'xxxxx1' size = 0;     // SMOV [WX]d, Vn.B
    when 'xxxx10' size = 1;     // SMOV [WX]d, Vn.H
    when '1xx100' size = 2;     // SMOV Xd, Vn.S
    otherwise     UnallocatedEncoding();

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

## Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

B when `imm5 = xxxx1`

H when `imm5 = xxx10`

The encoding `imm5 = xxx00` is reserved.

For the 64-bit variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

B when `imm5 = xxxx1`

H when `imm5 = xxx10`

S when `imm5 = xx100`

The encoding `imm5 = xx000` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<index> For the 32-bit variant: is the element index encoded in the "imm5" field. It can have the following values:

`imm5<4:1>` when `imm5 = xxxx1`

`imm5<4:2>` when `imm5 = xxx10`

The encoding `imm5 = xxx00` is reserved.

For the 64-bit variant: is the element index encoded in the "imm5" field. It can have the following values:

`imm5<4:1>` when `imm5 = xxxx1`

`imm5<4:2>` when `imm5 = xxx10`

`imm5<4:3>` when `imm5 = xx100`

The encoding `imm5 = xx000` is reserved.

## Operation

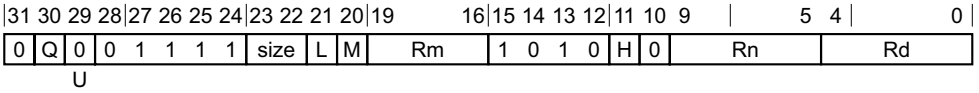
```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];

X[d] = SignExtend(Elem[operand, index, esize], datasize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.238 SMULL, SMULL2 (by element)

Signed multiply long (vector, by element)




### *Vector variant*

```
SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 4S when size = 01
- 2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(idxdsize)   operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

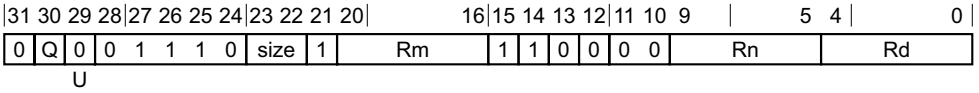
element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.239 SMULL, SMULL2 (vector)

Signed multiply long (vector)




### *Three registers, not all the same type variant*

```
SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> — Is the name of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1 * element2)<2*esize-1:0>;

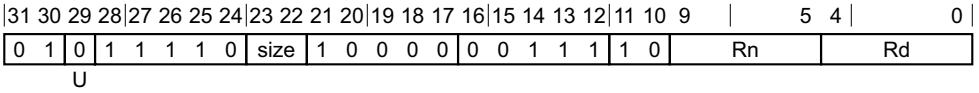
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.240 SQABS

Signed saturating absolute value

### Scalar




#### *Scalar variant*

```
SQABS <V><d>, <V><n>
```

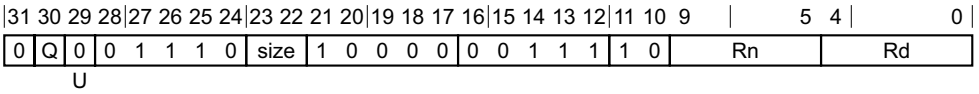
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean neg = (U == '1');
```

### Vector




#### *Vector variant*

```
SQABS <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean neg = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| B | when size = 00 |
| H | when size = 01 |
| S | when size = 10 |
| D | when size = 11 |

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

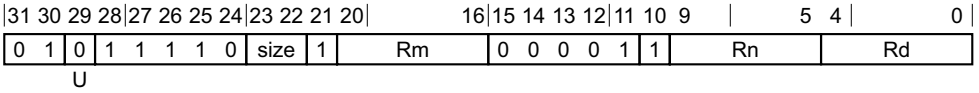
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.241 SQADD

Signed saturating add

### Scalar



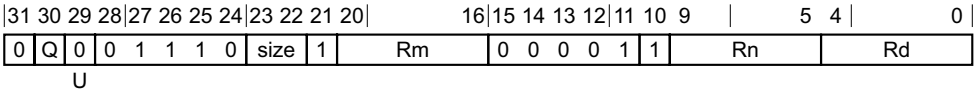

#### *Scalar variant*

```
SQADD <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
SQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

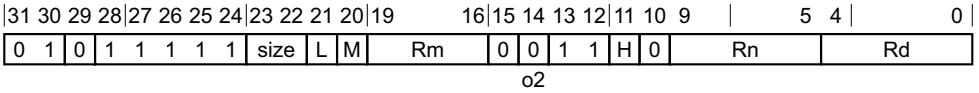
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.242 SQDMLAL, SQDMLAL2 (by element)

Signed saturating doubling multiply-add long (by element)

### Scalar




#### *Scalar variant*

```
SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

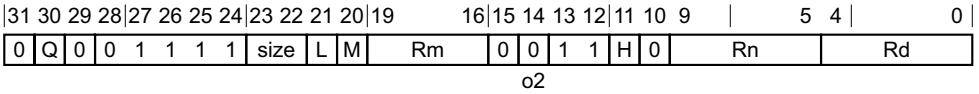
```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

### Vector




#### *Vector variant*

```
SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

4S when size = 01

2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Va> Is the destination width specifier, encoded in the "size" field. It can have the following values:

S when size = 01

D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> For the scalar variant: is the element width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

For the vector variant: is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(idxdsize)   operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

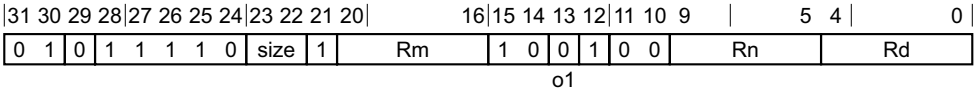
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.243 SQDMLAL, SQDMLAL2 (vector)

Signed saturating doubling multiply-add long

### Scalar




#### *Scalar variant*

```
SQDMLAL <Va><d>, <Vb><n>, <Vb><m>
```

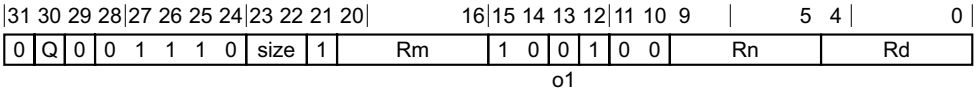
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

### Vector




#### *Vector variant*

```
SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

4S when size = 01

2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in the "size" field. It can have the following values:

S when size = 01

D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

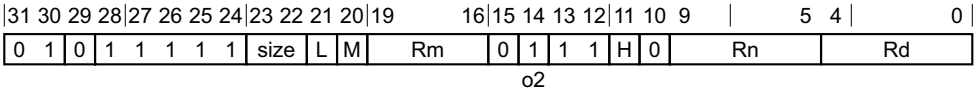
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.244 SQDMLSL, SQDMLSL2 (by element)

Signed saturating doubling multiply-subtract long (by element)

### Scalar




#### *Scalar variant*

```
SQDMLSL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

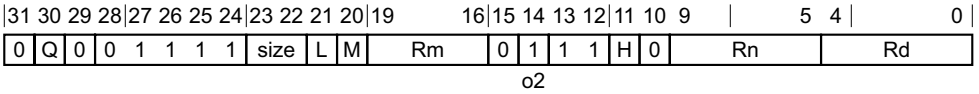
```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

### Vector




#### *Vector variant*

```
SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

4S when size = 01

2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Va> Is the destination width specifier, encoded in the "size" field. It can have the following values:

S when size = 01

D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> For the scalar variant: is the element width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

For the vector variant: is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(idxdsize)   operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

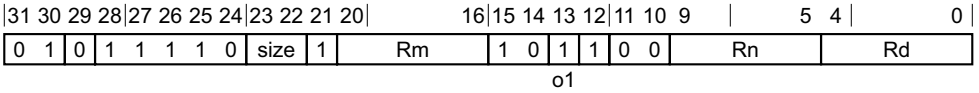
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.245 SQDMLSL, SQDMLSL2 (vector)

Signed saturating doubling multiply-subtract long

### Scalar




#### *Scalar variant*

```
SQDMLSL <Va><d>, <Vb><n>, <Vb><m>
```

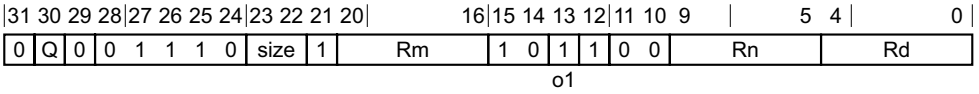
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

### Vector




#### *Vector variant*

```
SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

4S when size = 01

2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in the "size" field. It can have the following values:

S when size = 01

D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

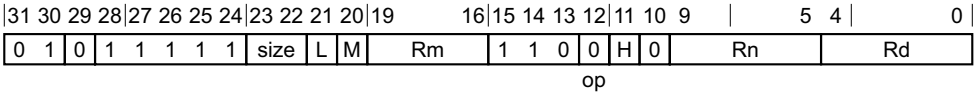
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.246 SQDMULH (by element)

Signed saturating doubling multiply returning high half (by element)

### Scalar




#### *Scalar variant*

```
SQDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

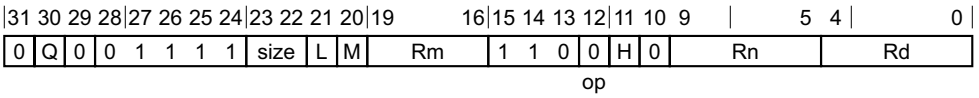
```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

### Vector




#### *Vector variant*

```
SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> For the scalar variant: is the element width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

For the vector variant: is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

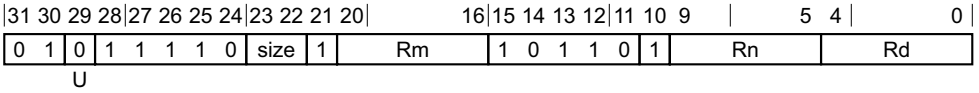
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.247 SQDMULH (vector)

Signed saturating doubling multiply returning high half

### Scalar



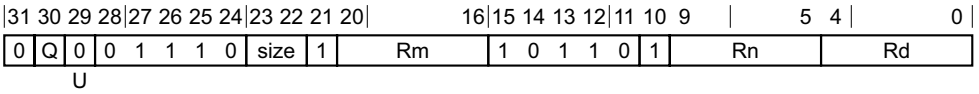

#### *Scalar variant*

```
SQDMULH <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

### Vector




#### *Vector variant*

```
SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when `size = 01`

S when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

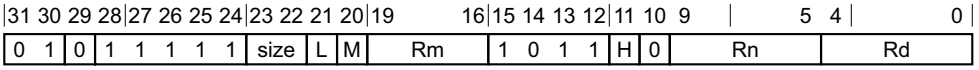
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.248 SQDMULL, SQDMULL2 (by element)

Signed saturating doubling multiply long (by element)

### Scalar




#### *Scalar variant*

```
SQDMULL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

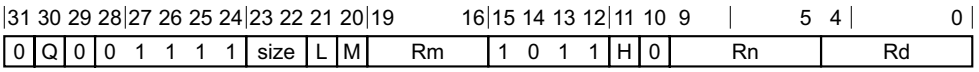
#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

### Vector




#### *Vector variant*

```
SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

4S when size = 01

2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Va> Is the destination width specifier, encoded in the "size" field. It can have the following values:

S when size = 01

D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> For the scalar variant: is the element width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

For the vector variant: is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(datasize)   operand1 = Vpart[n, part];
bits(idxdsize)   operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2*esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

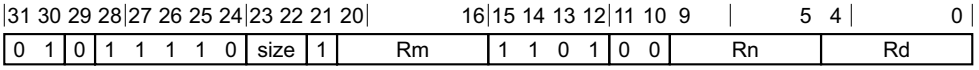
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.249 SQDMULL, SQDMULL2 (vector)

Signed saturating doubling multiply long

### Scalar




#### *Scalar variant*

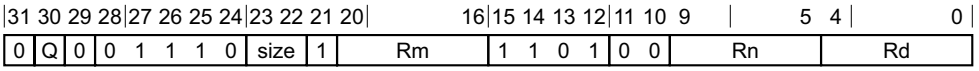
```
SQDMULL <Va><d>, <Vb><n>, <Vb><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

### Vector




#### *Vector variant*

```
SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

4S when size = 01

2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in the "size" field. It can have the following values:

S when size = 01

D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2*esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            Elem[result, e, 2*esize] = product;
            if sat then FPSR.QC = '1';

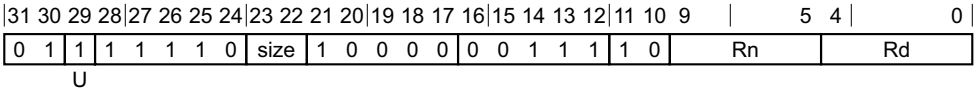
        V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.250 SQNEG

Signed saturating negate

### Scalar




#### *Scalar variant*

```
SQNEG <V><d>, <V><n>
```

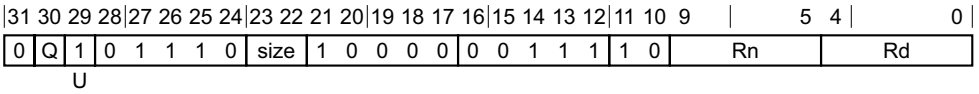
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean neg = (U == '1');
```

### Vector




#### *Vector variant*

```
SQNEG <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean neg = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| B | when size = 00 |
| H | when size = 01 |
| S | when size = 10 |
| D | when size = 11 |

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

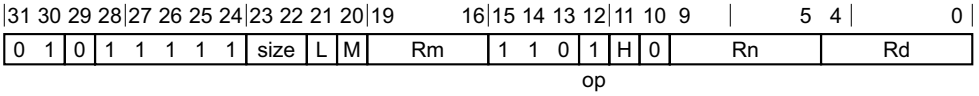
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.251 SQRDMULH (by element)

Signed saturating rounding doubling multiply returning high half (by element)

### Scalar




#### *Scalar variant*

```
SQRDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

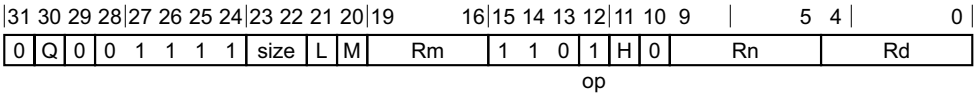
```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

### Vector




#### *Vector variant*

```
SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

#### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> For the scalar variant: is the element width specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

For the vector variant: is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

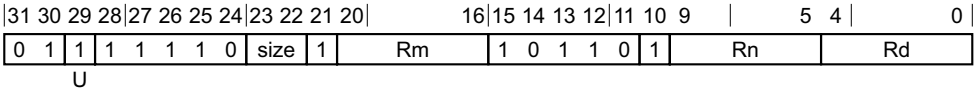
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.252 SQRDMULH (vector)

Signed saturating rounding doubling multiply returning high half

### Scalar



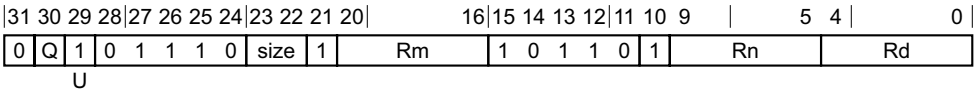

#### *Scalar variant*

```
SQRDMULH <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

### Vector




#### *Vector variant*

```
SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

H when `size = 01`

S when `size = 10`

The following encodings are reserved:

- `size = 00`.
- `size = 11`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

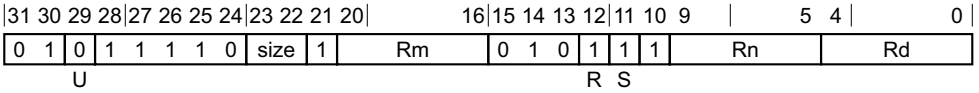
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.253 SQRSHL

Signed saturating rounding shift left (register)

### Scalar



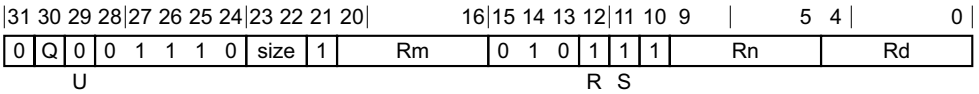

#### *Scalar variant*

```
SQRSHL <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

### Vector




#### *Vector variant*

```
SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

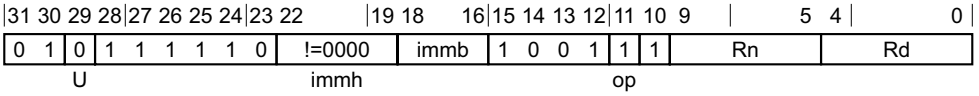
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.254 SQRSHRN, SQRSHRN2

Signed saturating rounded shift right narrow (immediate)

### Scalar




#### *Scalar variant*

```
SQRSHRN <Vb><d>, <Va><n>, #<shift>
```

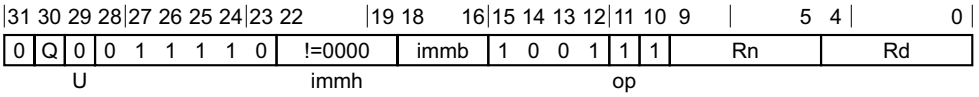
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
SQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when immh = 0001

4S when immh = 001x

2D when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vb> Is the destination width specifier, encoded in the "immh" field. It can have the following values:

B when immh = 0001

H when immh = 001x

S when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 0001

S when immh = 001x

D when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

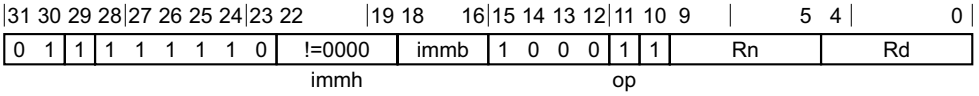
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.255 SQRSHRUN, SQRSHRUN2

Signed saturating rounded shift right unsigned narrow (immediate)

### Scalar




#### *Scalar variant*

```
SQRSHRUN <Vb><d>, <Va><n>, #<shift>
```

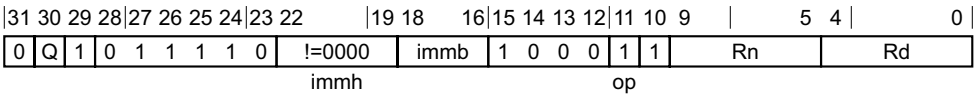
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Vector




#### *Vector variant*

```
SQRSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when immh = 0001

4S when immh = 001x

2D when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vb> Is the destination width specifier, encoded in the "immh" field. It can have the following values:

B when immh = 0001

H when immh = 001x

S when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 0001

S when immh = 001x

D when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

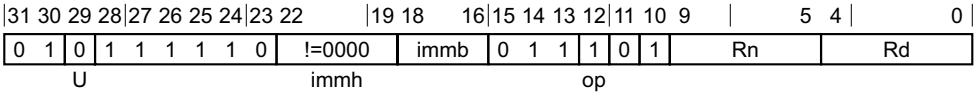
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.256 SQSHL (immediate)

Signed saturating shift left (immediate)

### Scalar




#### *Scalar variant*

```
SQSHL <V><d>, <V><n>, #<shift>
```

#### *Decode for this encoding*

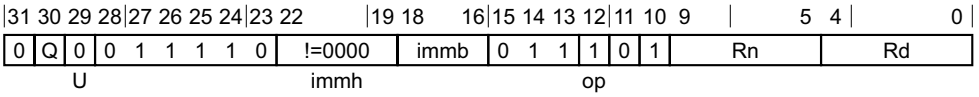
```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE;  dst_unsigned = TRUE;
```

### Vector




#### *Vector variant*

```
SQSHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
when '11' src_unsigned = TRUE;  dst_unsigned = TRUE;
```

## Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

B when immh = 0001

H when immh = 001x

S when immh = 01xx

D when immh = 1xxx

The encoding immh = 0000 is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

(UInt(immh:immb)-8) when immh = 0001

(UInt(immh:immb)-16) when immh = 001x

(UInt(immh:immb)-32) when immh = 01xx

(UInt(immh:immb)-64) when immh = 1xxx

The encoding immh = 0000 is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

(UInt(immh:immb)-8) when immh = 0001

(UInt(immh:immb)-16) when immh = 001x

(UInt(immh:immb)-32) when immh = 01xx

(UInt(immh:immb)-64) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

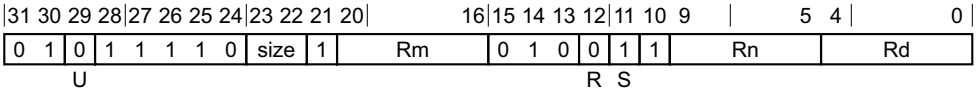
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.257 SQSHL (register)

Signed saturating shift left (register)

### Scalar



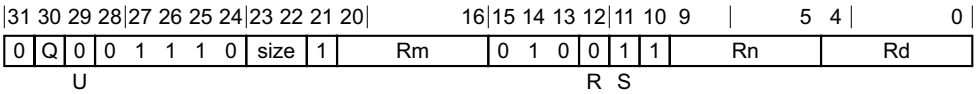

#### *Scalar variant*

```
SQSHL <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

### Vector




#### *Vector variant*

```
SQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| B | when size = 00 |
| H | when size = 01 |
| S | when size = 10 |
| D | when size = 11 |

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

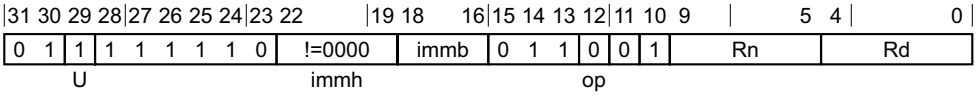
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.258 SQSHLU

Signed saturating shift left unsigned (immediate)

### Scalar




#### *Scalar variant*

```
SQSHLU <V><d>, <V><n>, #<shift>
```

#### *Decode for this encoding*

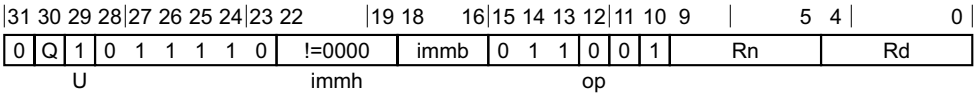
```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE;  dst_unsigned = TRUE;
```

### Vector




#### *Vector variant*

```
SQSHLU <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
when '11' src_unsigned = TRUE;  dst_unsigned = TRUE;
```

## Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

- `B` when `immh = 0001`
- `H` when `immh = 001x`
- `S` when `immh = 01xx`
- `D` when `immh = 1xxx`

The encoding `immh = 0000` is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- `8B` when `immh = 0001`, `Q = 0`
- `16B` when `immh = 0001`, `Q = 1`
- `4H` when `immh = 001x`, `Q = 0`
- `8H` when `immh = 001x`, `Q = 1`
- `2S` when `immh = 01xx`, `Q = 0`
- `4S` when `immh = 01xx`, `Q = 1`
- `2D` when `immh = 1xxx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = 0` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

- `(UInt(immh:immb)-8)` when `immh = 0001`
- `(UInt(immh:immb)-16)` when `immh = 001x`
- `(UInt(immh:immb)-32)` when `immh = 01xx`
- `(UInt(immh:immb)-64)` when `immh = 1xxx`

The encoding `immh = 0000` is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

- `(UInt(immh:immb)-8)` when `immh = 0001`
- `(UInt(immh:immb)-16)` when `immh = 001x`
- `(UInt(immh:immb)-32)` when `immh = 01xx`
- `(UInt(immh:immb)-64)` when `immh = 1xxx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

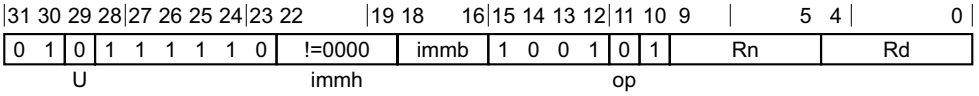
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.259 SQSHRN, SQSHRN2

Signed saturating shift right narrow (immediate)

### Scalar




#### *Scalar variant*

```
SQSHRN <Vb><d>, <Va><n>, #<shift>
```

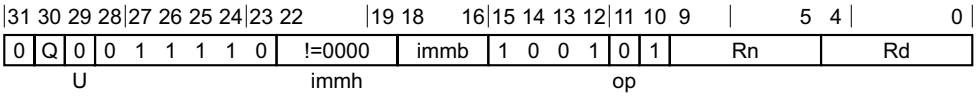
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
SQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when immh = 0001

4S when immh = 001x

2D when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vb> Is the destination width specifier, encoded in the "immh" field. It can have the following values:

B when immh = 0001

H when immh = 001x

S when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 0001

S when immh = 001x

D when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

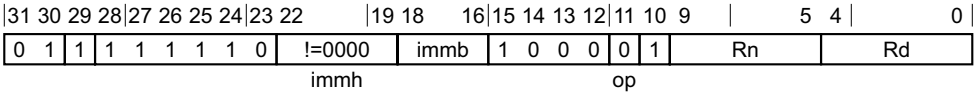
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.260 SQSHRUN, SQSHRUN2

Signed saturating shift right unsigned narrow (immediate)

### Scalar




#### Scalar variant

```
SQSHRUN <Vb><d>, <Va><n>, #<shift>
```

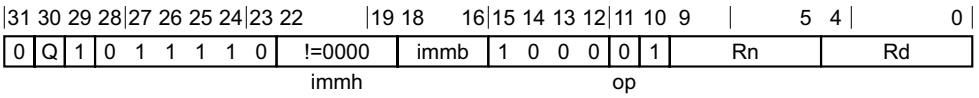
#### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Vector




#### Vector variant

```
SQSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

#### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when immh = 0001

4S when immh = 001x

2D when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vb> Is the destination width specifier, encoded in the "immh" field. It can have the following values:

B when immh = 0001

H when immh = 001x

S when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 0001

S when immh = 001x

D when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

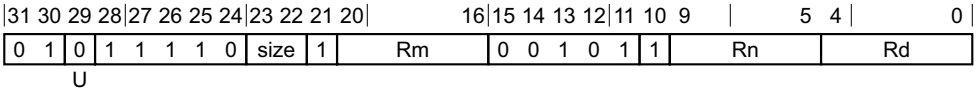
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.261 SQSUB

Signed saturating subtract

### Scalar



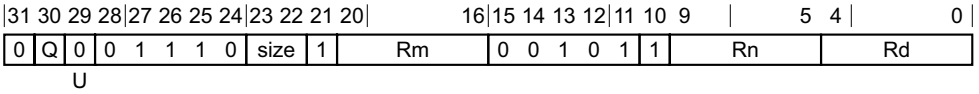

#### *Scalar variant*

```
SQSUB <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
SQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

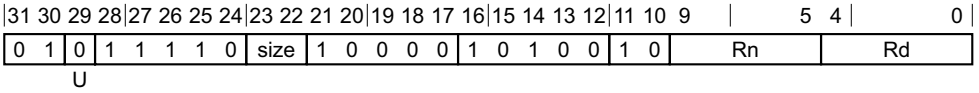
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.262 SQXTN, SQXTN2

Signed saturating extract narrow

### Scalar




#### *Scalar variant*

```
SQXTN <Vb><d>, <Va><n>
```

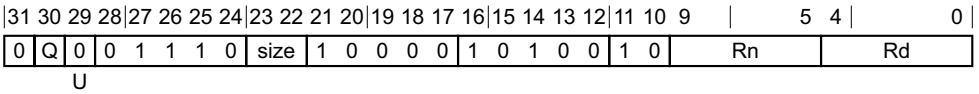
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
SQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler symbols

| | |
|---|---|
| 2 | Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:<br>`[absent]` when `Q = 0`<br>`[present]` when `Q = 1` |
| <Vd> | Is the name of the SIMD&FP destination register, encoded in the "Rd" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

8H when size = 00

4S when size = 01

2D when size = 10

The encoding size = 11 is reserved.

<Vb> Is the destination width specifier, encoded in the "size" field. It can have the following values:

B when size = 00

H when size = 01

S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 00

S when size = 01

D when size = 10

The encoding size = 11 is reserved.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

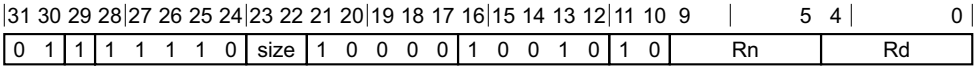
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.263 SQXTUN, SQXTUN2

Signed saturating extract unsigned narrow

### Scalar




#### *Scalar variant*

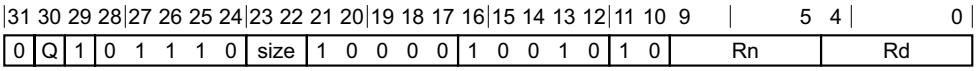
```
SQXTUN <Vb><d>, <Va><n>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;
```

### Vector




#### *Vector variant*

```
SQXTUN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00, Q = 0`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

8H when size = 00

4S when size = 01

2D when size = 10

The encoding size = 11 is reserved.

<Vb> Is the destination width specifier, encoded in the "size" field. It can have the following values:

B when size = 00

H when size = 01

S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 00

S when size = 01

D when size = 10

The encoding size = 11 is reserved.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

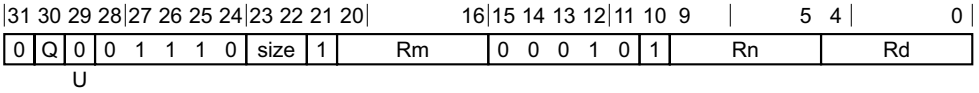
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.264 SRHADD

Signed rounding halving add




### *Three registers of the same type variant*

```
SRHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1 + element2 + 1)<esize:1>;

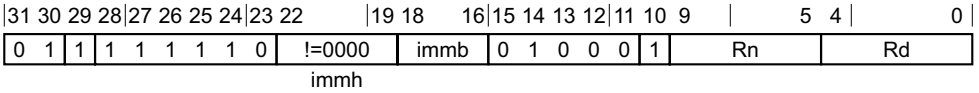
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.265 SRI

Shift right and insert (immediate)

### Scalar




#### *Scalar variant*

```
SRI <V><d>, <V><n>, #<shift>
```

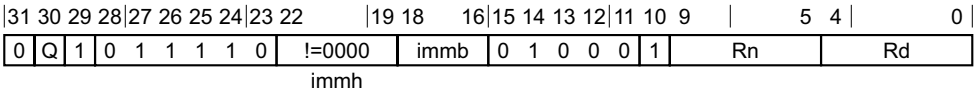
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
```

### Vector




#### *Vector variant*

```
SRI <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
bits(esize) mask = LSR(Ones(esize), shift);
bits(esize) shifted;

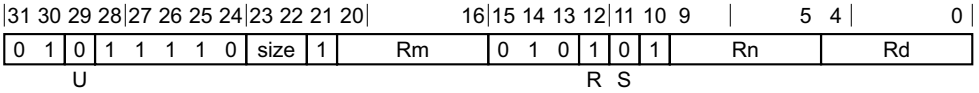
for e = 0 to elements-1
    shifted = LSR(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.266 SRSHL

Signed rounding shift left (register)

### Scalar



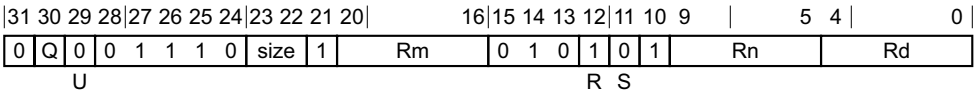

#### *Scalar variant*

```
SRSHL <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

### Vector




#### *Vector variant*

```
SRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

`D` when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

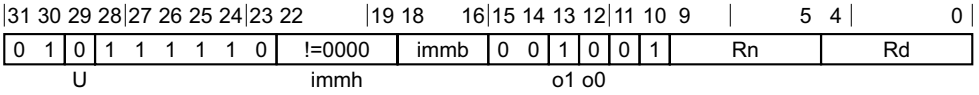
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.267 SRSHR

Signed rounding shift right (immediate)

### Scalar




#### *Scalar variant*

```
SRSHR <V><d>, <V><n>, #<shift>
```

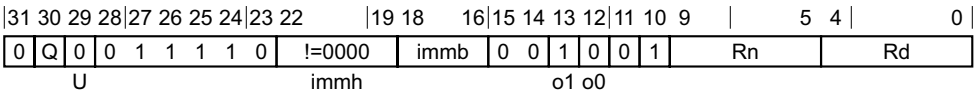
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector




#### *Vector variant*

```
SRSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

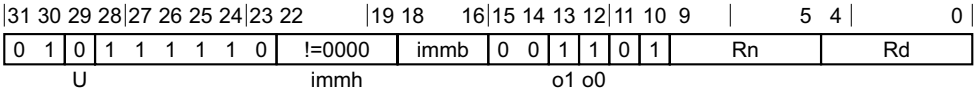
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.268 SRSRA

Signed rounding shift right and accumulate (immediate)

### Scalar




#### *Scalar variant*

```
SRSRA <V><d>, <V><n>, #<shift>
```

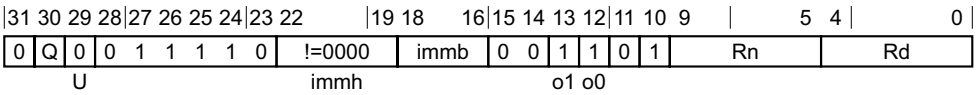
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector




#### *Vector variant*

```
SRSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

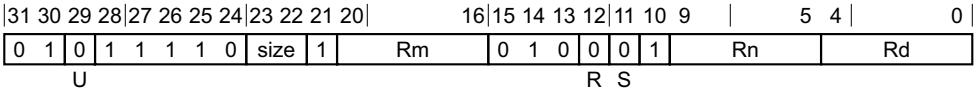
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.269 SSHL

Signed shift left (register)

### Scalar



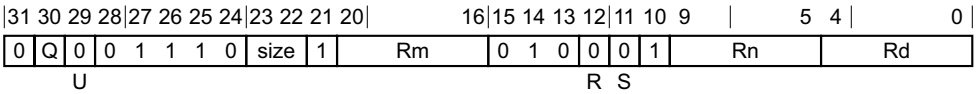

#### *Scalar variant*

```
SSHL <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

### Vector




#### *Vector variant*

```
SSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

`D` when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

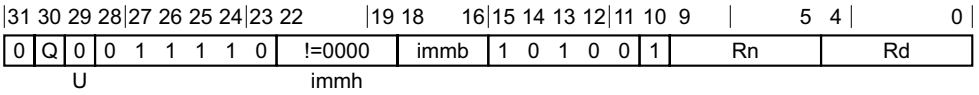
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.270 SSHLL, SSHLL2

Signed shift left long (immediate)

This instruction is used by the alias SXTL, SXTL2. See the *Alias conditions* table for details of when each alias is preferred.




### *Vector variant*

```
SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

### Alias conditions

| Alias | is preferred when |
|---|---|
| SXTL, SXTL2 | `immb == '000' && BitCount(immh) == 1` |

### Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

(UInt(immh:immb)-8) when immh = 0001

(UInt(immh:immb)-16) when immh = 001x

(UInt(immh:immb)-32) when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

The encoding immh = 1xxx is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

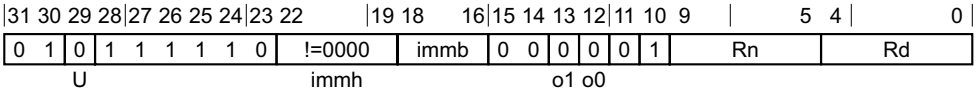
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.271 SSHR

Signed shift right (immediate)

### Scalar




#### *Scalar variant*

```
SSHR <V><d>, <V><n>, #<shift>
```

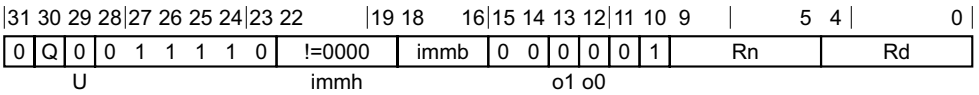
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector




#### *Vector variant*

```
SSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

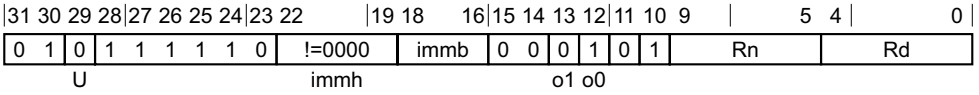
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.272 SSRA

Signed shift right and accumulate (immediate)

### Scalar




#### *Scalar variant*

```
SSRA <V><d>, <V><n>, #<shift>
```

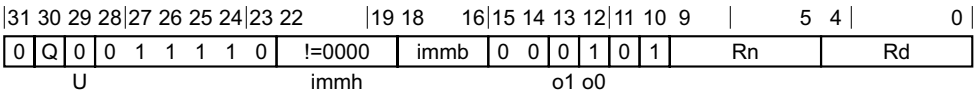
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector




#### *Vector variant*

```
SSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

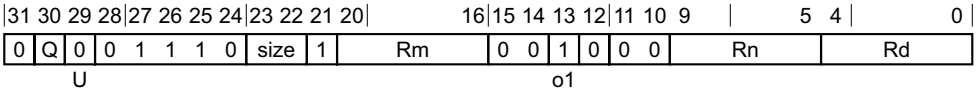
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.273 SSUBL, SSUBL2

Signed subtract long




### *Three registers, not all the same type variant*

```
SSUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

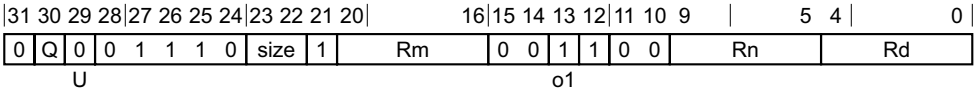
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.274 SSUBW, SSUBW2

Signed subtract wide




### *Three registers, not all the same type variant*

```
SSUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> — Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `size = 11, Q = x` is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

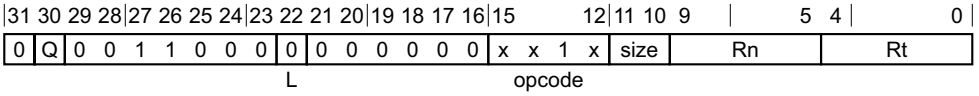
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.275 ST1 (multiple structures)

Store multiple 1-element structures from one, two three or four registers

### No offset




#### *One register variant*

Applies when `opcode = 0111`.

```
ST1 { <Vt>.<T> }, [<Xn|SP>]
```

#### *Two registers variant*

Applies when `opcode = 1010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

#### *Three registers variant*

Applies when `opcode = 0110`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

#### *Four registers variant*

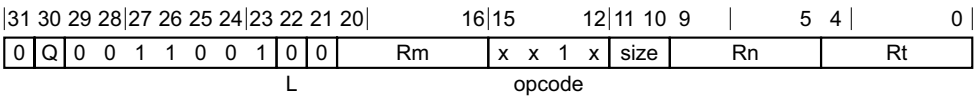
Applies when `opcode = 0010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *One register, immediate offset variant*

Applies when `Rm = 11111 && opcode = 0111`.

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

#### *One register, register offset variant*

Applies when `Rm != 11111 && opcode = 0111`.

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

#### *Two registers, immediate offset variant*

Applies when `Rm = 11111 && opcode = 1010`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

#### *Two registers, register offset variant*

Applies when `Rm != 11111 && opcode = 1010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

#### *Three registers, immediate offset variant*

Applies when `Rm = 11111 && opcode = 0110`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

#### *Three registers, register offset variant*

Applies when `Rm != 11111 && opcode = 0110`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

#### *Four registers, immediate offset variant*

Applies when `Rm = 11111 && opcode = 0010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

#### *Four registers, register offset variant*

Applies when `Rm != 11111 && opcode = 0010`.

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| `8B` | when `size = 00, Q = 0` |
| `16B` | when `size = 00, Q = 1` |
| `4H` | when `size = 01, Q = 0` |
| `8H` | when `size = 01, Q = 1` |
| `2S` | when `size = 10, Q = 0` |
| `4S` | when `size = 10, Q = 1` |
| `1D` | when `size = 11, Q = 0` |
| `2D` | when `size = 11, Q = 1` |

`<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

`<Vt3>` Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

`<Vt4>` Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#8 when Q = 0

#16 when Q = 1

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#16 when Q = 0

#32 when Q = 1

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#24 when Q = 0

#48 when Q = 1

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#32 when Q = 0

#64 when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

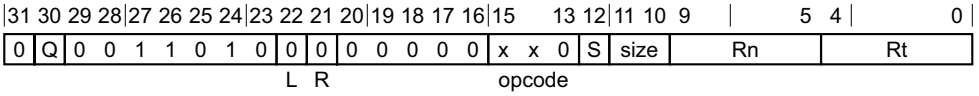
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.276 ST1 (single structure)

Store single 1-element structure from one lane of one register

### No offset




#### *8-bit variant*

Applies when `opcode = 000`.

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>]
```

#### *16-bit variant*

Applies when `opcode = 010 && size = x0`.

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>]
```

#### *32-bit variant*

Applies when `opcode = 100 && size = 00`.

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>]
```

#### *64-bit variant*

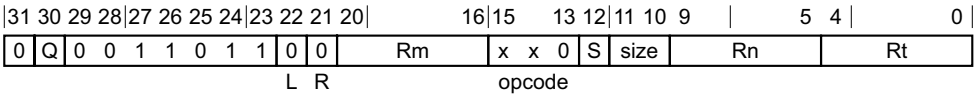
Applies when `opcode = 100 && S = 0 && size = 01`.

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>]
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *8-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 000`.

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

#### *8-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 000`.

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

#### *16-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 010 && size = x0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

#### *16-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 010 && size = x0`.

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

#### *32-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 100 && size = 00`.

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

#### *32-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 100 && size = 00`.

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

#### *64-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 100 && S = 0 && size = 01`.

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

#### *64-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 100 && S = 0 && size = 01`.

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

| | |
|---|---|
| `<Vt>` | Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<index>` | For the 8-bit variant: is the element index, encoded in "Q:S:size".<br>For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".<br>For the 32-bit variant: is the element index, encoded in "Q:S".<br>For the 64-bit variant: is the element index, encoded in "Q". |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field. |

### Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

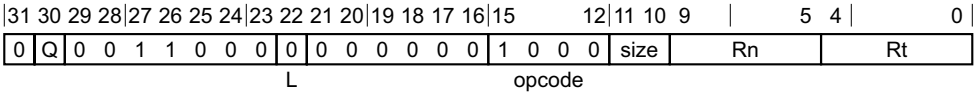
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.277 ST2 (multiple structures)

Store multiple 2-element structures from two registers

### No offset



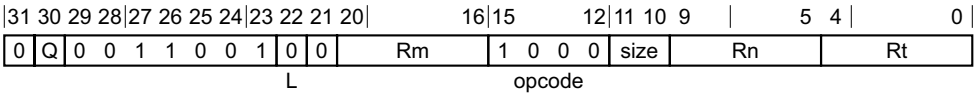

#### *No offset variant*

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

#### *Decode for this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *Immediate offset variant*

Applies when `Rm = 11111`.

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

#### *Register offset variant*

Applies when `Rm != 11111`.

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when `size = 00, Q = 0`
- 16B when `size = 00, Q = 1`
- 4H when `size = 01, Q = 0`
- 8H when `size = 01, Q = 1`
- 2S when `size = 10, Q = 0`
- 4S when `size = 10, Q = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#16 when Q = 0

#32 when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            tt = (tt + 1) MOD 32;

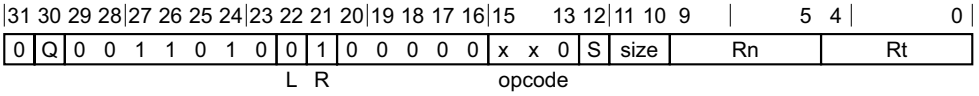
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.278 ST2 (single structure)

Store single 2-element structure from one lane of two registers

### No offset




#### *8-bit variant*

Applies when `opcode = 000`.

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

#### *16-bit variant*

Applies when `opcode = 010 && size = x0`.

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

#### *32-bit variant*

Applies when `opcode = 100 && size = 00`.

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

#### *64-bit variant*

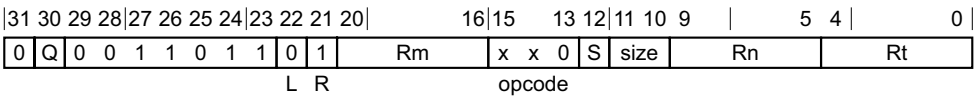
Applies when `opcode = 100 && S = 0 && size = 01`.

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *8-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 000`.

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

#### *8-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 000`.

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

#### *16-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 010 && size = x0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

#### *16-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 010 && size = x0`.

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

#### *32-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 100 && size = 00`.

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

#### *32-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 100 && size = 00`.

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

#### *64-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 100 && S = 0 && size = 01`.

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

#### *64-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 100 && S = 0 && size = 01`.

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

| | |
|---|---|
| `<Vt>` | Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Vt2>` | Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32. |
| `<index>` | For the 8-bit variant: is the element index, encoded in "Q:S:size".<br>For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".<br>For the 32-bit variant: is the element index, encoded in "Q:S".<br>For the 64-bit variant: is the element index, encoded in "Q". |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field. |

### Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

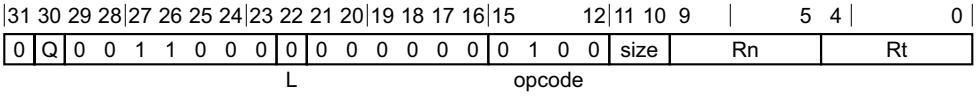
```
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.279 ST3 (multiple structures)

Store multiple 3-element structures from three registers

### No offset



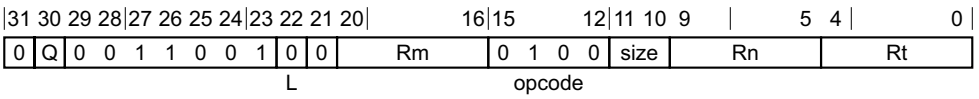

#### *No offset variant*

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

#### *Decode for this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *Immediate offset variant*

Applies when `Rm = 11111`.

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

#### *Register offset variant*

Applies when `Rm != 11111`.

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when `size = 00, Q = 0` |
| 16B | when `size = 00, Q = 1` |
| 4H | when `size = 01, Q = 0` |
| 8H | when `size = 01, Q = 1` |
| 2S | when `size = 10, Q = 0` |
| 4S | when `size = 10, Q = 1` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#24 when Q = 0

#48 when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

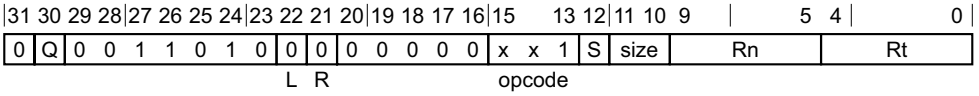
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.280 ST3 (single structure)

Store single 3-element structure from one lane of three registers

### No offset




#### *8-bit variant*

Applies when `opcode = 001`.

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

#### *16-bit variant*

Applies when `opcode = 011 && size = x0`.

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

#### *32-bit variant*

Applies when `opcode = 101 && size = 00`.

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

#### *64-bit variant*

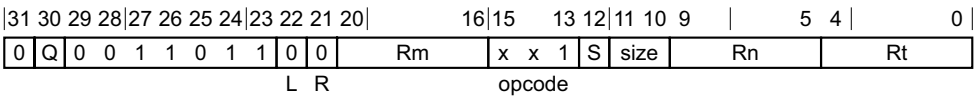
Applies when `opcode = 101 && S = 0 && size = 01`.

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *8-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 001`.

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

#### *8-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 001`.

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

#### *16-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 011 && size = x0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

#### *16-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 011 && size = x0`.

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

#### *32-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 101 && size = 00`.

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

#### *32-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 101 && size = 00`.

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

#### *64-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 101 && S = 0 && size = 01`.

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

#### *64-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 101 && S = 0 && size = 01`.

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

| | |
|---|---|
| `<Vt>` | Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Vt2>` | Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32. |
| `<Vt3>` | Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32. |
| `<index>` | For the 8-bit variant: is the element index, encoded in "Q:S:size".<br>For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".<br>For the 32-bit variant: is the element index, encoded in "Q:S".<br>For the 64-bit variant: is the element index, encoded in "Q". |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field. |

### Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

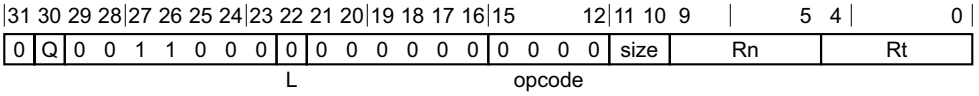
```
    SP[] = address + offs;
else
    X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.281 ST4 (multiple structures)

Store multiple 4-element structures from four registers

### No offset



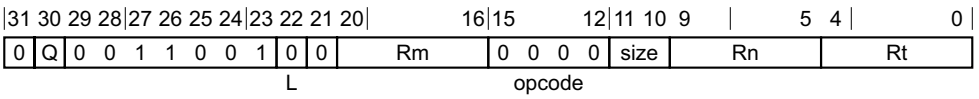

#### *No offset variant*

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

#### *Decode for this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *Immediate offset variant*

Applies when `Rm = 11111`.

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

#### *Register offset variant*

Applies when `Rm != 11111`.

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

`<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| `8B` | when `size = 00, Q = 0` |
| `16B` | when `size = 00, Q = 1` |
| `4H` | when `size = 01, Q = 0` |
| `8H` | when `size = 01, Q = 1` |
| `2S` | when `size = 10, Q = 0` |
| `4S` | when `size = 10, Q = 1` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#32 when Q = 0

#64 when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared decode for all encodings

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                    V[tt] = rval;
                else // memop == MemOp_STORE
                    Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

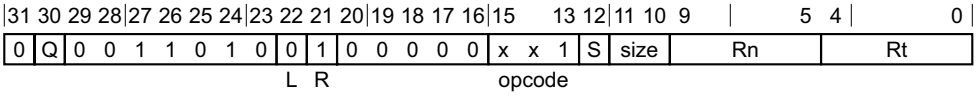
    if wback then
        if m != 31 then
            offs = X[m];
        if n == 31 then
            SP[] = address + offs;
        else
            X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.282 ST4 (single structure)

Store single 4-element structure from one lane of four registers

### No offset




#### *8-bit variant*

Applies when `opcode = 001`.

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

#### *16-bit variant*

Applies when `opcode = 011 && size = x0`.

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

#### *32-bit variant*

Applies when `opcode = 101 && size = 00`.

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

#### *64-bit variant*

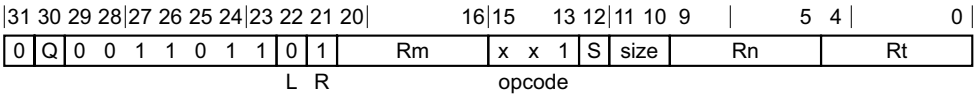
Applies when `opcode = 101 && S = 0 && size = 01`.

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index




#### *8-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 001`.

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

#### *8-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 001`.

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

#### *16-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 011 && size = x0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

#### *16-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 011 && size = x0`.

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

#### *32-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 101 && size = 00`.

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

#### *32-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 101 && size = 00`.

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

#### *64-bit, immediate offset variant*

Applies when `Rm = 11111 && opcode = 101 && S = 0 && size = 01`.

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

#### *64-bit, register offset variant*

Applies when `Rm != 11111 && opcode = 101 && S = 0 && size = 01`.

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

#### *Decode for all variants of this encoding*

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler symbols

| | |
|---|---|
| `<Vt>` | Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Vt2>` | Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32. |
| `<Vt3>` | Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32. |
| `<Vt4>` | Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32. |
| `<index>` | For the 8-bit variant: is the element index, encoded in "Q:S:size".<br>For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".<br>For the 32-bit variant: is the element index, encoded in "Q:S".<br>For the 64-bit variant: is the element index, encoded in "Q". |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<Xm>` | Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field. |

### Shared decode for all encodings

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
```

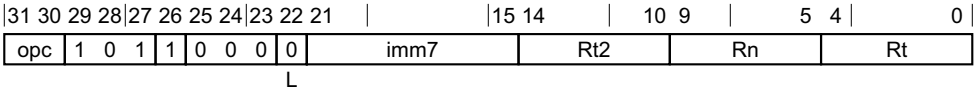

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    offs = X[m];
if n == 31 then
    SP[] = address + offs;
else
    X[n] = address + offs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.283 STNP (SIMD&FP)

Store pair of SIMD&FP registers, with non-temporal hint




### *32-bit variant*

Applies when opc = 00.

```
STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### *64-bit variant*

Applies when opc = 01.

```
STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### *128-bit variant*

Applies when opc = 10.

```
STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

### *Decode for all variants of this encoding*

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

## Assembler symbols

| | |
|---|---|
| <Dt1> | Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Dt2> | Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| <Qt1> | Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Qt2> | Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| <St1> | Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <St2> | Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <imm> | For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.<br>For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.<br>For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VECSTREAM;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t]  = data1;
        V[t2] = data2;

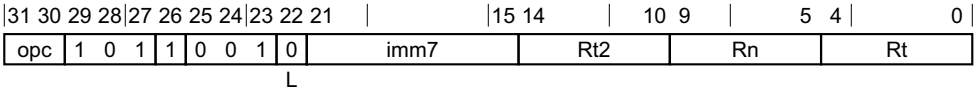
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.284 STP (SIMD&FP)

Store pair of SIMD&FP registers

### Post-index




#### *32-bit variant*

Applies when opc = 00.

```
STP <St1>, <St2>, [<Xn|SP>], #<imm>
```

#### *64-bit variant*

Applies when opc = 01.

```
STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

#### *128-bit variant*

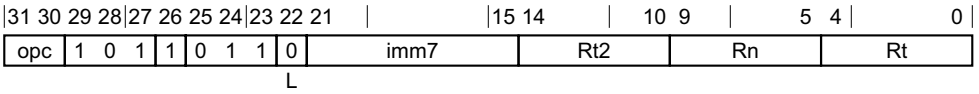
Applies when opc = 10.

```
STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

#### *Decode for all variants of this encoding*

```
boolean wback  = TRUE;
boolean postindex = TRUE;
```

### Pre-index




#### *32-bit variant*

Applies when opc = 00.

```
STP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

#### *64-bit variant*

Applies when opc = 01.

```
STP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

#### *128-bit variant*

Applies when opc = 10.

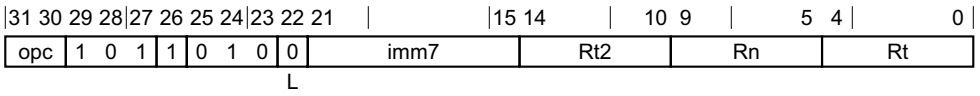
```
STP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

#### *Decode for all variants of this encoding*

```
boolean wback  = TRUE;
boolean postindex = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Signed offset




#### *32-bit variant*

Applies when `opc = 00`.

```
STP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

#### *64-bit variant*

Applies when `opc = 01`.

```
STP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

#### *128-bit variant*

Applies when `opc = 10`.

```
STP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

#### *Decode for all variants of this encoding*

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

### Assembler symbols

| | |
|---|---|
| `<Dt1>` | Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Dt2>` | Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| `<Qt1>` | Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<Qt2>` | Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| `<St1>` | Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field. |
| `<St2>` | Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field. |
| `<Xn\|SP>` | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| `<imm>` | For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.<br>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.<br>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.<br>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.<br>For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.<br>For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

### Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t]  = data1;
        V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.285 STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset)

### Post-index




#### *8-bit variant*

Applies when `size = 00 && opc = 00`.

```
STR <Bt>, [<Xn|SP>], #<simm>
```

#### *16-bit variant*

Applies when `size = 01 && opc = 00`.

```
STR <Ht>, [<Xn|SP>], #<simm>
```

#### *32-bit variant*

Applies when `size = 10 && opc = 00`.

```
STR <St>, [<Xn|SP>], #<simm>
```

#### *64-bit variant*

Applies when `size = 11 && opc = 00`.

```
STR <Dt>, [<Xn|SP>], #<simm>
```

#### *128-bit variant*

Applies when `size = 00 && opc = 10`.

```
STR <Qt>, [<Xn|SP>], #<simm>
```

#### *Decode for all variants of this encoding*

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index




#### *8-bit variant*

Applies when `size = 00 && opc = 00`.

```
STR <Bt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### 16-bit variant

Applies when size = 01 && opc = 00.

```
STR <Ht>, [<Xn|SP>, #<simm>]!
```

#### 32-bit variant

Applies when size = 10 && opc = 00.

```
STR <St>, [<Xn|SP>, #<simm>]!
```

#### 64-bit variant

Applies when size = 11 && opc = 00.

```
STR <Dt>, [<Xn|SP>, #<simm>]!
```

#### 128-bit variant

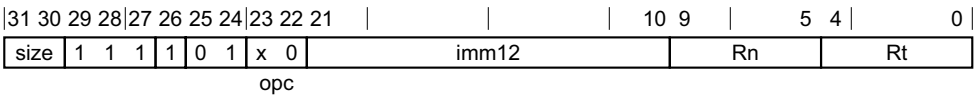
Applies when size = 00 && opc = 10.

```
STR <Qt>, [<Xn|SP>, #<simm>]!
```

#### Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset




#### 8-bit variant

Applies when size = 00 && opc = 00.

```
STR <Bt>, [<Xn|SP>{, #<pimm>}]
```

#### 16-bit variant

Applies when size = 01 && opc = 00.

```
STR <Ht>, [<Xn|SP>{, #<pimm>}]
```

#### 32-bit variant

Applies when size = 10 && opc = 00.

```
STR <St>, [<Xn|SP>{, #<pimm>}]
```

#### 64-bit variant

Applies when size = 11 && opc = 00.

```
STR <Dt>, [<Xn|SP>{, #<pimm>}]
```

#### 128-bit variant

Applies when size = 00 && opc = 10.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
STR <Qt>, [<Xn|SP>{, #<pimm>}]
```

### Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

## Assembler symbols

| | |
|---|---|
| <Bt> | Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Dt> | Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Ht> | Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Qt> | Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <St> | Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field. |
| <pimm> | For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.<br>For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.<br>For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.<br>For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16. |

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            data = V[t];
            Mem[address, datasize DIV 8, acctype] = data;

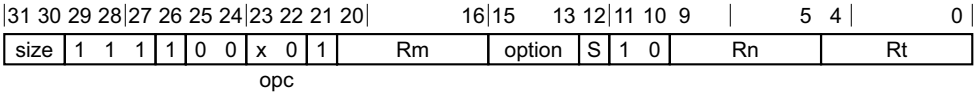
        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            V[t] = data;

    if wback then
        if postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.286 STR (register, SIMD&FP)

Store SIMD&FP register (register offset)




### 8-bit variant

Applies when size = 00 && opc = 00 && option != 011.

```
STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### 8-bit variant

Applies when size = 00 && opc = 00 && option = 011.

```
STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### 16-bit variant

Applies when size = 01 && opc = 00.

```
STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 32-bit variant

Applies when size = 10 && opc = 00.

```
STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-bit variant

Applies when size = 11 && opc = 00.

```
STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 128-bit variant

Applies when size = 00 && opc = 10.

```
STR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler symbols

| | |
|---|---|
| <Bt> | Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Dt> | Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Ht> | Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Qt>` Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<St>` Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` For the 8-bit variant: is the index extend specifier, encoded in the "option" field. It can have the following values:

`UXTW` when `option = 010`

`SXTW` when `option = 110`

`SXTX` when `option = 111`

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

`UXTW` when `option = 010`

`LSL` when `option = 011`

`SXTW` when `option = 110`

`SXTX` when `option = 111`

`<amount>` For the 8-bit variant: is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#1` when `S = 1`

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#2` when `S = 1`

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#3` when `S = 1`

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#4` when `S = 1`

## Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.287 STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset)




### 8-bit variant

Applies when size = 00 && opc = 00.

```
STUR <Bt>, [<Xn|SP>{, #<simm>}]
```

### 16-bit variant

Applies when size = 01 && opc = 00.

```
STUR <Ht>, [<Xn|SP>{, #<simm>}]
```

### 32-bit variant

Applies when size = 10 && opc = 00.

```
STUR <St>, [<Xn|SP>{, #<simm>}]
```

### 64-bit variant

Applies when size = 11 && opc = 00.

```
STUR <Dt>, [<Xn|SP>{, #<simm>}]
```

### 128-bit variant

Applies when size = 00 && opc = 10.

```
STUR <Qt>, [<Xn|SP>{, #<simm>}]
```

### Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler symbols

| | |
|---|---|
| <Bt> | Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Dt> | Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Ht> | Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Qt> | Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <St> | Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field. |
| <Xn\|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <simm> | Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

### Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

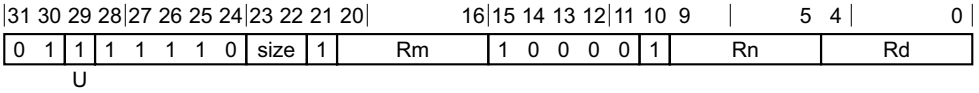
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.288 SUB (vector)

Subtract (vector)

### Scalar



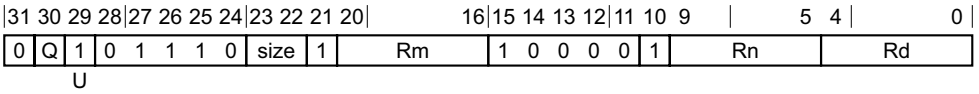

#### *Scalar variant*

```
SUB <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

### Vector




#### *Vector variant*

```
SUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

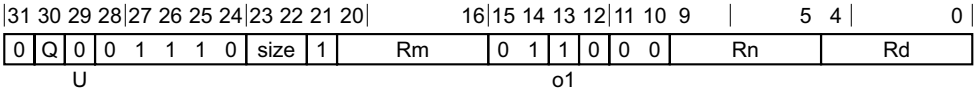
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.289 SUBHN, SUBHN2

Subtract returning high narrow




### *Three registers, not all the same type variant*

```
SUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize)   result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

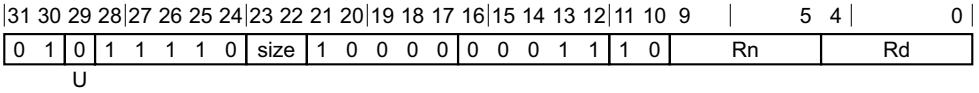
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.290 SUQADD

Signed saturating accumulate of unsigned value

### Scalar




#### *Scalar variant*

```
SUQADD <V><d>, <V><n>
```

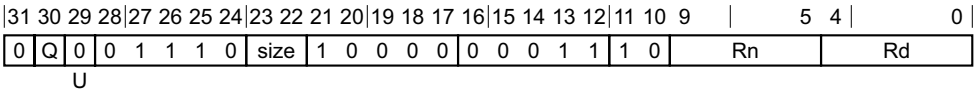
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
SUQADD <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| B | when size = 00 |
| H | when size = 01 |
| S | when size = 10 |
| D | when size = 11 |

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

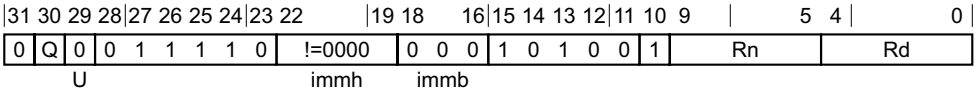

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.291 SXTL, SXTL2

Signed extend long

This instruction is an alias of the SSHLL, SSHLL2 instruction. This means that:

- The encodings in this description are named to match the encodings of SSHLL, SSHLL2.
- The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.




### *Vector variant*

`SXTL{2} <Vd>.<Ta>, <Vn>.<Tb>`

is equivalent to

`SSHLL{2}  <Vd>.<Ta>, <Vn>.<Tb>, #0`

and is the preferred disassembly when `BitCount(immh) == 1`.

### Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001, Q = 0`

`16B` when `immh = 0001, Q = 1`

`4H` when `immh = 001x, Q = 0`

`8H` when `immh = 001x, Q = 1`

`2S` when `immh = 01xx, Q = 0`

`4S` when `immh = 01xx, Q = 1`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000, Q = x`.

The encoding `immh = 1xxx, Q = x` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

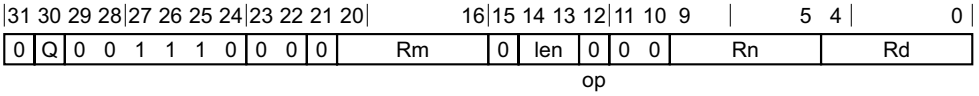
### Operation

The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.292 TBL

Table vector lookup




### *Two register table variant*

Applies when `len = 01`.

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

### *Three register table variant*

Applies when `len = 10`.

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

### *Four register table variant*

Applies when `len = 11`.

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

### *Single register table variant*

Applies when `len = 00`.

```
TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;
integer i;

// Create table from registers
for i = 0 to regs - 1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

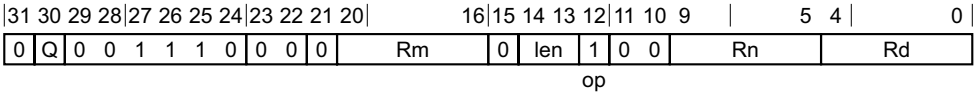
result = if is_tbl then Zeros() else V[d];
for i = 0 to elements - 1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.293 TBX

Table vector lookup extension




### *Two register table variant*

Applies when `len = 01`.

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

### *Three register table variant*

Applies when `len = 10`.

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

### *Four register table variant*

Applies when `len = 11`.

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

### *Single register table variant*

Applies when `len = 00`.

```
TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `8B` when `Q = 0`
- `16B` when `Q = 1`

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;
integer i;

// Create table from registers
for i = 0 to regs - 1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

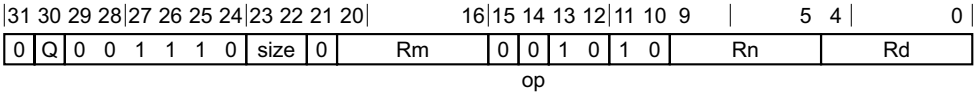
result = if is_tbl then Zeros() else V[d];
for i = 0 to elements - 1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.294 TRN1

Transpose vectors (primary)




### *Advanced SIMD variant*

```
TRN1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when size = 00, Q = 0 |
| 16B | when size = 00, Q = 1 |
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |
| 2D | when size = 11, Q = 1 |

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer p;

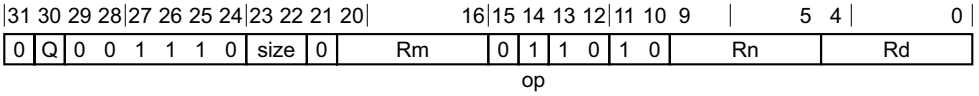
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.295 TRN2

Transpose vectors (secondary)




### *Advanced SIMD variant*

```
TRN2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when size = 00, Q = 0 |
| 16B | when size = 00, Q = 1 |
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |
| 2D | when size = 11, Q = 1 |

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer p;

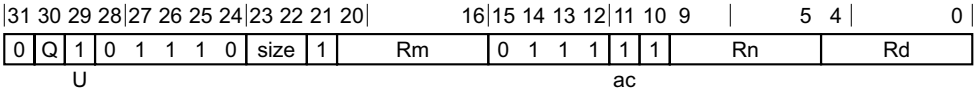
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.296 UABA

Unsigned absolute difference and accumulate




### *Three registers of the same type variant*

```
UABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
```

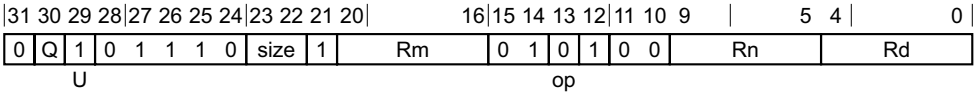

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        absdiff = Abs(element1 - element2)<esize-1:0>;
        Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.297 UABAL, UABAL2

Unsigned absolute difference and accumulate long




### *Three registers, not all the same type variant*

```
UABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

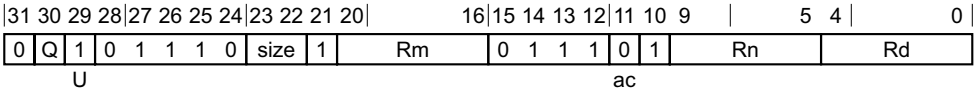
```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.298 UABD

Unsigned absolute difference (vector)




### *Three registers of the same type variant*

```
UABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
```

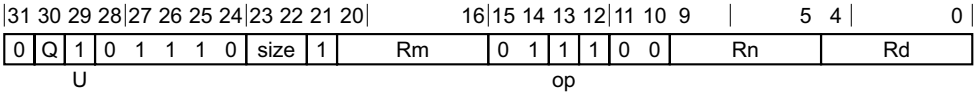

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        absdiff = Abs(element1 - element2)<esize-1:0>;
        Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
    V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.299 UABDL, UABDL2

Unsigned absolute difference long




### *Three registers, not all the same type variant*

```
UABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

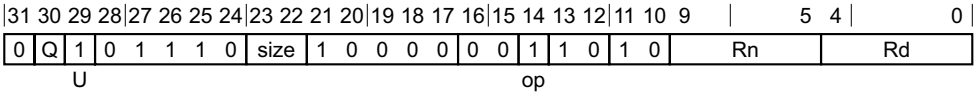
```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.300 UADALP

Unsigned add and accumulate long pairwise




### *Vector variant*

```
UADALP <Vd>.<Ta>, <Vn>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 4H when size = 00, Q = 0
- 8H when size = 00, Q = 1
- 2S when size = 01, Q = 0
- 4S when size = 01, Q = 1
- 1D when size = 10, Q = 0
- 2D when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(2*esize) sum;
integer op1;
integer op2;

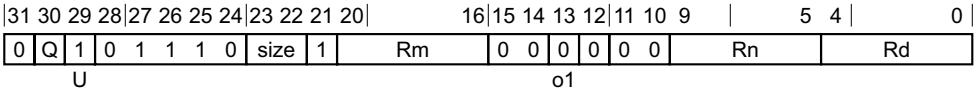
result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.301 UADDL, UADDL2

Unsigned add long (vector)




### *Three registers, not all the same type variant*

```
UADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

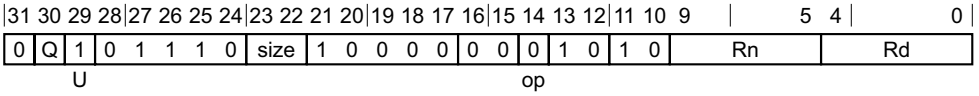
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.302 UADDLP

Unsigned add long pairwise




### *Vector variant*

```
UADDLP <Vd>.<Ta>, <Vn>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 4H when size = 00, Q = 0
- 8H when size = 00, Q = 1
- 2S when size = 01, Q = 0
- 4S when size = 01, Q = 1
- 1D when size = 10, Q = 0
- 2D when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(2*esize) sum;
integer op1;
integer op2;

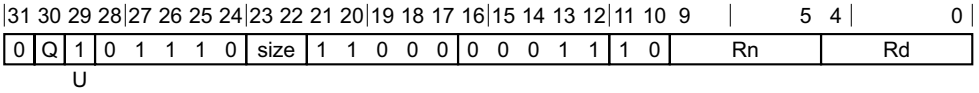
result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.303 UADDLV

Unsigned sum long across vector




### *Advanced SIMD variant*

```
UADDLV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

## Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- H when size = 00
- S when size = 01
- D when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

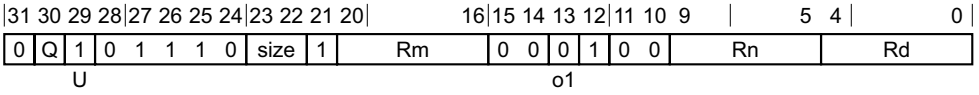
```
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d] = sum<2*esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.304 UADDW, UADDW2

Unsigned add wide




### *Three registers, not all the same type variant*

```
UADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "size" field. It can have the following values:

`8H` when `size = 00`

`4S` when `size = 01`

`2D` when `size = 10`

The encoding `size = 11` is reserved.

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

`<Tb>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

`8B` when `size = 00, Q = 0`

`16B` when `size = 00, Q = 1`

`4H` when `size = 01, Q = 0`

`8H` when `size = 01, Q = 1`

`2S` when `size = 10, Q = 0`

`4S` when `size = 10, Q = 1`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `size = 11, Q = x` is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

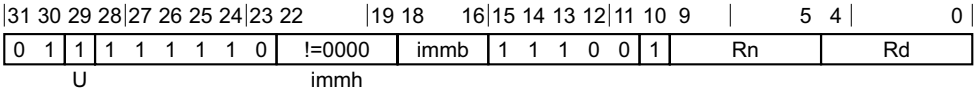
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.305 UCVTF (vector, fixed-point)

Unsigned fixed-point convert to floating-point (vector)

### Scalar




#### *Scalar variant*

```
UCVTF <V><d>, <V><n>, #<fbits>
```

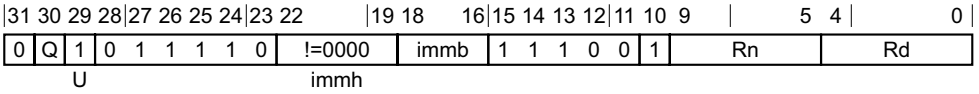
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '00xx' then ReservedValue();
integer esize = 32 << UInt(immh<3>);
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

### Vector




#### *Vector variant*

```
UCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh == '00xx' then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = 32 << UInt(immh<3>);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

S when `immh = 01xx`

D when `immh = 1xxx`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `immh = 00xx` is reserved.

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

`2D` when `immh = 1xxx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`, `Q = x`.

The following encodings are reserved:

- `immh = 0001`, `Q = x`.
- `immh = 001x`, `Q = x`.
- `immh = 1xxx`, `Q = 0`.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<fbits>` For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immh:immb" field. It can have the following values:

`(64-UInt(immh:immb))` when `immh = 01xx`

`(128-UInt(immh:immb))` when `immh = 1xxx`

The encoding `immh = 00xx` is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immh:immb" field. It can have the following values:

`(64-UInt(immh:immb))` when `immh = 01xx`

`(128-UInt(immh:immb))` when `immh = 1xxx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The following encodings are reserved:

- `immh = 0001`.
- `immh = 001x`.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding);

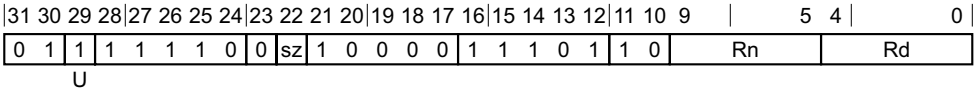
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.306 UCVTF (vector, integer)

Unsigned integer convert to floating-point (vector)

### Scalar




#### *Scalar variant*

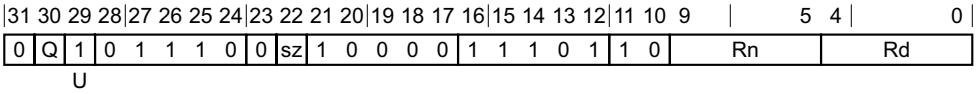
```
UCVTF <V><d>, <V><n>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
UCVTF <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

4S when sz = 0, Q = 1

2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
FPRounding rounding = FPRoundingMode(FPCR);
bits(esize) element;

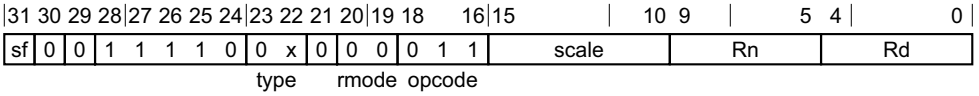
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.307 UCVTF (scalar, fixed-point)

Unsigned fixed-point convert to floating-point (scalar) : `Vd = unsigned_convertFromInt(Rn/(2^fbits))`




### 32-bit to single-precision variant

Applies when `sf = 0 && type = 00`.

```
UCVTF <Sd>, <Wn>, #<fbits>
```

### 32-bit to double-precision variant

Applies when `sf = 0 && type = 01`.

```
UCVTF <Dd>, <Wn>, #<fbits>
```

### 64-bit to single-precision variant

Applies when `sf = 1 && type = 00`.

```
UCVTF <Sd>, <Xn>, #<fbits>
```

### 64-bit to double-precision variant

Applies when `sf = 1 && type = 01`.

```
UCVTF <Dd>, <Xn>, #<fbits>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '1x' UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
    when '00 11'        // FCVTZ
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    otherwise
        UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<fbits> For the 32-bit to double-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".

For the 64-bit to double-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

### Operation

```
CheckFPAdvSIMDEnabled64();

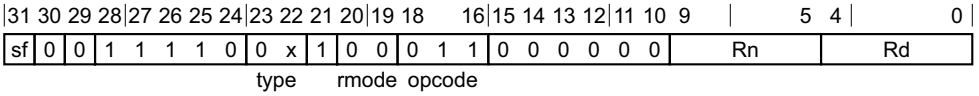
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
        V[d] = fltval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.308 UCVTF (scalar, integer)

Unsigned integer convert to floating-point (scalar) : `Vd = unsigned_convertFromInt(Rn)`




### 32-bit to single-precision variant

Applies when `sf = 0 && type = 00`.

```
UCVTF <Sd>, <Wn>
```

### 32-bit to double-precision variant

Applies when `sf = 0 && type = 01`.

```
UCVTF <Dd>, <Wn>
```

### 64-bit to single-precision variant

Applies when `sf = 1 && type = 00`.

```
UCVTF <Sd>, <Xn>
```

### 64-bit to double-precision variant

Applies when `sf = 1 && type = 01`.

```
UCVTF <Dd>, <Xn>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
        if fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
    otherwise
        UnallocatedEncoding();
```

## Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();

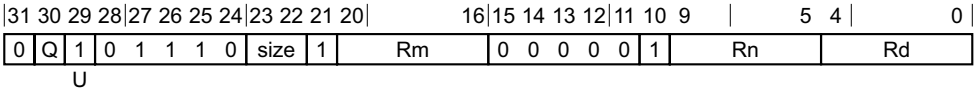
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = Vpart[n,part];
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        Vpart[d,part] = intval;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.309 UHADD

Unsigned halving add




### *Three registers of the same type variant*

```
UHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

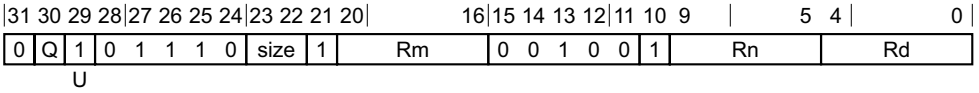
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.310 UHSUB

Unsigned halving subtract




### *Three registers of the same type variant*

```
UHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

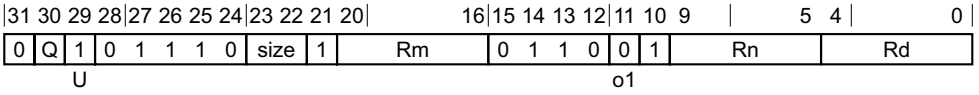
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.311 UMAX

Unsigned maximum (vector)




### *Three registers of the same type variant*

```
UMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

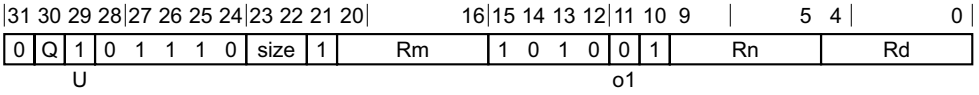
```
            Elem[result, e, esize] = maxmin<esize-1:0>;

        V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.312 UMAXP

Unsigned maximum pairwise




### *Three registers of the same type variant*

```
UMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

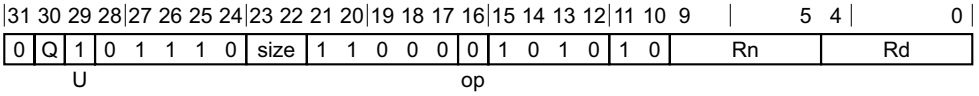
```
            Elem[result, e, esize] = maxmin<esize-1:0>;

        V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.313 UMAXV

Unsigned maximum across vector




### *Advanced SIMD variant*

```
UMAXV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

## Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

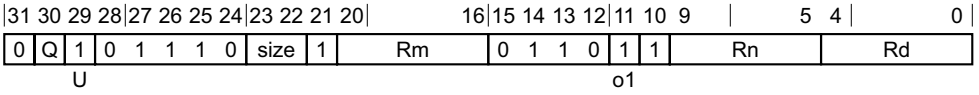
```
maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.314 UMIN

Unsigned minimum (vector)




### *Three registers of the same type variant*

```
UMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

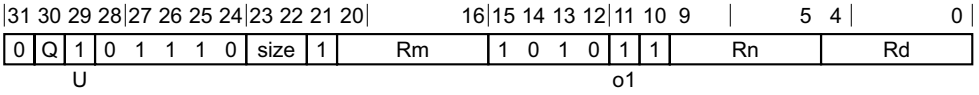
```
            Elem[result, e, esize] = maxmin<esize-1:0>;

        V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.315 UMINP

Unsigned minimum pairwise




### *Three registers of the same type variant*

```
UMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

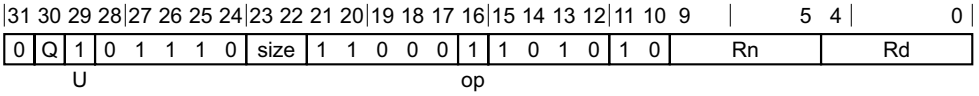
```
            Elem[result, e, esize] = maxmin<esize-1:0>;

        V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.316 UMINV

Unsigned minimum across vector




### *Advanced SIMD variant*

```
UMINV <V><d>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

## Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

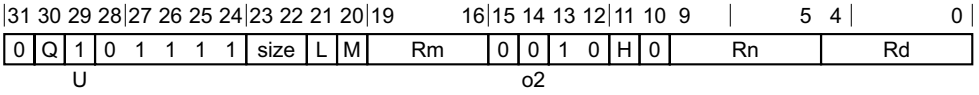
```
maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.317 UMLAL, UMLAL2 (by element)

Unsigned multiply-add long (vector, by element)




### *Vector variant*

```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

4S when size = 01

2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(idxdsize)   operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

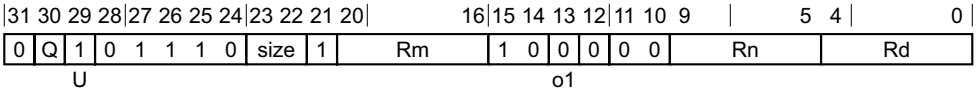
element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.318 UMLAL, UMLAL2 (vector)

Unsigned multiply-add long (vector)




### *Three registers, not all the same type variant*

```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

## Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

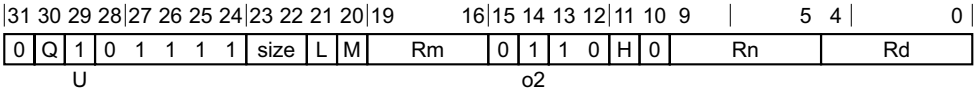
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.319 UMLSL, UMLSL2 (by element)

Unsigned multiply-subtract long (vector, by element)




### *Vector variant*

```
UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

4S when size = 01

2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(idxdsize)   operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

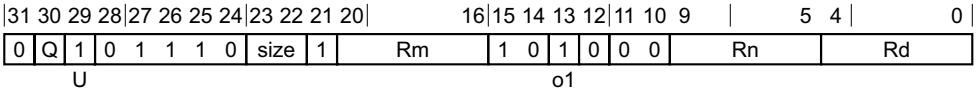
element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.320 UMLSL, UMLSL2 (vector)

Unsigned multiply-subtract long (vector)




### *Three registers, not all the same type variant*

```
UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

## Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

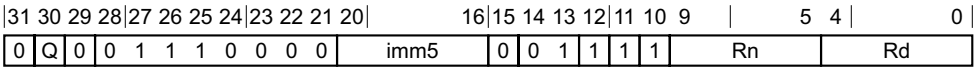
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.321 UMOV

Unsigned move vector element to general-purpose register

This instruction is used by the alias MOV (to general). See the *Alias conditions* table for details of when each alias is preferred.




### *32-bit variant*

Applies when `Q = 0`.

```
UMOV <Wd>, <Vn>.<Ts>[<index>]
```

### *64-bit variant*

Applies when `Q = 1 && imm5 = x1000`.

```
UMOV <Xd>, <Vn>.<Ts>[<index>]
```

### *Decode for all variants of this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when '0xxxx1' size = 0;     // UMOV Wd, Vn.B
    when '0xxx10' size = 1;     // UMOV Wd, Vn.H
    when '0xx100' size = 2;     // UMOV Wd, Vn.S
    when '1x1000' size = 3;     // UMOV Xd, Vn.D
    otherwise     UnallocatedEncoding();

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| MOV (to general) | `imm5 == 'x1000'` |
| MOV (to general) | `imm5 == 'xx100'` |

## Assembler symbols

| | |
|---|---|
| `<Wd>` | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Xd>` | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| `<Vn>` | Is the name of the SIMD&FP source register, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Ts> For the 32-bit variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

B when imm5 = xxxx1

H when imm5 = xxx10

S when imm5 = xx100

The encoding imm5 = xx000 is reserved.

For the 64-bit variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

D when imm5 = x1000

The following encodings are reserved:

- imm5 = x0000.
- imm5 = xxxx1.
- imm5 = xxx10.
- imm5 = xx100.

<index> For the 32-bit variant: is the element index encoded in the "imm5" field. It can have the following values:

imm5<4:1> when imm5 = xxxx1

imm5<4:2> when imm5 = xxx10

imm5<4:3> when imm5 = xx100

The encoding imm5 = xx000 is reserved.

For the 64-bit variant: is the element index encoded in "imm5<4>".

## Operation

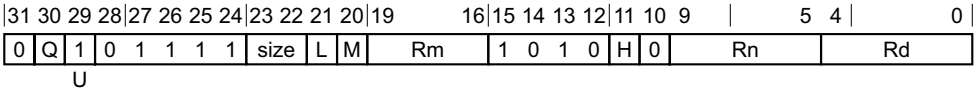
```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];

X[d] = ZeroExtend(Elem[operand, index, esize], datasize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.322 UMULL, UMULL2 (by element)

Unsigned multiply long (vector, by element)




### *Vector variant*

```
UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

### *Decode for this encoding*

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

4S when size = 01

2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01

M:Rm when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01

S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01

H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(idxdsize)   operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

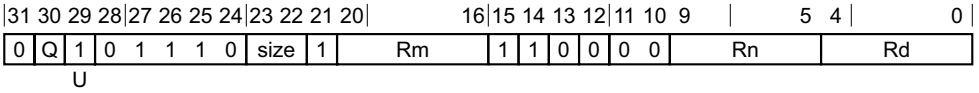
element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.323 UMULL, UMULL2 (vector)

Unsigned multiply long (vector)




### *Three registers, not all the same type variant*

```
UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> — Is the name of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1 * element2)<2*esize-1:0>;

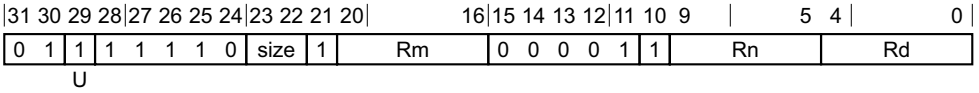
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.324 UQADD

Unsigned saturating add

### Scalar



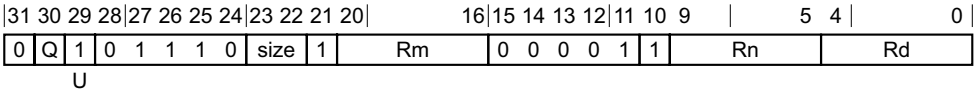

#### *Scalar variant*

```
UQADD <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
UQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

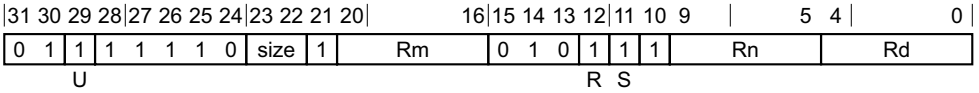
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.325 UQRSHL

Unsigned saturating rounding shift left (register)

### Scalar



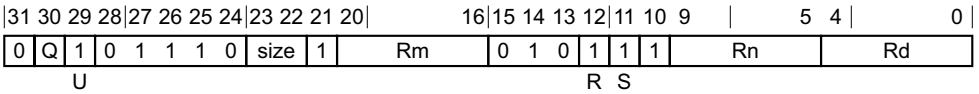

#### *Scalar variant*

```
UQRSHL <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

### Vector




#### *Vector variant*

```
UQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

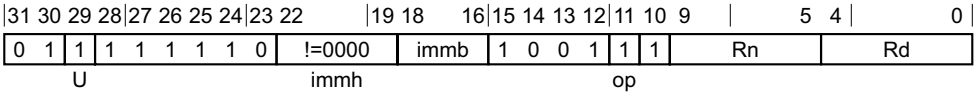
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.326 UQRSHRN, UQRSHRN2

Unsigned saturating rounded shift right narrow (immediate)

### Scalar




#### *Scalar variant*

```
UQRSHRN <Vb><d>, <Va><n>, #<shift>
```

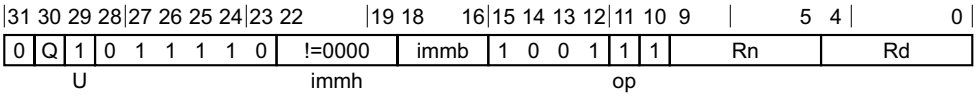
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
UQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001`, `Q = 0`

`16B` when `immh = 0001`, `Q = 1`

`4H` when `immh = 001x`, `Q = 0`

`8H` when `immh = 001x`, `Q = 1`

`2S` when `immh = 01xx`, `Q = 0`

`4S` when `immh = 01xx`, `Q = 1`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`, `Q = x`.

The encoding `immh = 1xxx`, `Q = x` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

<Vb> Is the destination width specifier, encoded in the "immh" field. It can have the following values:

`B` when `immh = 0001`

`H` when `immh = 001x`

`S` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in the "immh" field. It can have the following values:

`H` when `immh = 0001`

`S` when `immh = 001x`

`D` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

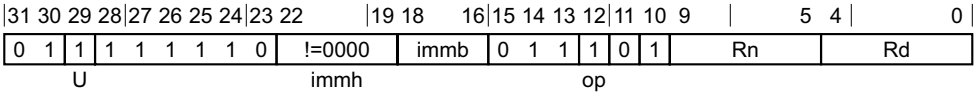
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.327 UQSHL (immediate)

Unsigned saturating shift left (immediate)

### Scalar




#### Scalar variant

```
UQSHL <V><d>, <V><n>, #<shift>
```

#### Decode for this encoding

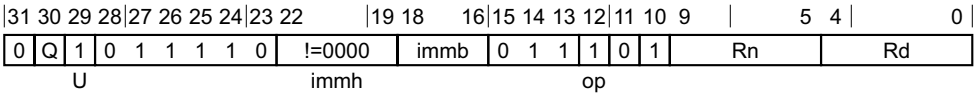
```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE;  dst_unsigned = TRUE;
```

### Vector




#### Vector variant

```
UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
when '11' src_unsigned = TRUE;  dst_unsigned = TRUE;
```

## Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

- B when immh = 0001
- H when immh = 001x
- S when immh = 01xx
- D when immh = 1xxx

The encoding immh = 0000 is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- 8B when immh = 0001, Q = 0
- 16B when immh = 0001, Q = 1
- 4H when immh = 001x, Q = 0
- 8H when immh = 001x, Q = 1
- 2S when immh = 01xx, Q = 0
- 4S when immh = 01xx, Q = 1
- 2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

- (UInt(immh:immb)-8) when immh = 0001
- (UInt(immh:immb)-16) when immh = 001x
- (UInt(immh:immb)-32) when immh = 01xx
- (UInt(immh:immb)-64) when immh = 1xxx

The encoding immh = 0000 is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

- (UInt(immh:immb)-8) when immh = 0001
- (UInt(immh:immb)-16) when immh = 001x
- (UInt(immh:immb)-32) when immh = 01xx
- (UInt(immh:immb)-64) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

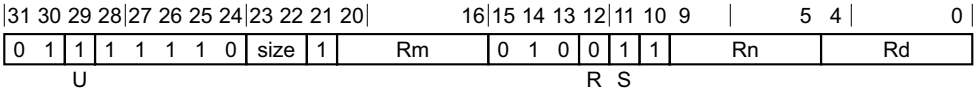
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.328 UQSHL (register)

Unsigned saturating shift left (register)

### Scalar



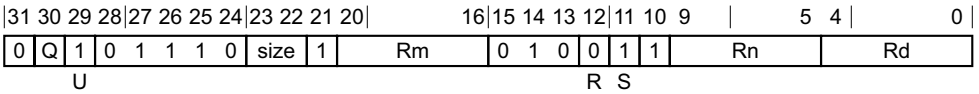

#### *Scalar variant*

```
UQSHL <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

### Vector




#### *Vector variant*

```
UQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| B | when size = 00 |
| H | when size = 01 |
| S | when size = 10 |
| D | when size = 11 |

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

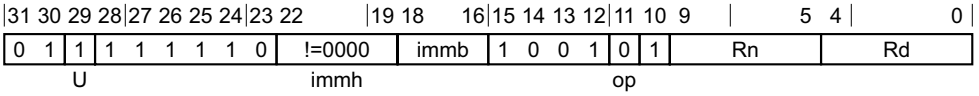
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.329 UQSHRN, UQSHRN2

Unsigned saturating shift right narrow (immediate)

### Scalar




#### *Scalar variant*

```
UQSHRN <Vb><d>, <Va><n>, #<shift>
```

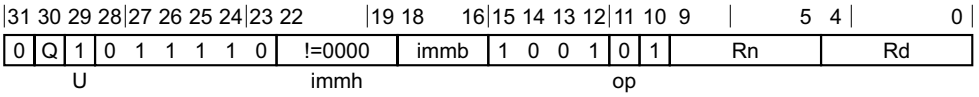
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
UQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0

[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when immh = 0001

4S when immh = 001x

2D when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vb> Is the destination width specifier, encoded in the "immh" field. It can have the following values:

B when immh = 0001

H when immh = 001x

S when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 0001

S when immh = 001x

D when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

The following encodings are reserved:

- `immh = 0000`.
- `immh = 1xxx`.

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

`(16-UInt(immh:immb))` when `immh = 0001`

`(32-UInt(immh:immb))` when `immh = 001x`

`(64-UInt(immh:immb))` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

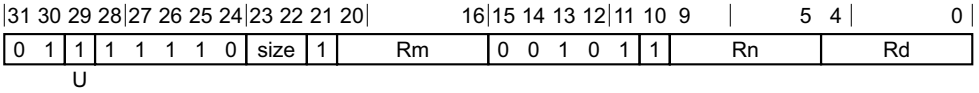
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.330 UQSUB

Unsigned saturating subtract

### Scalar



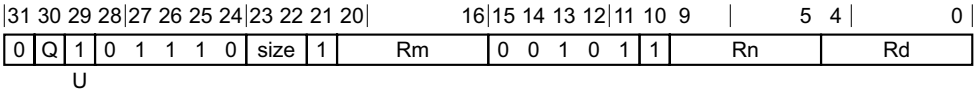

#### *Scalar variant*

```
UQSUB <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
UQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

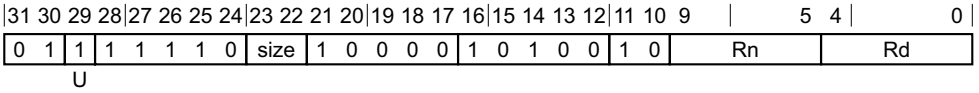
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.331 UQXTN, UQXTN2

Unsigned saturating extract narrow

### Scalar




#### *Scalar variant*

```
UQXTN <Vb><d>, <Va><n>
```

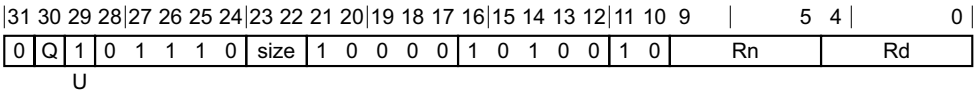
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
UQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

8H when size = 00

4S when size = 01

2D when size = 10

The encoding size = 11 is reserved.

<Vb> Is the destination width specifier, encoded in the "size" field. It can have the following values:

B when size = 00

H when size = 01

S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 00

S when size = 01

D when size = 10

The encoding size = 11 is reserved.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

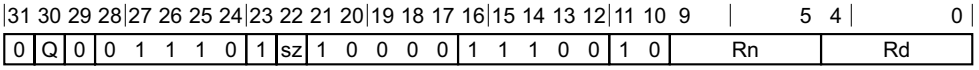
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.332 URECPE

Unsigned reciprocal estimate




### *Vector variant*

```
URECPE <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then ReservedValue();
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when `sz = 0, Q = 0`

4S when `sz = 0, Q = 1`

The encoding `sz = 1, Q = x` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;

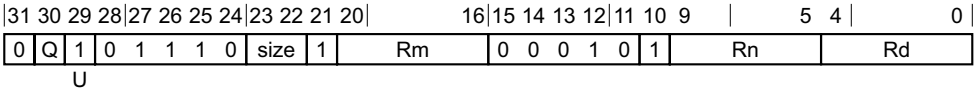
for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRecipEstimate(element);

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.333 URHADD

Unsigned rounding halving add




### *Three registers of the same type variant*

```
URHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1 + element2 + 1)<esize:1>;

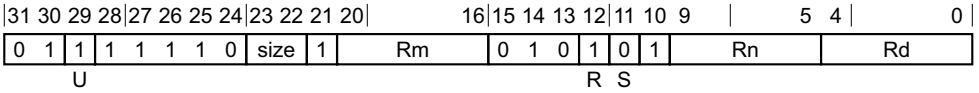
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.334 URSHL

Unsigned rounding shift left (register)

### Scalar



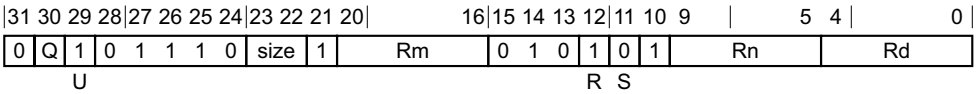

#### *Scalar variant*

```
URSHL <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

### Vector




#### *Vector variant*

```
URSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

`D` when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

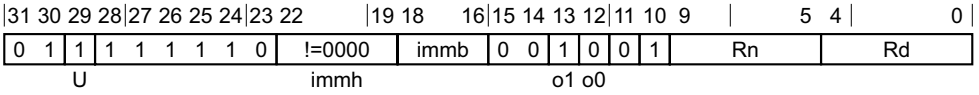
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.335 URSHR

Unsigned rounding shift right (immediate)

### Scalar




#### *Scalar variant*

```
URSHR <V><d>, <V><n>, #<shift>
```

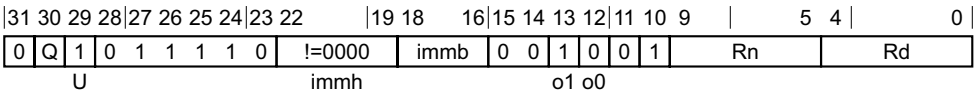
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector




#### *Vector variant*

```
URSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

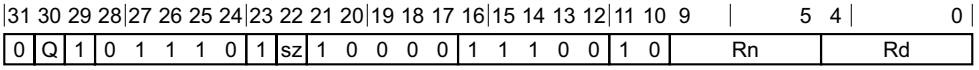
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.336 URSQRTE

Unsigned reciprocal square root estimate




### *Vector variant*

```
URSQRTE <Vd>.<T>, <Vn>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then ReservedValue();
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when `sz = 0, Q = 0`

4S when `sz = 0, Q = 1`

The encoding `sz = 1, Q = x` is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRSqrtEstimate(element);

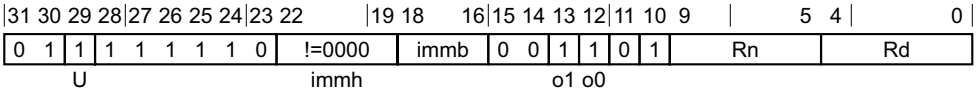
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.337 URSRA

Unsigned rounding shift right and accumulate (immediate)

### Scalar




#### *Scalar variant*

```
URSRA <V><d>, <V><n>, #<shift>
```

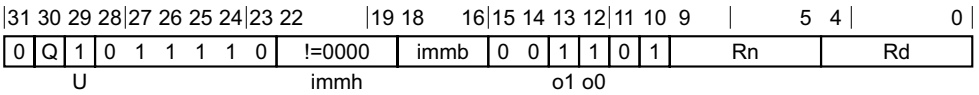
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector




#### *Vector variant*

```
URSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

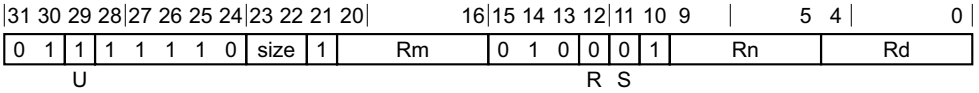
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.338 USHL

Unsigned shift left (register)

### Scalar



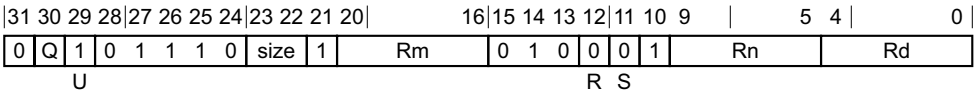

#### *Scalar variant*

```
USHL <V><d>, <V><n>, <V><m>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

### Vector




#### *Vector variant*

```
USHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

`D` when `size = 11`

The following encodings are reserved:

- `size = 0x`.
- `size = 10`.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

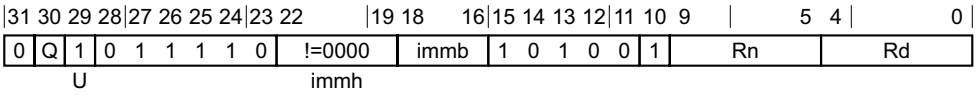
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.339 USHLL, USHLL2

Unsigned shift left long (immediate)

This instruction is used by the alias UXTL, UXTL2. See the *Alias conditions* table for details of when each alias is preferred.




### *Vector variant*

```
USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| UXTL, UXTL2 | `immb == '000' && BitCount(immh) == 1` |

## Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

(UInt(immh:immb)-8) when immh = 0001

(UInt(immh:immb)-16) when immh = 001x

(UInt(immh:immb)-32) when immh = 01xx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

The encoding immh = 1xxx is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

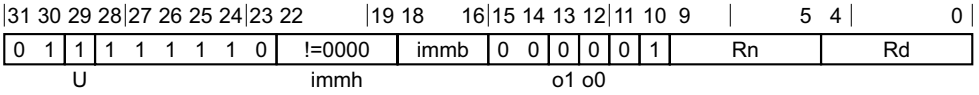
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.340 USHR

Unsigned shift right (immediate)

### Scalar




#### *Scalar variant*

```
USHR <V><d>, <V><n>, #<shift>
```

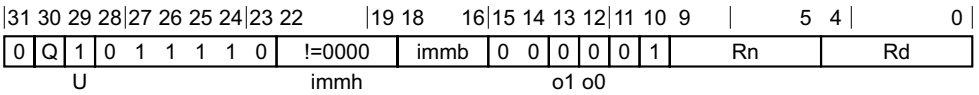
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector




#### *Vector variant*

```
USHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

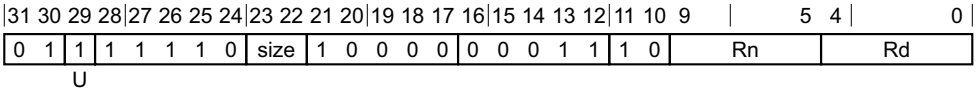
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.341 USQADD

Unsigned saturating accumulate of signed value

### Scalar




#### *Scalar variant*

```
USQADD <V><d>, <V><n>
```

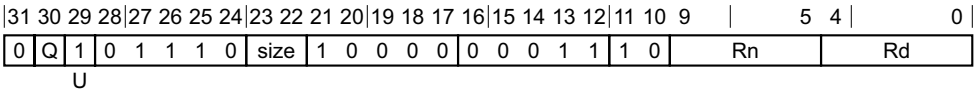
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector




#### *Vector variant*

```
USQADD <Vd>.<T>, <Vn>.<T>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| B | when size = 00 |
| H | when size = 01 |
| S | when size = 10 |
| D | when size = 11 |

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0

16B when size = 00, Q = 1

4H when size = 01, Q = 0

8H when size = 01, Q = 1

2S when size = 10, Q = 0

4S when size = 10, Q = 1

2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

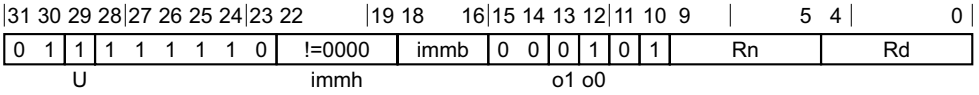
for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.342 USRA

Unsigned shift right and accumulate (immediate)

### Scalar




#### *Scalar variant*

```
USRA <V><d>, <V><n>, #<shift>
```

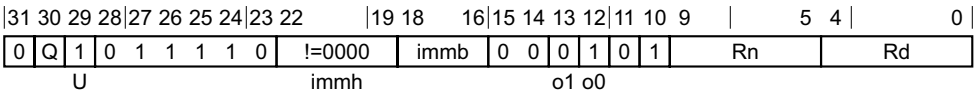
#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector




#### *Vector variant*

```
USRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

#### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Assembler symbols

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when `immh = 1xxx`

The encoding `immh = 0xxx` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0

16B when immh = 0001, Q = 1

4H when immh = 001x, Q = 0

8H when immh = 001x, Q = 1

2S when immh = 01xx, Q = 0

4S when immh = 01xx, Q = 1

2D when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

(128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

(16-UInt(immh:immb)) when immh = 0001

(32-UInt(immh:immb)) when immh = 001x

(64-UInt(immh:immb)) when immh = 01xx

(128-UInt(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-231 when immh = 0000.

## Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

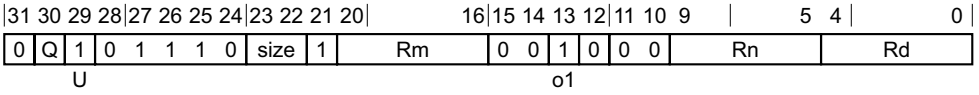
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.343 USUBL, USUBL2

Unsigned subtract long




### *Three registers, not all the same type variant*

```
USUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

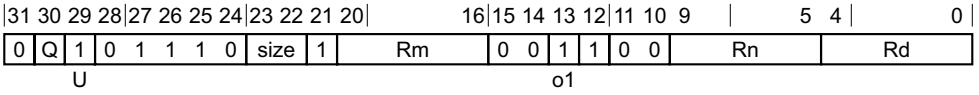
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.344 USUBW, USUBW2

Unsigned subtract wide




### *Three registers, not all the same type variant*

```
USUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding `size = 11, Q = x` is reserved.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize)   operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

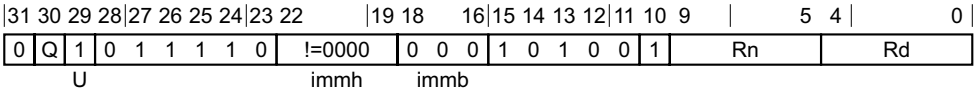

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.345 UXTL, UXTL2

Unsigned extend long

This instruction is an alias of the USHLL, USHLL2 instruction. This means that:

- The encodings in this description are named to match the encodings of USHLL, USHLL2.
- The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.




### *Vector variant*

`UXTL{2} <Vd>.<Ta>, <Vn>.<Tb>`

is equivalent to

`USHLL{2}  <Vd>.<Ta>, <Vn>.<Tb>, #0`

and is the preferred disassembly when `BitCount(immh) == 1`.

### Assembler symbols

`2` Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

`[absent]` when `Q = 0`

`[present]` when `Q = 1`

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

`8H` when `immh = 0001`

`4S` when `immh = 001x`

`2D` when `immh = 01xx`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000`.

The encoding `immh = 1xxx` is reserved.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

`8B` when `immh = 0001, Q = 0`

`16B` when `immh = 0001, Q = 1`

`4H` when `immh = 001x, Q = 0`

`8H` when `immh = 001x, Q = 1`

`2S` when `immh = 01xx, Q = 0`

`4S` when `immh = 01xx, Q = 1`

See *Advanced SIMD modified immediate* on page C4-231 when `immh = 0000, Q = x`.

The encoding `immh = 1xxx, Q = x` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

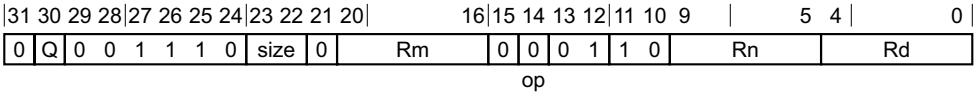
### Operation

The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.346 UZP1

Unzip vectors (primary)




### *Advanced SIMD variant*

```
UZP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when `size = 00, Q = 0`
- 16B when `size = 00, Q = 1`
- 4H when `size = 01, Q = 0`
- 8H when `size = 01, Q = 1`
- 2S when `size = 10, Q = 0`
- 4S when `size = 10, Q = 1`
- 2D when `size = 11, Q = 1`

The encoding `size = 11, Q = 0` is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;
integer e;

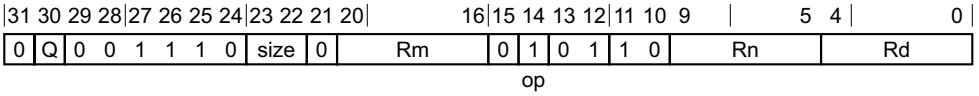
bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.347 UZP2

Unzip vectors (secondary)




### *Advanced SIMD variant*

```
UZP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;
integer e;

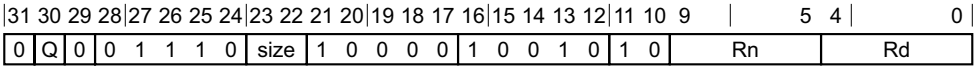
bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.348 XTN, XTN2

Extract narrow




### *Vector variant*

```
XTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- `[absent]` when `Q = 0`
- `[present]` when `Q = 1`

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> — Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- `8B` when `size = 00, Q = 0`
- `16B` when `size = 00, Q = 1`
- `4H` when `size = 01, Q = 0`
- `8H` when `size = 01, Q = 1`
- `2S` when `size = 10, Q = 0`
- `4S` when `size = 10, Q = 1`

The encoding `size = 11, Q = x` is reserved.

<Vn> — Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> — Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- `8H` when `size = 00`
- `4S` when `size = 01`
- `2D` when `size = 10`

The encoding `size = 11` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

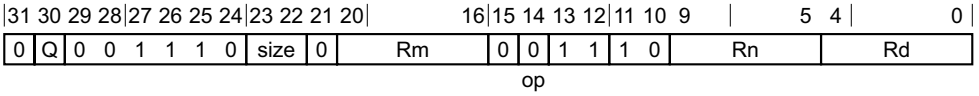
```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.349 ZIP1

Zip vectors (primary)




### *Advanced SIMD variant*

```
ZIP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when size = 00, Q = 0 |
| 16B | when size = 00, Q = 1 |
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |
| 2D | when size = 11, Q = 1 |

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;
integer p;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

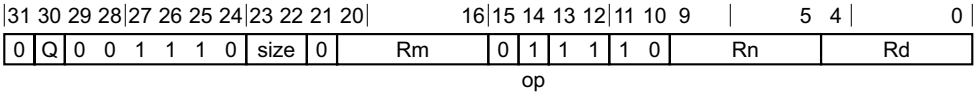
```
            Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

        V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## C7.3.350 ZIP2

Zip vectors (secondary)




### *Advanced SIMD variant*

```
ZIP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

### *Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

## Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| | |
|---|---|
| 8B | when size = 00, Q = 0 |
| 16B | when size = 00, Q = 1 |
| 4H | when size = 01, Q = 0 |
| 8H | when size = 01, Q = 1 |
| 2S | when size = 10, Q = 0 |
| 4S | when size = 10, Q = 1 |
| 2D | when size = 11, Q = 1 |

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;
integer p;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

        V[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Part D

## The AArch64 System Level Architecture

# Chapter D1
# The AArch64 System Level Programmers' Model

This chapter describes the AArch64 system level programmers' model. It contains the following sections:


- *Exception levels* on page D1-1514.
- *Exception terminology* on page D1-1515.
- *Execution state* on page D1-1517.
- *Security state* on page D1-1518.
- *Virtualization* on page D1-1520.
- *Registers for instruction processing and exception handling* on page D1-1523.
- *Process state, PSTATE* on page D1-1529.
- *Program counter and stack pointer alignment* on page D1-1531.
- *Reset* on page D1-1533.
- *Exception entry* on page D1-1537.
- *Exception return* on page D1-1552.
- *The Exception level hierarchy* on page D1-1556.
- *Synchronous exception types, routing and priorities* on page D1-1563.
- *Asynchronous exception types, routing, masking and priorities* on page D1-1569.
- *Configurable instruction enables and disables, and trap controls* on page D1-1575.
- *System calls* on page D1-1610.
- *Mechanisms for entering a low-power state* on page D1-1611.
- *Self-hosted debug* on page D1-1616.
- *The Performance Monitors Extension* on page D1-1618.
- *Interprocessing* on page D1-1619.
- *The effect of implementation choices on the programmers' model* on page D1-1631.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D1.1 Exception levels

The ARMv8-A architecture defines a set of Exception levels, EL0 to EL3, where:

- If ELn is the Exception level, increased values of *n* indicate increased software execution privilege.
- Execution at EL0 is called *unprivileged execution*.
- EL2 provides support for virtualization of Non-secure operation.
- EL3 provides support for switching between two Security states, Secure state and Non-secure state.

An implementation might not include all of the Exception levels. All implementations must include EL0 and EL1. EL2 and EL3 are optional.

**Note**

A PE is not required to implement a contiguous set of Exception levels. For example, it is permissible for an implementation to include only EL0, EL1, and EL3.

*The effect of implementation choices on the programmers' model* on page D1-1631 shows some example implementations.

When executing in AArch64 state, execution can move between Exception levels only on taking an exception or on returning from an exception:

- On taking an exception, the Exception level can only increase or remain the same.
- On returning from an exception, the Exception level can only decrease or remain the same.

The Exception level that execution changes to or remains in on taking an exception is called the *target Exception level* of the exception.

Each exception type has a target Exception level that is either:

- Implicit in the nature of the exception.
- Defined by configuration bits in the System registers.

An exception cannot target EL0.

Exception levels exist within a particular Security state. *The ARMv8-A security model* on page D1-1518 describes this. When executing at an Exception level, the PE can access both of the following:

- The resources that are available for the combination of the current Exception level and the current Security state.
- The resources that are available at all lower Exception levels, provided that those resources are available to the current Security state.

This means that if the implementation includes EL3, then when execution is at EL3, the PE can access all resources available at all Exception levels, for both Security states.

Each Exception level other than EL0 has its own translation regime and associated control registers. For information on the translation regimes, see Chapter D4 *The AArch64 Virtual Memory System Architecture*.

### D1.1.1 Typical Exception level usage model

The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

**EL0** Applications.

**EL1** OS kernel and associated functions that are typically described as *privileged*.

**EL2** Hypervisor.

**EL3** Secure monitor.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.2 Exception terminology

The following subsections define the terms used when describing exceptions:

- *Terminology for taking an exception*.
- *Terminology for returning from an exception*.
- *Exception levels*.
- *Definition of a precise exception*.
- *Definitions of synchronous and asynchronous exceptions* on page D1-1516.

### D1.2.1 Terminology for taking an exception

An exception is *generated* when the PE first responds to an exceptional condition.The PE state at this time is the state the exception is *taken from*. The PE state immediately after taking the exception is the state the exception is *taken to*.

### D1.2.2 Terminology for returning from an exception

To return from an exception, the PE must execute an exception return instruction.The PE state when an exception return instruction is committed for execution is the state the exception *returns from*. The PE state immediately after the execution of that instruction is the state the exception *returns to*.

### D1.2.3 Exception levels

An Exception level, EL*n*, with a larger value of *n* than another Exception level, is described as being a *higher* Exception level than the other Exception level. For example, EL3 is a higher Exception level than EL1.

An Exception level with a smaller value of *n* than another Exception level is described as being a *lower* Exception level than the other Exception level. For example, EL0 is a lower Exception level than EL1.

An Exception level is described as:

- *Using AArch64* when execution in that Exception level is in the AArch64 Execution state.
- *Using AArch32* when execution in that Exception level is in the AArch32 Execution state.

### D1.2.4 Definition of a precise exception

An exception is described as *precise* when the exception handler receives the PE state and memory system state that is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken, and none afterwards.

Other than the *SError interrupt*, all exceptions taken to AArch64 state are required to be precise. For each occurrence of an SError interrupt, whether the interrupt is precise or imprecise is IMPLEMENTATION DEFINED

Where a synchronous exception that is taken to AArch64 state is generated as part of an instruction that performs more than one single-copy atomic memory access, the definition of precise permits that the values in registers or memory affected by the instructions can be UNKNOWN, provided that:

- The accesses affecting those registers or memory locations do not, themselves, generate exceptions.
- The registers are not involved in the calculation of the memory address used by the instruction.

Examples of instructions that perform more than one single-copy atomic memory access are the AArch32 `LDM` and `STM` instructions and the AArch64 `LDP` and `STP` instructions.

**Note**

- For the definition of a single-copy atomic access, see *Properties of single-copy atomic accesses* on page B2-81.
- SError interrupts are known as Asynchronous Aborts in AArch32 state.
- By definition, all synchronous aborts are precise.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D1.2.5 Definitions of synchronous and asynchronous exceptions

An exception is described as *synchronous* if all of the following apply:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The return address presented to the exception handler is guaranteed to indicate the instruction that caused the exception.
- The exception is precise.

For more information about synchronous exceptions, see *Synchronous exception types, routing and priorities* on page D1-1563.

An exception is described as *asynchronous* if any of the following apply:

- The exception is not generated as a result of direct execution or attempted execution of the instruction stream.
- The return address presented to the exception handler is not guaranteed to indicate the instruction that caused the exception.
- The exception is imprecise.

For more information about asynchronous exceptions, see *Asynchronous exception types, routing, masking and priorities* on page D1-1569.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.3 Execution state

The Execution states are:

**AArch64** The 64-bit Execution state.

**AArch32** The 32-bit Execution state. Operation in this state is compatible with ARMv7-A operation.

*Execution state* on page A1-33 gives more information about them.

Exception levels *use* Execution states. For example, EL0, EL1 and EL2 might all be using AArch32, under EL3 using AArch64.

This means that:

- Different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.
- The PE can change Execution states only either:
  - At reset.
  - On a change of Exception level.

**Note**

- *Typical Exception level usage model* on page D1-1514 shows which Exception levels different software layers might typically use.
- *The effect of implementation choices on the programmers' model* on page D1-1631 gives information on supported configurations of Exception levels and Execution states.

The interaction between the AArch64 and AArch32 Execution states is called *interprocessing*. *Interprocessing* on page D1-1619 describes this.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.4 Security state

The ARMv8-A architecture provides two Security states, each with an associated physical memory address space, as follows:

**Secure state** When in this state, the PE can access both the Secure physical address space and the Non-secure physical address space.

**Non-secure state** When in this state, the PE:

- Can access only the Non-secure physical address space.
- Cannot access the Secure system control resources.

For information on how virtual addresses translate onto Secure physical and Non-secure addresses, see *About the Virtual Memory System Architecture (VMSA)* on page D4-1742.

### D1.4.1 The ARMv8-A security model

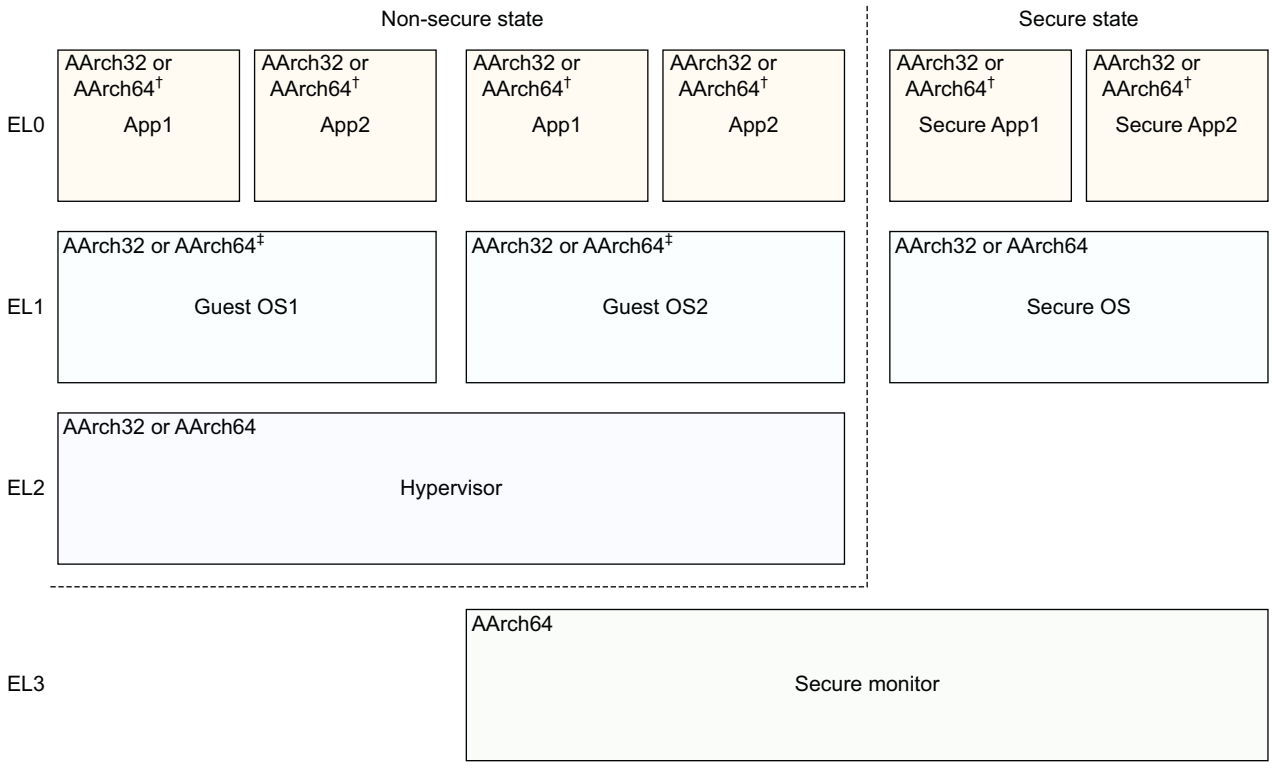
The general principles of the ARMv8-A security model are:

- If the implementation includes EL3 then it has two Security states, Secure and Non-secure, and:
  - EL3 exists only in Secure state.
  - A change from Non-secure state to Secure state can only occur on taking an exception to EL3.
  - A change from Secure state to Non-secure state can only occur on an exception return from EL3.
  - If EL2 is implemented, it exists only in Non-secure state.
- If the implementation does not include EL3 it has one Security state, that is:
  - IMPLEMENTATION DEFINED, if the implementation does not include EL2.
  - Non-secure state if the implementation includes EL2.

#### Security model when EL3 is using AArch64

Figure D1-1 on page D1-1519 shows the security model when EL3 is using AArch64. The figure shows how instances of EL0 and EL1 are present in both Security states. It also shows the expected software usage of the different Exception levels.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




† AArch64 permitted only if EL1 is using AArch64
‡ AArch64 permitted only if EL2 is using AArch64

**Figure D1-1 ARMv8-A security model when EL3 is using AArch64**

For an overview of the Security model when EL3 is using AArch32, see Figure G1-1 on page G1-3754.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.5 Virtualization

The support for virtualization described in this section applies only to an implementation that includes EL2.

EL2 provides a set of features that support virtualizing the Non-secure state of an ARMv8-A implementation. The basic model of a virtualized system involves:

- A hypervisor, running in EL2, that is responsible for switching between *virtual machines*. A virtual machine comprises Non-secure EL1 and Non-secure EL0.
- A number of Guest operating systems. A Guest OS runs on a virtual machine in Non-secure EL1.
- For each Guest operating system, applications, that run on the virtual machine of that Guest OS, usually in Non-secure EL0.

**Note**

In some systems, a Guest OS is unaware that it is running on a virtual machine, and is unaware of any other Guest OS. In other systems, a hypervisor makes the Guest OS aware of these facts. The ARMv8-A architecture supports both of these models.

The hypervisor assigns a *virtual machine identifier* (VMID) to each virtual machine.

EL2 is implemented only in Non-secure state, to support Guest OS management. EL2 provides controls to:

- Provide virtual values for the contents of a small number of identification registers. A read of one of these registers by a Guest OS or the applications for a Guest OS returns the virtual value.
- *Trap* various operations, including memory management operations and accesses to many other registers. A trapped operation generates an exception that is taken to EL2. See *Configurable instruction enables and disables, and trap controls* on page D1-1575.
- Route interrupts to the appropriate one of:
  - The current Guest OS.
  - A Guest OS that is not currently running.
  - The hypervisor.

In Non-secure state:

- The implementation provides an independent *translation regime* for memory accesses from EL2.
- For the EL1&0 translation regime, address translation occurs in two stages:
  - Stage 1 maps the *virtual address* (VA) to an *intermediate physical address* (IPA). This is managed at EL1, usually by a Guest OS. The Guest OS believes that the IPA is the *physical address* (PA).
  - Stage 2 maps the IPA to the PA. This is managed at EL2. The Guest OS might be completely unaware of this stage.

For more information on the translation regimes, see Chapter D4 *The AArch64 Virtual Memory System Architecture*.

### D1.5.1 The effect of implementing EL2 on the Exception model

An implementation that includes EL2 implements the following exceptions:

- Hypervisor Call (HVC) exception.
- Traps to EL2. *EL2 configurable controls* on page D1-1584, describes these.
- All of the virtual interrupts:
  - Virtual SError.
  - Virtual IRQ.
  - Virtual FIQ.

Hypervisor call exceptions taken from EL3 are taken to EL3. Otherwise, Hypervisor call exceptions are taken from Non-secure state to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All virtual interrupts are always taken to Non-secure EL1, and can only be taken from Non-secure EL1 or EL0.

Each of the virtual interrupts can be independently enabled using controls at EL2.

Each of the virtual interrupts has a corresponding physical interrupt. See *Virtual interrupts*.

When a virtual interrupt is enabled, in Non-secure state its corresponding physical exception is taken to EL2, unless EL3 has configured that physical exception to be taken to EL3.

For more information, see *Asynchronous exception types, routing, masking and priorities* on page D1-1569.

An implementation that includes EL2 also:

- Provides controls that can be used to route some synchronous exceptions, taken from Non-secure state, to EL2. For more information see:
  - *Routing exceptions to EL2* on page D1-1564.
  - *Routing debug exceptions* on page D2-1641.
- Provides mechanisms to trap Non-secure PE operations to EL2. See *EL2 configurable controls* on page D1-1584.

  When an operation is trapped to EL2, the hypervisor typically either:
  - Emulates the required operation. The application running in the Guest OS is unaware of the trap.
  - Returns an error to the Guest OS.

## Virtual interrupts

The virtual interrupts have names that correspond to the physical interrupts, as shown in Table D1-1.

**Table D1-1 The virtual interrupt**

| Physical interrupt | Corresponding virtual interrupt |
|---|---|
| SError | Virtual SError |
| IRQ | Virtual IRQ |
| FIQ | Virtual FIQ |

Software executing in EL2 can use virtual interrupts to signal physical interrupts to Non-secure EL1 and Non-secure EL0. Example D1-1 shows a usage model for virtual interrupts.

**Example D1-1 Virtual interrupt usage model**

A usage model is as follows:

1. Software executing at EL2 routes a physical interrupt to EL2.
2. When a physical interrupt of that type occurs, the exception handler executing in EL2 determines whether the interrupt can be handled in EL2 or requires routing to a Guest OS in EL1. If the interrupt requires routing to a Guest OS:
   - If the Guest OS is currently running, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.
   - If the Guest OS is not currently running, the physical interrupt is marked as pending for the guest OS. When the hypervisor next switches to the virtual machine that is running that Guest OS, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.

A hypervisor can prevent Non-secure EL1 and Non-secure EL0 from distinguishing a virtual interrupt from a physical interrupt.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information see:

- *Asynchronous exception types, routing, masking and priorities* on page D1-1569.
- *Virtual interrupts* on page D1-1572.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.6 Registers for instruction processing and exception handling

In the ARM architecture, registers fall into two main categories:

- Registers that provide system control or status reporting. These are described in Chapter D7 *AArch64 System Register Descriptions*.
- Registers that are used in instruction processing, for example to accumulate a result, and in handling exceptions. This section introduces these registers, for execution in AArch64 state.

This section contains the following subsections:

- *The general purpose registers, R0-R30*.
- *The stack pointer registers*.
- *The SIMD and floating-point registers, V0-V31* on page D1-1524.
- *Saved Program Status Registers (SPSRs)* on page D1-1524.
- *Exception Link Registers (ELRs)* on page D1-1527.

### D1.6.1 The general purpose registers, R0-R30

The general purpose register bank is used when processing instructions in the base instruction set. It comprises 31 general purpose registers, R0-R30.

These registers can be accessed as 31 64-bit registers, X0-X30, or 31 32-bit registers, W0-W30. See *Register size* on page C6-427.

For information on the format of these registers, see *Registers in AArch64 state* on page B1-59.

### D1.6.2 The stack pointer registers

In AArch64 state, in addition to the general purpose registers, a dedicated stack pointer register is implemented for each implemented Exception level. The stack pointer registers are:

- SP_EL0 and SP_EL1.
- If the implementation includes EL2, SP_EL2.
- If the implementation includes EL3, SP_EL3.

**Note**

The four stack pointer register names define an architecture state requirement for four registers. For information on how to access these registers, and access restrictions, see *Special-purpose registers* on page C5-289.

For information on stack pointer alignment restrictions, see *Stack pointer alignment checking* on page D1-1531.

#### Stack pointer register selection

When executing at EL0, the PE uses the EL0 stack pointer, SP_EL0.

When executing at any other Exception level, the PE can be configured to use either SP_EL0 or the stack pointer for that Exception level, SP_ELx.

By default, taking an exception selects the stack pointer for the target Exception level, SP_ELx. For example, taking an exception to EL1 selects SP_EL1. Software executing at the target Exception level can then choose to change the stack pointer to SP_EL0 by updating PSTATE.SP.

This applies even if taking the exception does not change the Exception level. For example, if the PE is executing at EL1 and the PE is using the SP_EL0 stack pointer, then on taking an exception that targets EL1, the stack pointer changes to SP_EL1.

The selected stack pointer can be indicated by a suffix to the Exception level:

**t** Indicates use of the SP_EL0 stack pointer.

**h** Indicates use of the SP_ELx stack pointer.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The t and h suffixes are based on the terminology of *thread* and *handler*, introduced in ARMv7-M

Table D1-2 shows the set of stack pointer options.

**Table D1-2 AArch64 stack pointer options**

| Exception level | AArch64 stack pointer options |
|---|---|
| EL0 | EL0t |
| EL1 | EL1t, EL1h |
| EL2 | EL2t, EL2h |
| EL3 | EL3t, EL3h |

### D1.6.3 The SIMD and floating-point registers, V0-V31

The SIMD and floating-point instructions share a common bank of registers for floating-point, vector, and other SIMD-related scalar operations.

The SIMD and floating-point register bank comprises 32 quadword (128-bit) registers, V0-V31.

These registers can be accessed as:

- 32 doubleword (64-bit) registers, D0-D31.
- 32 word (32-bit) registers, S0-S31.
- 32 halfword (16-bit) registers, H0-H31.
- 32 byte (8-bit) registers, B0-B31.

For information on the format of these registers, see *Registers in AArch64 state* on page B1-59.

### D1.6.4 Saved Program Status Registers (SPSRs)

The *Saved Program Status Registers* (SPSRs) are used to save PE state on taking exceptions.

In AArch64 state, there is an SPSR at each Exception level exceptions can be taken to, as follows:

- SPSR_EL1, for exceptions taken to EL1 using AArch64.
- If EL2 is implemented, SPSR_EL2, for exceptions taken to EL2 using AArch64.
- If EL3 is implemented, SPSR_EL3, for exceptions taken to EL3 using AArch64.

**Note**

Exceptions cannot be taken to EL0.

When the PE takes an exception, the PE state is saved from PSTATE in the SPSR at the Exception level the exception is taken to. For example, if the PE takes an exception to EL1, the PE state is saved in SPSR_EL1. For more information on PSTATE, see *Process state, PSTATE* on page D1-1529.

Saving the PE state means the exception handler can:

- On return from the exception, restore the PE state to the state stored in the SPSR at the Exception level the exception is returning from. For example, on returning from EL1, the PE state is restored to the state stored in SPSR_EL1.
- Examine the value that PSTATE had when the exception was taken, for example to determine the Execution state and Exception level in which the instruction that caused an Undefined Instruction exception was executed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

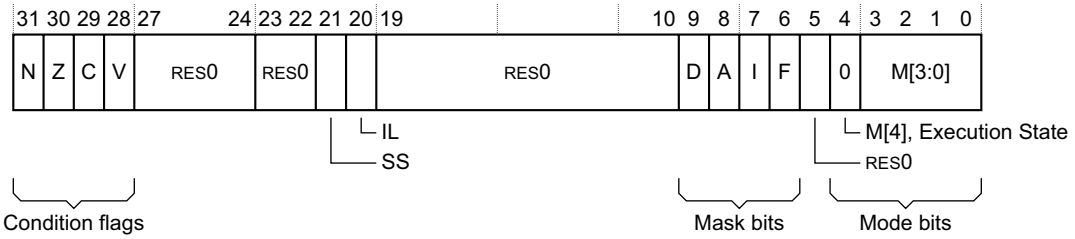
- All PSTATE fields are saved, including those which have no direct read and write access, and those that are meaningful only in AArch32 state.
- Those PSTATE fields that are meaningful only in AArch32 state are saved when an exception is taken from AArch32 state to AArch64 state.

The SPSRs are UNKNOWN on reset.

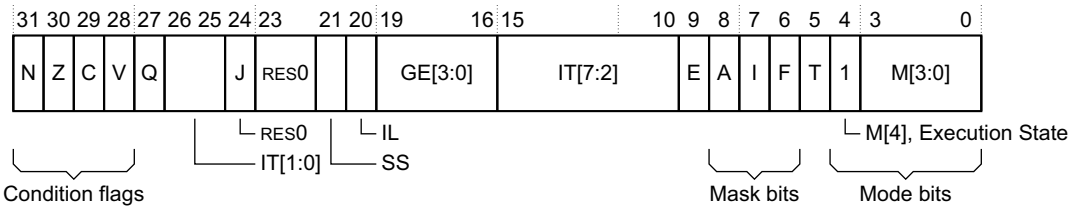
### SPSR format for exceptions taken to AArch64 state

Exceptions can be taken to AArch64 state from AArch64 state or AArch32 state:

- For an exception taken to AArch64 state from AArch64 state, the SPSR bit assignments are:




- For an exception taken to AArch64 state from AArch32 state, the SPSR bit assignments are:




The following list describes the bit assignments:

**N, Z, C, V, bits[31:28]**

Shows the values of the PSTATE.{N, Z, C, V} condition flags immediately before the exception was taken.

**Bits[27:22]** Reserved, RES0, for exceptions taken from AArch64 state.

For exceptions taken from AArch32 state:

**Q, bit [27]**

Shows the value of PSTATE.Q immediately before the exception was taken.

**IT[1:0], bits [26:25]**

See *Bits[19:10]* on page D1-1526.

**J, bit [24]**

Shows the value of PSTATE.J immediately before the exception was taken. This bit is RES0.

For the definitions of the Q, IT, and J fields, see *Process state, PSTATE* on page G1-3767.

**SS, bit[21]** The Software Step bit.

SPSR_ELx.SS is used by a debugger to initiate a Software Step exception. The SS bit also indicates which software step state machine state the PE was in. See *Software Step exceptions* on page D2-1686.

**IL, bit[20]** Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken. See *Illegal return events from AArch64 state* on page D1-1553.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits[19:10]** Reserved, RES0, for exceptions taken from AArch64 state.

For exceptions taken from AArch32 state:

**GE[3:0], bits [19:16]**
Shows the value of PSTATE.GE immediately before the exception was taken.

**IT[7:2], bits [15:10]**
In conjunction with IT[1:0], shows the value of PSTATE.IT before the exception was taken.

For definitions of the GE and IT fields, see *Process state, PSTATE* on page G1-3767.

**Bit[9]** D, the debug exception mask bit, for exceptions taken from AArch64 state. Shows the value of PSTATE.D immediately before the exception was taken. See *The PSTATE debug mask bit, D* on page D1-1616.

E, for exceptions taken from AArch32 state. Shows the value of PSTATE.E immediately before the exception was taken. For the definition of the E bit, see *Process state, PSTATE* on page G1-3767.

**A, I, F, bits[8:6]**

Shows the values of the PSTATE.{A,I, F} exception mask bits immediately before the exception was taken:

**A, bit[8]** SError interrupt mask bit.

**I, bit[7]** IRQ mask bit.

**F, bit[6]** FIQ mask bit.

See *Asynchronous exception masking* on page D1-1570.

**Bit[5]** Reserved, RES0, for exceptions taken from AArch64 state.

T, for exceptions taken from AArch32 state. Shows the value of PSTATE.T immediately before the exception was taken.

**M[4:0], bits[4:0]**

Mode field.

**Note**

The name of this field is inherited from ARMv7, where the M field specified the PE *mode.*

For exceptions taken from AArch64 state:

**M[4]** The value of this is 0. M[4] encodes the value of PSTATE.nRW.

**M[3:0]** Encodes the Exception level and the stack pointer register selection, as shown in Table D1-3.

**Table D1-3 M[3:0] encodings, for exceptions taken from AArch64 state**

| M[3:0][a] | Exception level and stack pointer |
|---|---|
| 0b1101 | EL3h |
| 0b1100 | EL3t |
| 0b1001 | EL2h |
| 0b1000 | EL2t |
| 0b0101 | EL1h |
| 0b0100 | EL1t |
| 0b0000 | EL0t |

a. All M[3:0] encodings not shown in the table are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The M[3:0] encoding comprises:

**M[3:2]** Encodes the Exception level, 0-3.

**M[1]** Reserved, RES0. If set to 1 at the time of an exception return, then that exception return is treated as an Illegal Execution state exception return.

**M[0]** Selects the SP:

- **0** SP_EL0. Indicated by a t suffix on the Exception level.
- **1** SP_ELx, where *x* is the value of M[3:2]. Indicated by an h suffix on the Exception level.

See *Stack pointer register selection* on page D1-1523.

For exceptions taken from AArch32 state:

**M[4]** The value of this is 1. M[4] encodes the value of PSTATE.nRW.

**M[3:0]** Encodes the AArch32 mode that the PE was in immediately before the exception was taken, as shown in Table D1-4.

**Table D1-4 M[3:0] encodings, for exceptions taken from AArch32 state**

| M[3:0] | AArch32 mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0111 | Abort |
| 0b1011 | Undefined |
| 0b1111 | System |

Bits [27:22] and [19:10] of an SPSR are ignored on an exception return to AArch64 state. Bits [23:22] of an SPSR are ignored on an exception return to AArch32 state.

### Pseudocode description of SPSR operations

The SPSR[] pseudocode function accesses the current SPSR, and is common to AArch32 and AArch64 operations.

The SetPSTATEFromPSR() pseudocode function updates PSTATE from an SPSR.

## D1.6.5 Exception Link Registers (ELRs)

Exception Link Registers hold preferred exception return addresses.

Whenever the PE takes an exception, the preferred return address is saved in the ELR at the Exception level the exception is taken to. For example, whenever the PE takes an exception to EL1, the preferred return address is saved in ELR_EL1.

On an exception return, the PC is restored to the address stored in the ELR. For example, on returning from EL1, the PC is restored to the address stored in ELR_EL1.

AArch64 state provides an ELR for each Exception level exceptions can be taken to. The ELRs that AArch64 state provides are:

- ELR_EL1, for exceptions taken to EL1.
- If EL2 is implemented, ELR_EL2, for exceptions taken to EL2.
- If EL3 is implemented, ELR_EL3, for exceptions taken to EL3.

On taking an exception from AArch32 state to AArch64 state, bits[63:32] of the ELR are set to zero.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The preferred return address depends on the nature of the exception. For more information, see *Preferred exception return address* on page D1-1537.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D1.7 Process state, PSTATE

In the ARMv8-A architecture, Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

PSTATE includes all of the following:

- Fields that are meaningful only in AArch32 state.
- Fields that are meaningful only in AArch64 state.
- Fields that are meaningful in both Execution states.

PSTATE is defined in pseudocode as the `PSTATE` structure, of type `ProcState`. `ProcState` is defined in Chapter J1 *ARMv8 Pseudocode*.

The PSTATE fields that are meaningful in AArch64 state are:

**The condition flags**

**N** Negative Condition flag.

**Z** Zero Condition flag.

**C** Carry Condition flag.

**V** Overflow Condition flag.

*Process state, PSTATE* on page B1-61 gives more information about these flags.

**The Execution state controls**

**SS** Software Step bit, see *Software Step exceptions* on page D2-1686. On a reset or taking an exception to AArch64 state, this bit is set to 0.

**IL** Illegal Execution state bit, see *The Illegal Execution state exception* on page D1-1555. On a reset or taking an exception to AArch64 state, this bit is set to 0.

**nRW** Current Execution state, see *Execution state* on page D1-1517. This bit is 0 when the current Execution state is AArch64. This bit is set to 0:

- On reset into an Exception level that is using AArch64.
- On taking an exception to an Exception level that is using AArch64.

**EL** Current Exception level, see *Exception levels* on page D1-1514. On a reset to AArch64 state, this field holds the encoding for the highest implemented Exception level.

**Note**

The ARM architecture requires that a PE resets into the highest implemented Exception level.

**SP** Stack pointer register selection bit, see *Stack pointer register selection* on page D1-1523. On a reset or taking an exception to AArch64 state, this bit is set to 1, meaning that SP_ELx is selected.

**The exception mask bits**

**D** Debug exception mask bit, see *The PSTATE debug mask bit, D* on page D1-1616. On a reset or taking an exception to AArch64 state, this bit is set to 1.

**A, I, F** Asynchronous exception mask bits:

**A** SError interrupt mask bit.

**I** IRQ interrupt mask bit.

**F** FIQ interrupt mask bit.

See *Asynchronous exception types, routing, masking and priorities* on page D1-1569. On a reset or taking an exception to AArch64 state, each of these bits is set to 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D1.7.1 Accessing PSTATE fields

In AArch64 state, PSTATE fields can be accessed using Special-purpose registers that can be directly read using the MRS instruction, and directly written using the MSR (register) instructions. Table D1-5 shows the Special-purpose registers that access the PSTATE fields that hold AArch64 state, when the PE is in AArch64 state. All other PSTATE fields do not have direct read and write access.

**Table D1-5 Accessing PSTATE fields using MRS and MSR (register)**

| Special-purpose register | PSTATE fields |
|---|---|
| NZCV | N, Z, C, V |
| DAIF | D, A, I, F |
| CurrentEL | EL |
| SPSel | SP |

Software can also use the MSR (immediate) instruction to directly write to PSTATE.{D, A, I, F, SP}. Table D1-6 shows the MSR (immediate) operands that can directly write to these PSTATE fields when the PE is in AArch64 state.

**Table D1-6 Accessing PSTATE.{D, A, I, F, SP} using MSR (immediate)**

| Operand | PSTATE fields | Notes |
|---|---|---|
| DAIFSet | D, A, I, F | Directly sets any of the PSTATE.{D,A, I, F} bits to 1 |
| DAIFClr | D, A, I, F | Directly clears any of the PSTATE.{D, A, I, F} bits to 0 |
| SPSel | SP | Directly sets PSTATE.SP to either 1 or 0 |

PSTATE.{N, Z, C, V} can be accessed at EL0. Access to PSTATE.{D, A, I, F} at EL0 using AArch64 depends on SCTLR_EL1.UMA, see *Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks* on page D1-1579. All other PSTATE access instructions can be executed at EL1 or higher and are UNDEFINED at EL0.

Writes to the PSTATE fields have side-effects on various aspects of the PE operation. All of these side-effects are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

### D1.7.2 The Saved Program Status Registers (SPSRs)

On taking an exception, PSTATE is preserved in the SPSR of the Exception level the exception is taken to. The SPSRs are described in *Saved Program Status Registers (SPSRs)* on page D1-1524.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.8 Program counter and stack pointer alignment

This section contains the following:

- *PC alignment checking*.
- *Stack pointer alignment checking*.

### D1.8.1 PC alignment checking

PC alignment checking generates an exception that is associated with instruction fetch if, in AArch64 state, there is an attempt to architecturally execute an instruction that was fetched with a misaligned PC. A misaligned PC is when bits[1:0] of the PC are not `0b00`.

**Note**

As with Instruction Aborts, speculative fetching of an instruction does not generate an exception. An exception occurs only on an attempt to architecturally execute the instruction.

If an exception is generated as a result of an instruction fetch at EL0, it is taken to EL1, unless the exception occurs in Non-secure state and HCR_EL2.TGE bit is 1, when it is taken to EL2 instead. If an exception is generated as a result of an instruction fetch at any other Exception level, the Exception level is unchanged.

A PC misalignment sets the EC field in the *Exception Syndrome Register* (ESR) to `0x22`, for the ESR associated with the target Exception level.

When the exception is taken to an Exception level using AArch64, the associated Exception Link Register holds the entire PC in its misaligned form, as does the FAR_ELx for the Exception level that the exception is taken to.

*Exception return and PC alignment* on page D1-1553 gives more information on PC alignment checking associated with exception returns.

**Note**

A misalignment of the PC is a common indication of a serious error, for example software corruption of an address.

The pseudocode function `AArch64.CheckPCAlignment()` performs PC alignment checking in AArch64 state. When necessary it calls `AArch64.PCAlignmentFault()` to generates an exception.

### D1.8.2 Stack pointer alignment checking

A *misaligned stack pointer* is where bits[3:0] of the stack pointer are not `0b0000`, when the stack pointer is used as the base address of the calculation, regardless of any offset applied by the instruction.

The PE can be configured so that if a load or store instruction uses a misaligned stack pointer, the PE generates an exception on the attempt to execute the instruction. In this configuration, `CheckSPAlignment()` performs the stack pointer check, and calls `AArch64.SPAlignmentFault()` if a misaligned stack pointer is found.

**Note**

- As with Data Aborts, a speculative data access to memory using the stack pointer does not generate the exception. The exception occurs only on an attempt to architecturally execute the instruction.
- Prefetch memory abort instructions do not cause synchronous exceptions. See *Prefetch memory* on page C3-151.

Stack pointer alignment checking is only performed in AArch64, and can be enabled for each Exception level as follows:

- SCTLR_EL1.{SA0, SA} controls EL0 and EL1, respectively
- SCTLR_EL2.SA controls EL2
- SCTLR_EL3.SA controls EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If an exception is generated as a result of a load or store at EL0, it is taken as an exception to EL1 unless the HCR_EL2.TGE bit is set in the Non-secure state, when it is taken to EL2. If an exception is generated as a result of a load or store at any other Exception level, the Exception level is unchanged.

A stack pointer misalignment sets the EC field to `0x26`, in the ESR associated with the target Exception level. If memory alignment checking and stack pointer alignment checking are enabled, then a stack pointer alignment fault has priority in setting the value of the EC field, in the ESR associated with the target Exception level.

The pseudocode function `CheckSPAlignment()` performs the stack pointer alignment check. When necessary it calls `AArch64.SPAlignmentFault()` to generate an exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.9 Reset

The ARMv8-A architecture supports the following resets:

**Cold reset** Resets all of the logic on which the PE executes, including the integrated debug functionality.

In some contexts, this logic is described as belonging to the *Cold reset domain*.

**Warm reset** Resets the logic on which the PE executes, but does not reset the integrated debug functionality.

In some contexts, this logic is described as belonging to the *Warm reset domain*.

**Note**

The ARMv8-A architecture also supports an *external debug reset*. See *External debug register resets* on page H8-4963.

The difference between a Cold reset and a Warm reset is relevant only to the debug functionality and the RMR_ELx register, if an RMR_ELx register is implemented:

- A Warm reset permits debugging across a reset of the PE logic.
- Writing 1 to RMR_ELx.RR requests a Warm reset.

The mechanisms, other than RMR_ELx.RR, to assert these resets are IMPLEMENTATION DEFINED. It is IMPLEMENTATION DEFINED whether:

- It is possible to independently assert an External Debug reset and a Cold reset.
- It is possible to assert a Warm reset, as opposed to asserting a Cold reset, other than by the use of RMR_ELx.RR.

**Note**

ARM recommends that:

- If separate Core and Debug power domains are implemented, as described in *Reset and debug* on page H6-4935, then a Cold reset can be asserted independently of External Debug reset.
- A Warm reset can be asserted to permit debugging across a reset of the PE logic.

This means that an implementation can define other resets according to the requirements the implementation or system must fulfil. These other resets are outside the scope of the ARMv8-A architecture. However, they can be mapped on to the resets described here.

In the description that follows, the term *reset* is used in contexts where there is no difference between the effect of a Cold reset and the effect of a Warm reset.

On a reset, the PE enters the highest implemented Exception level.

If the highest implemented Exception level can use either Execution state, then:

- The implementation must include a *Reset Management Register* (RMR). Only one RMR is implemented. The RMR implemented is the RMR is associated with the highest Exception level.
- On a Cold reset, the Execution state entered is determined by a configuration input signal.
- On a Warm reset, the Execution state entered is determined by RMR_ELx.AA64.

If the highest implemented Exception level is configured to use AArch64 state, then on reset:

- The stack pointer for the highest implemented Exception level, SP_ELx, is selected.
- Execution starts at an IMPLEMENTATION DEFINED address, anywhere in the physical address range. The RVBAR associated with the highest implemented Exception level, RVBAR_EL1, RVBAR_EL2, or RVBAR_EL3, holds this address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The remainder of this section contains the following:

- *PE state on reset to AArch64 state*.
- *Code sequence to use RMR_ELx.RR to request a Warm reset* on page D1-1535.

For more information about reset see:

- *Behavior of caches at reset* on page D3-1709.
- *TLB behavior at reset* on page D4-1835.
- *Reset and debug* on page H6-4935.

### D1.9.1 PE state on reset to AArch64 state

Immediately after a reset, much of the PE state is UNKNOWN. However, some of the PE state is defined. If the PE resets to AArch64 state using either a Cold or a Warm reset, the PE state that is defined is as follows:

- Each of the PSTATE.{D, A, I, F} interrupt masks is set to 1.
- The Software step control bit, PSTATE.SS, is set to 0.
- The IL process state bit, PSTATE.IL, is set to 0.
- All general-purpose, and SIMD and floating-point registers are UNKNOWN.
- The ELR and SPSR for each Exception level are UNKNOWN.
- The stack pointer register for each Exception level is UNKNOWN.
- Unless explicitly defined in this subsection, each System register at each Exception level is in an architecturally UNKNOWN state.
- The TLBs and caches are in an IMPLEMENTATION DEFINED state. This means that the TLBs, the caches, or both, might require invalidation using IMPLEMENTATION DEFINED invalidation sequences before the memory management system or any cache is enabled.

  **Note**

  — The implementation might include IMPLEMENTATION DEFINED resets. If it does, each of these resets might treat the cache and TLB state differently. The ARMv8-A architecture permits this.

  — Different IMPLEMENTATION DEFINED invalidation sequences might be required for different IMPLEMENTATION DEFINED resets.

  — In some implementations, the IMPLEMENTATION DEFINED invalidation sequence might be a NOP.

- In the SCTLR_ELx for the highest implemented Exception level:
  — Each of the {M, C, I} bits is set to 0
  — The EE bit is set to an IMPLEMENTATION DEFINED value, typically defined by a configuration input.
- If an RMR is implemented, RMR_ELx.RR is set to 0. EL*x* in this context is the highest implemented Exception level.
- The enables for the counter event stream are set to 0. This means that the following bits are set to 0:
  — CNTKCTL_EL1.EVNTEN.
  — If the implementation includes EL2, CNTHCTL_EL2.EVNTEN.
- PMCR_EL0.E is set to 0.

  **Note**

  This means the Performance Monitors cannot assert interrupts at reset.

- OSDLR_EL1.DLK bit is set to 0.
- Each of MDCCINT_EL1.{TX, RX} is set to 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- EDPRCR.CWRR is set to 0.
- EDPRSR.SR is set to 1.
- If the implementation includes EL3, then each of MDCR_EL3.{EPMAD, EDAD, SPME} is set to 0.
- If the implementation includes EL2, then MDCR_EL2.HPMN is set to the value of PMCR_EL0.N.
- Each of EDESR.{SS, RC, OSUC} is set to 0.

Additionally, for a Cold reset into AArch64 state:

- If an RMR is implemented, RMR_ELx.AA64 is set to 1. EL*x* in this context is the highest implemented Exception level.
- Each of MDCCSR_EL0.{TXfull, RXfull} is set to 0.
- The DBGPRCR_EL1.CORENPRDRQ is set to the value of EDPRCR.COREPURQ.

  An External Debug reset sets EDPRCR.COREPURQ to 0, see *External debug register resets* on page H8-4963. If an External Debug reset and a Cold reset coincide, both DBGPRCR_EL1.CORENPRDRQ and EDPRCR.COREPURQ are reset to 0.
- The debug CLAIM bits are reset to 0.

  **Note**

  These are the bits that are set to 1 by writing to DBGCLAIMSET_EL1.CLAIM, and cleared to 0 by writing to DBGCLAIMCLR_EL1.CLAIM.
- Each of EDSCR.{RXO, TXU, INTdis, TDA, MA, HDE, ERR, RXfull, TXfull} is set to 0.

  **Note**

  MDCCSR_EL0.{RXfull, TXfull} reflect the values in EDSCR.{RXfull, TXfull}.
- Each of EDECCR.{NSE, SE} is set to 0.
- OSLSR_EL1.OSLK is set to 1.
- In the EDPRSR:
  - The SPMAD, SDAD fields are set to 0.
  - The SPD field is set to 1.

### D1.9.2 Code sequence to use RMR_ELx.RR to request a Warm reset

The following assembler sequence uses RMR_ELx.RR to request a Warm reset:

```
; in addition, interrupts and debug requests for this core should be disabled
; in the system before running this sequence to ensure the WFI suspends execution
    MOV Wy, #3        ; for AArch64, #2 for AArch32; y is any register
    DSB               ; ensure all stores etc are complete
    MSR RMR_ELx, Wy   ; request the reset
    ISB               ; synchronise change to the RMR
Loop
    WFI               ; enter a quiescent state
    B Loop
```

### D1.9.3 Pseudocode description of reset

The `AArch64.TakeReset()` pseudocode function performs a reset into AArch64 state.

`AArch64.TakeReset()` calls the functions `AArch64.ResetGeneralRegisters()`, `AArch64.ResetSIMDFPRegisters()`, `AArch64.ResetSpecialRegisters()`, `AArch64.ResetSystemRegisters()`, and `ResetExternalDebugRegisters()`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`AArch64.ResetSystemRegisters()` resets all System registers to their reset state as defined in the register descriptions in *PE state on reset to AArch64 state* on page D1-1534 and Chapter D7 *AArch64 System Register Descriptions*.

**Note**

The `AArch64.ResetSystemRegisters()` function only resets the System registers.

`ResetExternalDebugRegisters()` resets all external debug registers to their reset state as defined in the register descriptions in Chapter H9 *External Debug Register Descriptions*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.10 Exception entry

Exceptions are targeted at particular Exception levels. The Exception level that an exception targets is either programmed by software, or is determined by the nature of the exception.

Under no circumstances do exceptions cause execution to move to a lower Exception level.

If an asynchronous exception targets a lower Exception level, the exception is not taken and remains pending. See *Asynchronous exception routing* on page D1-1569 and *Asynchronous exception masking* on page D1-1570.

**Note**

The construction of the architecture means that usually, it is impossible for an exception to target a lower Exception level.

The Security state can only change on taking an exception if the exception is taken from Non-secure state to EL3.

**Note**

Taking an exception to EL3 from any Exception level has no effect on the value of the SCR_EL3.NS bit.

On taking an exception to AArch64 state:

- The PE state is saved in the SPSR_ELx at the Exception level the exception is taken to. See *Saved Program Status Registers (SPSRs)* on page D1-1524.
- The preferred return address is saved in the ELR_ELx at the Exception level the exception is taken to. See *Exception Link Registers (ELRs)* on page D1-1527.
- All of PSTATE.{D, A, I, F} are set to 1. See *Process state, PSTATE* on page D1-1529.
- If the exception is a synchronous exception or an SError interrupt, information characterizing the reason for the exception is saved in the ESR_ELx at the Exception level the exception is taken to. See *Use of the ESR_EL1, ESR_EL2, and ESR_EL3* on page D1-1539.
- Execution moves to the target Exception level, and starts at the address defined by the exception vector. Which exception vector is used is also an indicator of whether the exception came from a lower Exception level or the current Exception level. See *Exception vectors* on page D1-1538.
- The stack pointer register selected is the dedicated stack pointer register for the target Exception level. See *The stack pointer registers* on page D1-1523.

The remainder of this section contains the following:

- *Preferred exception return address*.
- *Exception vectors* on page D1-1538.
- *Pseudocode description of exception entry to AArch64 state* on page D1-1539.
- *Exception classes and the ESR_ELx syndrome registers* on page D1-1539.
- *Summary of register updates on faults taken to an Exception level that is using AArch64* on page D1-1551.

### D1.10.1 Preferred exception return address

For an exception taken to an Exception level using AArch64, the Exception Link Register for that Exception level, ELR_ELx, holds the preferred exception return address. The preferred exception return address depends on the nature of the exception, as follows:

- For asynchronous exceptions, it is the address of the instruction following the instruction boundary at which the interrupt occurs. Therefore, it is the address of the first instruction that did not execute, or did not complete execution, as a result of taking the interrupt.
- For synchronous exceptions other than system calls, it is the address of the instruction that generates the exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For system calls, it is the address of the instruction that follows the system call instruction.

**Note**

- If a system call instruction is trapped, disabled, or is UNDEFINED because the Exception level has insufficient privilege to execute the instruction, the preferred exception return address is the address of the system call instruction.
- A system call is generated by the execution of an SVC, HVC, or SMC instruction.

When an exception is taken from an Exception level using AArch32 to an Exception level using AArch64, the top 32 bits of the modified ELR_ELx are 0.

### D1.10.2 Exception vectors

When the PE takes an exception to an Exception level that is using AArch64, execution is forced to an address that is the *exception vector* for the exception. The exception vector exists in a *vector table* at the Exception level the exception is taken to.

A vector table occupies a number of consecutive word-aligned addresses in memory, starting at the *vector base address*.

Each Exception level has an associated *Vector Base Address Register* (VBAR), that defines the exception base address for the table at that Exception level.

For exceptions taken to AArch64 state, the vector table provides the following information:

- Whether the exception is one of the following:
  - Synchronous exception.
  - SError.
  - IRQ.
  - FIQ.
- Information about the Exception level that the exception came from, combined with information about the stack pointer in use, and the state of the register file.

Table D1-7 shows this.

**Table D1-7 Vector offsets from vector table base address**

| Exception taken from | Offset for exception type | | | |
|---|---|---|---|---|
| | Synchronous | IRQ or vIRQ | FIQ or vFIQ | SError or vSError |
| Current Exception level with SP_EL0. | 0x000 | 0x080 | 0x100 | 0x180 |
| Current Exception level with SP_ELx, x>0. | 0x200 | 0x280 | 0x300 | 0x380 |
| Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.[a] | 0x400 | 0x480 | 0x500 | 0x580 |
| Lower Exception level, where the implemented level immediately lower than the target level is using AArch32.[a] | 0x600 | 0x680 | 0x700 | 0x780 |

a. For exceptions taken to EL3, if EL2 is implemented, the level immediately lower than the target level is EL2 if the exception was taken from Non-secure state, but EL1 if the exception was taken from Secure EL1 or EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Reset is treated as a special vector for the highest implemented Exception level. This special vector uses an IMPLEMENTATION DEFINED address that is typically set either by a hardwired configuration of the PE or by configuration input signals. The RVBAR_ELx register contains this reset vector address, where *x* is the number of the highest implemented Exception level.

### D1.10.3 Pseudocode description of exception entry to AArch64 state

The `AArch64.TakeException()` pseudocode function describes the behavior when the PE takes an exception to an Exception level that is using AArch64. The `AArch64.ExceptionClass()` function determines the EC (Exception class) and IL (Instruction length) values required to report the exception, and `AArch64.ReportException()` reports the exception.

The pseudocode functions `AArch64.TakeException()`, `AArch64.ExceptionClass()`, and `AArch64.ReportException()` are described in Chapter J1 *ARMv8 Pseudocode*.

### D1.10.4 Exception classes and the ESR_ELx syndrome registers

If the exception is a synchronous exception or an SError interrupt, information characterizing the reason for the exception is saved in the ESR_ELx at the Exception level the exception is taken to. The information saved is determined at the time the exception is taken, and is not changed as a result of the explicit synchronization that takes place at the start of taking the exception. See *Synchronization requirements for System registers* on page D7-1908. The following sections give more information:

- *Use of the ESR_EL1, ESR_EL2, and ESR_EL3*.
- *EC encodings when routing exceptions to EL2* on page D1-1550.

#### Use of the ESR_EL1, ESR_EL2, and ESR_EL3

An ESR_ELx holds the syndrome information for an exception that is taken to AArch64 state.

**Note**

This use of a syndrome is also the reporting model used for exceptions taken to Hyp mode when they are taken to EL2 using AArch32.

Figure D1-2 shows the general format of the ESR_ELx registers.




**Figure D1-2 Overall format of the ESR_ELx registers**

The ESR_ELx fields are:

**EC, bits[31:26]** The Exception class field, that indicates the cause of the exception.

**IL, bit[25]** The Instruction length bit, for synchronous exceptions, that indicates whether a trapped instruction was a 16-bit or a 32-bit instruction.

**ISS, bits[24:0]** The Instruction specific syndrome field. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

*ESR_ELx, Exception Syndrome Register (ELx)* on page D7-1951 describes the register in full, including:

- Listing the valid EC field values.
- Describing the ISS for each Exception class.
- Giving a full description of the use of the IL field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-8 shows the encoding of the ESR_ELx.EC field, the Exception class field. For each EC value, the table references a subsection of the ESR_ELx register definition that describes the ISS format, with links to descriptions of possible causes of the exception, for example the configuration required to enable a trap.

**Table D1-8 ESR_ELx.EC field encoding**

| EC | Exception class | From, state | | To, Exception level | | | ISS encoding description |
|---|---|---|---|---|---|---|---|
| | | AArch32 | AArch64 | EL1 | EL2 | EL3 | |
| 000000 | Unknown reason | Yes | Yes | Yes | Yes | Yes | *ISS encoding for exceptions with an unknown reason* on page D7-1956 |
| 000001 | WFI or WFE instruction execution[a] | Yes | Yes | Yes | Yes | Yes | *ISS encoding for an exception from a WFI or WFE instruction* on page D7-1958 |
| 000011 | MCR or MRC access to CP15[a] that is not reported using EC 0b000000 | Yes | No | Yes | Yes | Yes[b] | *ISS encoding for an exception from an MCR or MRC access* on page D7-1959 |
| 000100 | MCRR or MRRC access to CP15[a] that is not reported using EC 0b000000 | Yes | No | Yes | Yes | Yes[c] | *ISS encoding for an exception from an MCRR or MRRC access* on page D7-1961 |
| 000101 | MCR or MRC access to CP14[a] | Yes | No | Yes | Yes | Yes | *ISS encoding for an exception from an MCR or MRC access* on page D7-1959 |
| 000110 | LDC or STC access to CP14[a] | Yes | No | Yes | Yes | Yes | *ISS encoding for an exception from an LDC or STC access to CP14* on page D7-1963 |
| 000111 | Access to Advanced SIMD or floating-point registers when CPACR_EL1.FPEN !== 0b11 or CPTR_ELx.TFP == 1, excluding (HCR_EL2.TGE==1) routing | Yes | Yes | Yes | Yes | Yes | *ISS encoding for an exception from an access to an Advanced SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP* on page D7-1965 |
| 001000 | MCR or MRC access to CP10 that is not reported using EC 0b000111. This applies only to ID Group traps[d] | Yes | No | No | Yes | No | *ISS encoding for an exception from an MCR or MRC access* on page D7-1959 |
| 001100 | MRRC access to CP14[a] | Yes | No | Yes | Yes | Yes | *ISS encoding for an exception from an MCRR or MRRC access* on page D7-1961 |
| 001110 | Illegal Execution state | Yes | Yes | Yes | Yes | Yes | *ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault* on page D7-1967 |
| 010001 | SVC instruction execution in AArch32 state | Yes | No | Yes | Yes[e] | No | *ISS encoding for an exception from HVC or SVC instruction execution* on page D7-1967 |
| 010010 | HVC instruction execution in AArch32 state, when HVC is not disabled | Yes | No | No | Yes | No | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D1-8 ESR_ELx.EC field encoding (continued)**

| EC | Exception class | From, state | | To, Exception level | | | ISS encoding description |
|---|---|---|---|---|---|---|---|
| | | AArch32 | AArch64 | EL1 | EL2 | EL3 | |
| 010011 | SMC instruction execution in AArch32 state, when SMC is not disabled | Yes | No | No | Yes[f] | Yes | *ISS encoding for an exception from SMC instruction execution in AArch32 state* on page D7-1968 |
| 010101 | SVC instruction execution in AArch64 state | No | Yes | Yes | Yes | Yes | *ISS encoding for an exception from HVC or SVC instruction execution* on page D7-1967 |
| 010110 | HVC instruction execution in AArch64 state, when HVC is not disabled | No | Yes | No | Yes | Yes | |
| 010111 | SMC instruction execution in AArch64 state, when SMC is not disabled | No | Yes | No | Yes[f] | Yes | *ISS encoding for an exception from SMC instruction execution in AArch64 state* on page D7-1969 |
| 011000 | MSR, MRS, or System instruction execution, that is not reported using EC 0x00, 0x01, or 0x07 | No | Yes | Yes | Yes | Yes | *ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state* on page D7-1970 |
| 011111 | IMPLEMENTATION DEFINED exception to EL3 | Yes | Yes | No | No | Yes | *ISS encoding for a IMPLEMENTATION DEFINED exception to EL3* on page D7-1971 |
| 100000 | Instruction Abort from a lower Exception level[g] | Yes | Yes | Yes | Yes | Yes | *ISS encoding for an exception from an Instruction abort* on page D7-1972 |
| 100001 | Instruction Abort taken without a change in Exception level[g] | Yes | Yes | Yes | Yes | Yes | |
| 100010 | Misaligned PC exception | Yes | Yes | Yes | Yes | Yes | *ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault* on page D7-1967 |
| 100100 | Data Abort from a lower Exception level[h] | Yes | Yes | Yes | Yes | Yes | *ISS encoding for an exception from a Data abort* on page D7-1974 |
| 100101 | Data Abort taken without a change in Exception level[h] | Yes | Yes | Yes | Yes | Yes | |
| 100110 | Stack Pointer Alignment exception | No | Yes | Yes | Yes | Yes | *ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault* on page D7-1967 |
| 101000 | Floating-point exception, if supported, from AArch32 state | Yes | No | Yes | Yes | No | *ISS encoding for an exception from a trapped Floating-point exception* on page D7-1977 |
| 101100 | Floating-point exception, if supported, from AArch64 state | No | Yes | Yes | Yes | Yes | |
| 101111 | SError interrupt | Yes[i] | Yes | Yes | Yes | Yes | *ISS encoding for an SError interrupt* on page D7-1979 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D1-8 ESR_ELx.EC field encoding (continued)**

| EC | Exception class | From, state | | To, Exception level | | | ISS encoding description |
|---|---|---|---|---|---|---|---|
| | | AArch32 | AArch64 | EL1 | EL2 | EL3 | |
| 110000 | Breakpoint exception from a lower Exception level | Yes | Yes | Yes | Yes[j] | No | *ISS encoding for an exception from a Breakpoint or Vector Catch debug event* on page D7-1980 |
| 110001 | Breakpoint exception taken without a change in Exception level | Yes | Yes | Yes | Yes[j] | No | |
| 110010 | Software Step exception from a lower Exception level | Yes | Yes | Yes | Yes[j] | No | *ISS encoding for an exception from a Software Step debug event* on page D7-1980 |
| 110011 | Software Step exception taken without a change in Exception level | Yes | Yes | Yes | Yes[j] | No | |
| 110100 | Watchpoint exception from a lower Exception level | Yes | Yes | Yes | Yes[j] | No | *ISS encoding for an exception from a Watchpoint debug event* on page D7-1981 |
| 110101 | Watchpoint exception taken without a change in Exception level | Yes | Yes | Yes | Yes[j] | No | |
| 111000 | BKPT instruction execution in AArch32 state | Yes | No | Yes | Yes[j] | No | *ISS encoding for an exception from execution of a Software Breakpoint instruction* on page D7-1982 |
| 111010 | Vector catch exception from AArch32 state | Yes | No | No | Yes[j] | No | *ISS encoding for an exception from a Breakpoint or Vector Catch debug event* on page D7-1980 |
| 111100 | BRK instruction execution in AArch64 state | No | Yes | Yes | Yes[j] | Yes[k] | *ISS encoding for an exception from execution of a Software Breakpoint instruction* on page D7-1982 |

a. Exceptions caused by configurable traps, enables, or disables.
b. See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-1602.
c. Only for MCRR or MRRC accesses to the PMCCNTR_EL0 or PMCCNTR.
d. Applies only to traps of accesses to MVFR0, MVFR1, MVFR2, or FPSID. Includes traps of VMRS accesses. Because the MVFR*n* registers are read-only and a VMSR access to the FPSID is ignored and not trapped, there are no MCR or VMSR accesses that can be trapped with this EC value.
e. Only as a result of HCR_EL2.TGE.
f. Only as a result of HCR_EL2.TSC.
g. Used for MMU faults generated by instruction accesses, and for synchronous external aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.
h. Used for MMU faults generated by data accesses, alignment faults other than stack pointer alignment faults, and for synchronous external aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.
i. In AArch32 state, these are known as Asynchronous aborts.
j. Only as a result of HCR_EL2.TGE ==1 or MDCR_EL2.TDE ==1.
k. Only if the BRK instruction is executed in EL3. This is the only debug exception that can be taken to EL3 when EL3 is using AArch64.

#### *Exceptions with an unknown reason*

These are the exceptions reported with an ESR_ELx.EC value of 0b000000

This encoding reports an exception with an unknown reason.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When ESR_ELx.EC returns a value of 0x00, all other fields of ESR_ELx are invalid, and defined as follows:

- IL is set to 1.
- ISS[24:0] is RES0.

An exception with an unknown reason occurs for the following reasons:

- The attempted execution of an instruction bit pattern that has no allocated instruction at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes at the current Exception level and Security state.
  - Instruction encodings for instructions that are not implemented.
- In Debug state, the attempted execution of an instruction bit pattern that is unallocated in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is unallocated in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.
- An exception generated by any of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS to SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction in Non-secure state from EL0 when the value of HCR_EL2.TGE is 1.
  - A DCPS2 instruction from EL1 or EL0 when the value of SCR_EL3.NS is 0, or when EL2 is not implemented.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution of an SRS instruction using R13_mon from Secure EL1. See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-1602.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (Banked register) or an MSR (Banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of 0x07.

#### *Exception from a WFI or WFE instruction, from AArch32 or AArch64 state*

This is the exception syndrome with EC value 0b000001.

This reports exceptions from WFI or WFE instructions executed in either Execution state that result from configurable traps, enables, or disables.

The returned syndrome indicates whether the trapped instruction was a WFI or a WFE. *ISS encoding for an exception from a WFI or WFE instruction* on page D7-1958 describes the format of this syndrome.

The following sections describe configuration settings for generating these exceptions:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-1578.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page D1-1594.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-1602.

#### *Exception from an MCR or MRC access from AArch32 state*

These are the exception syndromes with the following EC values:

- `0b000011`, `MRC` or `MCR` access to CP15.
- `0b000101`, `MRC` or `MCR` access to CP14.
- `0b001000`, `MRC` or `VMRS` access to CP10.

These report exceptions from `MRC`, `MCR`, or `VMRS` instructions executed in AArch32 state that result from configurable traps, enables, or disables and are not reported using the EC code of `0b000000`.

The returned syndrome indicates whether the instruction was an `MRC` or an `MCR`, and the instruction arguments. *ISS encoding for an exception from an MCR or MRC access* on page D7-1959 describes the format of this syndrome.

The following sections describe configuration settings for generating exceptions that are reported using EC value `0b000011`:

- *Traps to EL1 of EL0 accesses to the Generic Timer registers* on page D1-1582.
- *Traps to EL1 of EL0 accesses to Performance Monitors registers* on page D1-1582.
- *Traps to EL2 of Non-secure EL1 accesses to virtual memory control registers* on page D1-1585.
- *Traps to EL2 of Non-secure EL1 execution of TLB maintenance instructions* on page D1-1587.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of cache maintenance instructions* on page D1-1587.
- *Traps to EL2 of Non-secure EL1 accesses to the Auxiliary Control Register* on page D1-1589.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page D1-1589.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to the ID registers* on page D1-1591.
- *Trapping to EL2 of Non-secure EL1 accesses to the CPACR_EL1 or CPACR* on page D1-1595.
- *General trapping to EL2 of Non-secure EL0 and EL1 accesses to System registers, from AArch32 state only* on page D1-1596.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to the Generic Timer registers* on page D1-1600.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page D1-1600.
- *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-1602.
- *Trapping to EL3 of EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR* on page D1-1605.
- *Traps to EL3 of EL2, EL1, and EL0 accesses to Performance Monitors registers* on page D1-1609.

The following sections describe configuration settings for generating exceptions that are reported using EC value `0b000101`:

- *Traps to EL1 of EL0 and EL1 System register accesses to the trace registers* on page D1-1580.
- *Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers* on page D1-1581.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to the ID registers* on page D1-1591, for trapped accesses to the JIDR.
- *Traps to EL2 of Non-secure System register accesses to the trace registers* on page D1-1596.
- *Trapping System register accesses to Debug ROM registers to EL2* on page D1-1598.
- *Trapping System register accesses to powerdown debug registers to EL2* on page D1-1598.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Trapping general System register accesses to debug registers to EL2* on page D1-1599.
- *Traps to EL3 of all System register accesses to the trace registers* on page D1-1606.
- *Trapping System register accesses to powerdown debug registers to EL3* on page D1-1607.
- *Trapping general System register accesses to debug registers to EL3* on page D1-1608.

*Traps to EL2 of Non-secure EL0 and EL1 accesses to the ID registers* on page D1-1591 describes configuration settings for generating exceptions that are reported using EC value `0b001000`.

#### *Exception from an `MCRR` or `MRRC` access from AArch32 state*

These are the exception syndromes with the following EC values:

- `0b000100`, `MRRC` or `MCRR` access to CP15.
- `0b001100`, `MRRC` access to CP14.

These report exceptions from `MCRR` or `MRRC` instructions executed in AArch32 state that result from configurable traps, enables, or disables and are not reported using the EC code of `0x00`.

The returned syndrome indicates whether the instruction was an `MCRR` or an `MRRC`, and the instruction arguments. *ISS encoding for an exception from an MCRR or MRRC access* on page D7-1961 describes the format of this syndrome.

The following sections describe configuration settings for generating exceptions that are reported using EC value `0b000100`:

- *Traps to EL1 of EL0 accesses to the Generic Timer registers* on page D1-1582.
- *Traps to EL1 of EL0 accesses to Performance Monitors registers* on page D1-1582.
- *Traps to EL2 of Non-secure EL1 accesses to virtual memory control registers* on page D1-1585.
- *General trapping to EL2 of Non-secure EL0 and EL1 accesses to System registers, from AArch32 state only* on page D1-1596.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to the Generic Timer registers* on page D1-1600.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page D1-1600.
- *Traps to EL3 of EL2, EL1, and EL0 accesses to Performance Monitors registers* on page D1-1609.

The following sections describe configuration settings for generating exceptions that are reported using EC value `0b001100`:

- *Traps to EL1 of EL0 and EL1 System register accesses to the trace registers* on page D1-1580.
- *Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers* on page D1-1581.
- *Traps to EL2 of Non-secure System register accesses to the trace registers* on page D1-1596.
- *Trapping System register accesses to Debug ROM registers to EL2* on page D1-1598.
- *Traps to EL3 of all System register accesses to the trace registers* on page D1-1606.
- *Trapping System register accesses to powerdown debug registers to EL3* on page D1-1607.

#### *Exception from an `LDC` or `STC` access to CP14 from AArch32 state*

This is the exception syndrome with EC value `0b000110`.

This reports exceptions from `LDC`, or `STC` instructions executed in AArch32 state that result from configurable traps, enables, or disables.

The returned syndrome indicates whether the instruction was an `MCRR` or an `MRRC`, and the instruction arguments. *ISS encoding for an exception from an LDC or STC access to CP14* on page D7-1963 describes the format of this syndrome.

**Note**

The only architected uses of these instructions to access CP14 are:

- An `STC` to write to DBGDTRRX_EL0 or DBGDTRRXint.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- An LDC to read DBGDTRTX_EL0 or DBGDTRTXint.

---

The following sections describe configuration settings for generating exceptions that are reported using EC value 0b000110:

- *Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers* on page D1-1581.
- *Trapping general System register accesses to debug registers to EL2* on page D1-1599
- *Trapping general System register accesses to debug registers to EL3* on page D1-1608.

### Exception from an access to an Advanced SIMD or floating-point register, from either Execution state, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP

This is the exception syndrome with EC value 0b000111.

**Note**

If HCR_EL2.TGE is 1, these exceptions are reported with EC value 0b000000 instead of 0b000111. *EC encodings when routing exceptions to EL2* on page D1-1550 describes this.

---

It reports exceptions from accesses to the Advanced SIMD and floating-point register bank, or to SIMD and floating-point System registers, when CPACR_EL1.FPEN !== 0b11 or CPTR_ELx.TFP == 1.

These are taken from either Execution state.

*ISS encoding for an exception from an access to an Advanced SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP* on page D7-1965 describes the format of the returned syndrome.

The following sections describe configuration settings for generating exceptions that are reported using EC value 0b000111:

- *Traps to EL1 of EL0 and EL1 accesses to SIMD and floating-point functionality* on page D1-1581.
- *General trapping to EL2 of Non-secure accesses to the SIMD and floating-point registers* on page D1-1595
- *Traps to EL3 of all System register accesses to the trace registers* on page D1-1606.

### Exception from an Illegal Execution state, Misaligned PC, or Misaligned stack pointer

These are the exception syndromes with the following EC values:

- 0b001110, Illegal Execution state.
- 0b100010, Misaligned PC.
- 0b100110, Misaligned stack pointer.

When ESR_ELx.EC returns one of these values, the ISS field does not return any syndrome information and the ISS field is RES0.

There are no configuration settings for generating Illegal Execution state exceptions and Misaligned PC exceptions. See *The Illegal Execution state exception* on page D1-1555 and *PC alignment checking* on page D1-1531. *Stack pointer alignment checking* on page D1-1531 describes the configuration settings for generating Misaligned Stack Pointer exceptions.

### Exception from HVC or SVC instruction execution

These are the exception syndromes with the following EC values:

- 0b010001, SVC instruction executed in AArch32 state.
- 0b010010, HVC instruction executed, when not disabled, in AArch32 state.
- 0b010101, SVC instruction executed in AArch64 state.
- 0b010110, HVC instruction executed, when not disabled, in AArch64 state.

The returned syndrome indicates the immediate value given as an instruction argument. *ISS encoding for an exception from HVC or SVC instruction execution* on page D7-1967 describes the format of this syndrome.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not include conditionality information.

See *System calls* on page D1-1610.

### Exception from SMC instruction execution in AArch32 state

This is the exception syndrome with EC value 0b010011.

This reports the exception from an SMC that is not disabled and is executed in AArch32 state.

*ISS encoding for an exception from SMC instruction execution in AArch32 state* on page D7-1968 describes the format of this syndrome.

**Note**

*ISS encoding for an exception from SMC instruction execution in AArch32 state* on page D7-1968 describes ISS[24:0] for each of the following cases:

- When an SMC instruction completes normally and generates an exception that is taken to EL3.
- When an SMC instruction is trapped to EL2 from Non-secure EL1 because HCR_EL2.TSC is set to 1.

*Traps to EL2 of Non-secure EL1 execution of SMC instructions* on page D1-1590 describes the configuration settings for trapping SMC instructions to EL2.

### Exception from SMC instruction execution in AArch64 state

This is the exception syndrome with EC value 0b010111.

This reports the exception from an SMC that is not disabled and is executed in AArch64 state.

The returned syndrome indicates the immediate value given as an instruction argument. *ISS encoding for an exception from SMC instruction execution in AArch64 state* on page D7-1969 describes the format of this syndrome.

**Note**

The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from Non-secure EL1.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

*Traps to EL2 of Non-secure EL1 execution of SMC instructions* on page D1-1590 describes the configuration settings for trapping SMC instructions to EL2.

### Exception from MSR, MRS, or System instruction execution in AArch64 state

This is the exception syndrome with the EC value 0b011000.

These report exceptions from MSR, MRS, or System instructions executed in AArch64 state that result from configurable traps, enables, or disables and are not reported using the EC codes of 0b000000, 0b000001, or 0b000111.

**Note**

*The System instruction class encoding space* on page C5-266 identifies the System instructions referred to in this description.

The returned syndrome indicates whether the instruction was an MSR or an MRS, and the instruction arguments. *ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state* on page D7-1970 describes the format of this syndrome.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For exceptions caused by System instructions, see *System* on page C4-193 for the instruction arguments returned in the syndrome.

The following sections describe configuration settings for generating the exception that is reported using EC value `0b011000`:

- In *EL1 configurable controls* on page D1-1576:
  - *Traps to EL1 of EL0 execution of cache maintenance instructions* on page D1-1577.
  - *Traps to EL1 of EL0 accesses to the CTR_EL0* on page D1-1577.
  - *Traps to EL1 of EL0 execution of DC ZVA instructions* on page D1-1578.
  - *Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks* on page D1-1579.
  - *Traps to EL1 of EL0 and EL1 System register accesses to the trace registers* on page D1-1580.
  - *Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers* on page D1-1581.
  - *Traps to EL1 of EL0 accesses to the Generic Timer registers* on page D1-1582.
  - *Traps to EL1 of EL0 accesses to Performance Monitors registers* on page D1-1582.
- In *EL2 configurable controls* on page D1-1584:
  - *Traps to EL2 of Non-secure EL1 accesses to virtual memory control registers* on page D1-1585.
  - *Traps to EL2 of Non-secure EL0 and EL1 execution of DC ZVA instructions* on page D1-1586.
  - *Traps to EL2 of Non-secure EL1 execution of TLB maintenance instructions* on page D1-1587.
  - *Traps to EL2 of Non-secure EL0 and EL1 execution of cache maintenance instructions* on page D1-1587.
  - *Traps to EL2 of Non-secure EL1 accesses to the Auxiliary Control Register* on page D1-1589.
  - *Traps to EL2 of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page D1-1589.
  - *Traps to EL2 of Non-secure EL0 and EL1 accesses to the ID registers* on page D1-1591.
  - *Trapping to EL2 of Non-secure EL1 accesses to the CPACR_EL1 or CPACR* on page D1-1595.
  - *Traps to EL2 of Non-secure System register accesses to the trace registers* on page D1-1596.
  - *Trapping System register accesses to Debug ROM registers to EL2* on page D1-1598.
  - *Trapping System register accesses to powerdown debug registers to EL2* on page D1-1598.
  - *Trapping general System register accesses to debug registers to EL2* on page D1-1599.
  - *Traps to EL2 of Non-secure EL0 and EL1 accesses to the Generic Timer registers* on page D1-1600.
  - *Traps to EL2 of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page D1-1600.
- In *EL3 configurable controls* on page D1-1601:
  - *Traps to EL3 of Secure EL1 accesses to the Counter-timer Physical Secure timer registers* on page D1-1604.
  - *Trapping to EL3 of EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR* on page D1-1605.
  - *Traps to EL3 of all System register accesses to the trace registers* on page D1-1606.
  - *Trapping System register accesses to powerdown debug registers to EL3* on page D1-1607.
  - *Trapping general System register accesses to debug registers to EL3* on page D1-1608.
  - *Traps to EL3 of EL2, EL1, and EL0 accesses to Performance Monitors registers* on page D1-1609.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Exception from an Instruction abort

These are the exception syndromes with the following EC values:

- `0100000`, for an Instruction abort exception taken from a lower Exception level, that could be using AArch64 or AArch32.
- `0b100001`, for an Instruction abort exception taken without a change in Exception level, meaning it is taken from an Exception level that is using AArch64.

These EC values are used for MMU faults and synchronous external aborts, including synchronous parity or ECC errors, that are generated by instruction accesses. They are not used for Debug exceptions.

The returned syndrome provides more information about the exception, including a fault code that indicates the cause of the exception. *ISS encoding for an exception from an Instruction abort* on page D7-1972 describes the format of this syndrome.

### Exception from a Data abort

These are the exception syndromes with the following EC values:

- `0b100100`, for a Data abort exception taken from a lower Exception level, that could be using AArch64 or AArch32.
- `0b100101`, for a Data abort exception taken without a change in Exception level, meaning it is taken from an Exception level that is using AArch64.

These EC values are used for the following exceptions if the exception is generated by a data access:

- MMU faults.
- Alignment faults other than those caused by stack pointer misalignment.
- Synchronous external aborts, including synchronous parity or ECC errors.

They are not used for Debug exceptions.

The returned syndrome provides more information about the exception, including a fault code that indicates the cause of the exception. *ISS encoding for an exception from a Data abort* on page D7-1974 describes the format of this syndrome.

### Floating-point exceptions

These are the exception syndromes with the following EC values:

- `0b101000`, trapped floating-point exception from AArch32.
- `0b101100`, trapped floating-point exception from AArch64.

These Exception classes are supported only when the SIMD and floating-point implementation supports the trapping of floating-point exceptions. Otherwise, the `0x28` and `0x2C` EC values are reserved. That is, these EC values are used to report the floating-point exceptions defined by IEEE 754, and input denormal.

The returned syndrome identifies the trapped floating-point exception or exceptions. *ISS encoding for an exception from a trapped Floating-point exception* on page D7-1977 describes the format of this syndrome.

In an implementation where the SIMD and floating-point implementation supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

### SError interrupt

This is the exception syndrome with EC value `0b101111`.

It is used to report the exception caused by an SError interrupt.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The returned syndrome is implementation specific. *ISS encoding for an SError interrupt* on page D7-1979 describes the format of this syndrome. See also *Asynchronous exception types, routing, masking and priorities* on page D1-1569.

#### Breakpoint exception or Vector Catch exception

These are the exception syndromes with the following EC values:

- `0b110000`, Breakpoint exception taken from a lower Exception level.
- `0b110001`, Breakpoint exception taken without a change of Exception level.
- `0b111010`, AArch32 Vector Catch exception.

The returned syndrome provides a fault code that indicates the cause of the exception. *ISS encoding for an exception from a Breakpoint or Vector Catch debug event* on page D7-1980 describes the format of this syndrome.

For more information about generating these exceptions, see:

- *Breakpoint exceptions* on page D2-1653.
- *Vector Catch exceptions* on page D2-1685.

#### Watchpoint exception

These are the exception syndromes with the following EC values:

- `0b110100`, Watchpoint exception taken from a lower Exception level.
- `0b110101`, Watchpoint exception taken without a change of Exception level.

The returned syndrome provides more information about the watchpoint, including a fault code that indicates the cause of the exception. *ISS encoding for an exception from a Watchpoint debug event* on page D7-1981 describes the format of this syndrome.

For more information about generating these exceptions, see *Watchpoint exceptions* on page D2-1671.

#### Software Step exception

These are the exception syndromes with the following EC values:

- `0b110010`, Software Step exception taken from a lower Exception level.
- `0b110011`, Software Step exception taken without a change of Exception level.

The returned syndrome provides more information about the watchpoint, including a fault code that indicates the cause of the exception. *ISS encoding for an exception from a Software Step debug event* on page D7-1980 describes the format of this syndrome.

For more information about generating these exceptions, see *Software Step exceptions* on page D2-1686.

#### Software Breakpoint Instruction exception

These are the exception syndromes with the following EC values:

- `0b111000`, `BKPT` instruction executed in AArch32 state.
- `0b111100`, `BRK` instruction executed in AArch64 state.

The returned syndrome provides the comment that was provided as an argument to the Software Breakpoint Instruction. *ISS encoding for an exception from execution of a Software Breakpoint instruction* on page D7-1982 describes the format of this syndrome.

For more information about generating these exceptions, see *Software Breakpoint Instruction exceptions* on page D2-1650.

### EC encodings when routing exceptions to EL2

When an exception is taken to EL2 because the exception routing control HCR_EL2.TGE is enabled, the EC encoding that would have been used if the exception had been taken to EL1 is recorded in ESR_EL2.EC instead, unless the encoding is `0x07`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Exceptions that use 0x07 when the HCR_EL2.TGE routing control is disabled use 0x00 when the HCR_EL2.TGE routing control is enabled.

### D1.10.5 Summary of register updates on faults taken to an Exception level that is using AArch64

For all exceptions taken to an Exception level using AArch64 that are not listed in *Validity of FAR_ELx*, the FAR_ELx for the Exception level the exception is taken to is UNKNOWN.

For all exceptions taken to EL2 using AArch64 that are not listed in *Validity of HPFAR_EL2*, the HPFAR_EL2 is UNKNOWN.

#### Validity of FAR_ELx

The faulting virtual address is saved in FAR_ELx for the Exception level the exception is taken to if an exception is one of:

- An Instruction Abort exception.
- A Data Abort exception.
- A Misaligned PC exception.
- A Watchpoint exception.

The architecture permits that the FAR_ELx is UNKNOWN for Synchronous External Aborts other than Synchronous External Aborts on Translation Table Walks. In this case, the ISS.FnV bit returned in ESR_ELx indicates whether FAR_ELx is valid.

If an exception is taken from an Exception level using AArch32 into an Exception level using AArch64, and that exception writes the FAR_ELx at the Exception level the exception is taken to, the most significant 32 bits of the FAR_ELx are all zero, unless both:

- The faulting address was generated by a load or store that sequentially incremented from address 0xFFFFFFFF. Such a load or store instruction is UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address.

The FAR_ELx for an Exception level is made UNKNOWN as a result of an exception return from that Exception level.

#### Validity of HPFAR_EL2

The faulting IPA is saved in HPFAR_EL2 if the exception is an Instruction Abort or Data Abort taken to EL2 and the fault is one of:

- A Translation or Access Flag fault on a stage 2 translation.
- A stage 2 Address Size fault.
- A fault on the stage 2 translation of an address accessed in a stage 1 translation table walk.

HPFAR_EL2 is made UNKNOWN as a result of an exception return from EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.11 Exception return

In the ARMv8-A architecture, an exception return is always to the same Exception level or a lower Exception level. An exception return is used for:

- A return to a previously executing thread.
- Entry to a new execution thread. For example:
  - The initialization of a hypervisor by a Secure monitor.
  - The initialization of an operating system by a hypervisor.
  - Application entry from an operating system or hypervisor.

An exception return requires the simultaneous restoration of the PC and PSTATE to values that are consistent with the desired state of execution on returning from the exception.

In AArch64 state, an ERET instruction causes an exception return. On an ERET instruction:

- The PC is restored with the value held in the ELR_ELx.
- PSTATE is restored by using the contents of the SPSR_ELx.

The ELR_ELx and SPSR_ELx are the ELR_ELx and SPSR_ELx at the Exception level the exception is returning from. The exception return makes this ELR_ELx and SPSR_ELx UNKNOWN.

See *Address tagging in AArch64 state* on page D4-1742 for details of how tagged addresses are handled in an Exception return from an Exception level using AArch64 to an Exception level using AArch64.

**Note**

When returning from an Exception level using AArch64 to an Exception level using AArch32, the top 32 bits of the ELR_ELx are ignored.

An ERET instruction also:

- Sets the Event Register for the PE executing the ERET instruction. See *Mechanisms for entering a low-power state* on page D1-1611.
- Resets the local exclusive monitor for the PE executing the ERET instruction. This removes the risk of errors that might be caused when a path to an exception return fails to include a CLREX instruction.

  **Note**

  This behavior prevents self-hosted debug from software stepping through an LDREX/STREX pair. However, when self-hosted debug is using software step, it is highly probable that the exclusive monitor state would be lost anyway, for other reasons. *Stepping code that uses exclusive monitors* on page D2-1698 describes this.

  It is IMPLEMENTATION DEFINED whether the resetting of the local exclusive monitor also resets the global exclusive monitor.

The ERET instruction is UNDEFINED in EL0.

When returning from an Exception level using AArch64 to an Exception level using AArch32, the AArch32 context is restored. The ARMv8-A architecture defines the relationship between AArch64 state and AArch32 state, for:

- General-purpose registers.
- Special-purpose registers.
- System registers.

In an implementation that includes EL3, the Security state can only change on returning from an exception if the return is from EL3 to a lower Exception level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following sections give more information:

- *Exception return and PC alignment*.
- *Illegal return events from AArch64 state*.
- *Legal returns that set PSTATE.IL to 1* on page D1-1555.
- *The Illegal Execution state exception* on page D1-1555.
- *Pseudocode description of exception return* on page D1-1555.

### D1.11.1 Exception return and PC alignment

When SPSR_ELx.M[4] == 0, indicating an Exception return to AArch64 state, the value of ELR_ELx is transferred to the PC. If this value is misaligned, subsequent execution results in a Misaligned PC exception.

When SPSR_ELx.M[4] == 1, indicating an Exception return to AArch32 state, the value of ELR_ELx is transferred to the PC except that, for a legal exception return:

- If SPSR_ELx.T is 0, ELR_ELx[1:0] are treated as being 0 for restoring the PC.
- If SPSR_ELx.T is 1, ELR_ELx[0] is treated as being 0 for restoring the PC.

This means that a Misaligned PC exception cannot occur following a legal exception return from AArch64 state to AArch32 state. However, where the Exception return with SPSR_ELx.M[4] == 1 is an illegal exception return then it is IMPLEMENTATION DEFINED whether a misaligned value in ELR_ELx is aligned when it is restored to the PC.

**Note**

In an implementation that forces the alignment of the PC value restored from SPSR_ELx on an illegal exception return with SPSR_ELx.M[4] == 1, if SPSR_ELx.T == 1 the restored PC value might give rise to a Misaligned PC exception, because the PE remains in AArch64 state and only ELR_ELx[0] is treated as being 0 for restoring the PC.

For more information about the illegal exception return cases see *Illegal return events from AArch64 state*.

### D1.11.2 Illegal return events from AArch64 state

In this section:

**Return** In AArch64 state, refers to any of:

- Execution of an `ERET` instruction.
- Execution of a `DRPS` instruction in Debug state.
- Exit from Debug state.

**Saved process state value**

In AArch64 state, refers to any of:

- The value held in the SPSR_ELx for an `ERET` instruction.
- The value held in the SPSR_ELx for a `DRPS` instruction executed in Debug state.
- The value held in the DSPSR_EL0 for a Debug state exit.

**Link address** In AArch64 state, refers to any of:

- The address held in ELR_ELx for an ERET instruction.
- The address held in DLR_EL0 for a Debug state exit.

**Configured from reset**

Indicates the state determined on powerup or reset by a configuration input signal, or by another IMPLEMENTATION DEFINED mechanism.

The ARMv8 architecture has a generic mechanism for handling returns to a mode or state that is illegal. In AArch64 state, this can occur as the result of any of the following situations:

- A return where the Exception level being returned to is higher than the current Exception level.
- A return where the Exception level being returned to is not implemented. For example a return to EL2 when EL2 is not implemented.
- A return to EL2 when EL3 is implemented and the value of the SCR_EL3.NS bit is 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A return to Non-secure EL1 when EL2 is implemented and the value of the HCR_EL2.TGE bit is 1.
- A return where the value of the saved process state M[4] bit is 0, indicating a return to AArch64 state, and one of the following is true:
  - The M[1] bit is 1.
  - The M[3:0] bits are `0b0001`.
  - The Exception level being returned to is using AArch32 state, as programmed by the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
- A return where the value of the saved process state M[4] bit is 1, indicating a return to AArch32 state, and one of the following is true:
  - The M field value is not a valid AArch32 state PE mode. Table D1-4 on page D1-1527 shows the valid M[3:0] values for AArch32 state PE modes. This includes the case where M[3:0] is `0b0000`, indicating User mode, and EL0 does not support AArch32 state.
  - The Exception level being returned to is using AArch64 state as determined by the SCR_EL3.RW or HCR_EL2.RW field or the configuration from reset. This includes the case where the Exception level being returned to does not support AArch32 state.

  **Note**

  This means that, in an implementation that supports only AArch64 state, any attempt to return to AArch32 state is an illegal exception return.
- A Debug state exit from EL0 using AArch64 state, to EL0 using AArch32 state.

In these cases:

- PSTATE.IL is set to 1, to indicate an illegal return.
- PSTATE.{EL, nRW, SP} are unchanged. This means the Exception level, Execution state, and stack pointer selection do not change as a result of the return.
- The following PSTATE bits are restored from the saved process state value:
  - The N, Z, C, V Condition flags.
  - The D, A, I, F exception mask bits.
- If the illegal return is an illegal exception return, the PSTATE.SS bit is handled as normal for a return. That is, the SS bit is handled in the same way as an exception return that is not an illegal exception return. See *Software Step exceptions* on page D2-1686.

  In all these cases the PSTATE.SS bit is handled as it would be for a normal return, as described in *Entering the active-not-pending state* on page D2-1688 and *Exiting Debug state* on page H2-4858. DRPS never sets the SS bit. This is indicated in *Entering the active-not-pending state* on page D2-1688.
- If the illegal return is not a DRPS instruction executed in Debug state, the PC is restored from the link address. However, if the value of the M[4] bit of the saved process state is 1, indicating a return to AArch32 state, then:
  - It is IMPLEMENTATION DEFINED whether the PC value is aligned by setting the bottom 1 or 2 bits of its value to 0, as determined by the T bit of the saved process state. See *Exception return and PC alignment* on page D1-1553.
  - It is CONSTRAINED UNPREDICTABLE whether bits[63:32] of the PC are all set to zero, or are set to the value of the corresponding bits of the link address.

  The implementation determines the choice of these two options, and the choice might vary dynamically. Therefore, software must tolerate both of these options.

*Relaxation of the tagged address handling requirements on an Illegal exception return* on page D4-1744 describes how tagged addresses are handled in an Illegal Exception return from an Exception level using AArch64 to an Exception level using AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the value of the PSTATE.IL bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception*.

All aspects of the illegal return, other than the effects described in this section, occur as they do for a legal return.

### D1.11.3 Legal returns that set PSTATE.IL to 1

In this section, *return*, *saved process state value*, and *link address* have the same meaning as defined in *Illegal return events from AArch64 state* on page D1-1553.

If the value of the IL bit in the saved process state is 1, then it is copied to PSTATE by a return, meaning that PSTATE.IL is set to 1. In this case, if the return is not an illegal return, and targets AArch32 state, then the PSTATE.{IT, T} bits are either:

- Set to 0.
- Copied from the saved process state value.

The choice between these two options is determined by an implementation, and might vary dynamically within the implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

The PSTATE.{IT, T} bits are only valid in AArch32 state, see *Process state, PSTATE* on page G1-3767.

When the PSTATE.IL bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception*.

### D1.11.4 The Illegal Execution state exception

When the value of the PSTATE.IL bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. In AArch64 state, the PSTATE.IL bit can be to set to 1 by any of:

- An illegal return, as described in *Illegal return events from AArch64 state* on page D1-1553.
- A legal return that sets PSTATE.IL to 1, as described in *Legal returns that set PSTATE.IL to 1*.

An Illegal Execution state exception sets ESR_ELx.EC for the target Exception level to the value of `0x0E`.

On taking any exception to an Exception level that is using AArch64 state:

1. The value of the PSTATE.IL bit is copied into the SPSR_ELx.IL bit for the Exception level to which the exception is taken.
2. The PSTATE.IL bit is cleared to 0.

**Note**

This means that it is not possible for software to observe the value of PSTATE.IL.

For the priority of this exception class, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

### D1.11.5 Pseudocode description of exception return

The `AArch64.ExceptionReturn()` pseudocode function transfers the return address to the PC, and restores PSTATE to its saved value by calling `SetPSTATEFromPSR()`.

The `IllegalExceptionReturn()` function checks for an Illegal Execution state exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.12 The Exception level hierarchy

The System registers provide controls that control PE behavior through the Exception level hierarchy.

If EL3 and EL2 are implemented, System registers at EL3 and EL2 provide controls that control the Execution state of lower Exception levels.

Table D1-9 shows the principal System registers.

**Table D1-9 Principal System registers**

| EL3 | EL2 | EL1 | Notes |
|---|---|---|---|
| SCTLR_EL3 | SCTLR_EL2 | SCTLR_EL1 | Controls execution for its own Exception level. |
| SCR_EL3 | HCR_EL2 | - | Controls execution at lower Exception levels. |
| - | HSTR_EL2 | - | Used only if at least one of EL1 and EL0 is using AArch32. |

The prioritization of exceptions generated as a result of controls in these registers is described in *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

The following sections describe the Exception level hierarchy:

- *The hierarchy of configuration and routing control*.
- *Control of SIMD, floating-point and trace functionality* on page D1-1561.
- *Control of IMPLEMENTATION DEFINED features* on page D1-1562.
- *Routing exceptions to EL2* on page D1-1564.

### D1.12.1 The hierarchy of configuration and routing control

The following subsections give a summary of the controls available at each Exception level for controlling execution at that Exception level and all lower Exception levels:

- *Controls provided at EL3*.
- *Controls provided at EL2* on page D1-1558.
- *Controls provided at EL1* on page D1-1560.

For information on how the controls summarized in these subsections affect PE behavior, see the definitions of the control bits in the register descriptions.

#### Controls provided at EL3

See:

- *Controls provided by the SCR_EL3*.
- *Controls provided by the SCTLR_EL3* on page D1-1557.
- *Controls provided by the MDCR_EL3* on page D1-1557.

##### *Controls provided by the SCR_EL3*

**SCR_EL3.NS** Determines the Security state of execution at EL1 and EL0.

**SCR_EL3.RW** Determines the Execution state of the next-lower Exception level.

**SCR_EL3.{EA, FIQ, IRQ}**

Route:

**EA** Physical SError interrupts and synchronous External Aborts to EL3.

**FIQ** Physical FIQ interrupts to EL3.

**IRQ** Physical IRQ interrupts to EL3.

**SCR_EL3.SMD** Disables the Secure Monitor Call exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SCR_EL3.HCE** Enables the Hypervisor Call exception.

**SCR_EL3.ST** Enables Secure EL1 access to the Secure timer.

**SCR_EL3.SIF** Secure Instruction Fetch. When in Secure state, disables instruction fetches from Non-secure memory.

**SCR_EL3.TWI** Trap Wait-For-Interrupt.

**SCR_EL3.TWE** Trap Wait-For-Event.

### *Controls provided by the SCTLR_EL3*

**SCTLR_EL3.{A, SA}** Enable alignment checking:

- **A** On data accesses from EL3.
- **SA** On the SP, when executing at EL3.

**SCTLR_EL3.{M, C, I, WXN}**

Memory system control bits:

- **M** Enables EL3 stage 1 address translation.
- **C** When set to 0, makes data accesses to Normal memory from EL3, and Normal memory accesses to the EL3 translation tables, Non-cacheable.
- **I** When set to 0, makes instruction accesses to Normal memory from EL3 Non-cacheable.
- **WXN** For accesses from EL3, enables treating all writable memory regions as XN, execute never.

**SCTLR_EL3.EE** Defines the endianness of data accesses from EL3, including stage 1 translation table walks at EL3.

**Note**

Instruction fetches are always little-endian.

### *Controls provided by the MDCR_EL3*

**MDCR_EL3.{EPMAD, EDAD}**

Enable external debugger accesses to:

- **EPMAD** Performance Monitors registers.
- **EDAD** Breakpoint and Watchpoint registers.

**MDCR_EL3.{SPME, SDD, SPD32}**

Secure debug controls:

- **SPME** Secure Performance Monitors enable. Enables event counting in Secure state.
- **SDD** Disables all debug exceptions taken from Secure state, if the *debug target Exception level*, $EL_D$, is using AArch64.
- **SPD32** Enables debug exceptions from Secure EL1 using AArch32.

**MDCR_EL3.{TDOSA, TDA, TPM}**

These are trap controls, that control traps to EL3 of EL2, EL1, and EL0 accesses to the following:

- **TDOSA** The OS-related debug registers.
- **TDA** Those debug registers not included in the MDCR_EL3.TDOSA trap.
- **TPM** The Performance Monitors registers.

For EL1 and EL0, these traps apply to accesses from both Security states.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Controls provided at EL2

EL2 is implemented only in Non-secure state, and its controls apply only to execution in Non-secure EL2, Non-secure EL1, and Non-secure EL0. See:

- *Controls provided by SCTLR_EL2*.
- *Controls provided by HCR_EL2*.
- *Controls provided by the HSTR_EL2* on page D1-1559.
- *Controls provided by the MDCR_EL2* on page D1-1560.

### *Controls provided by SCTLR_EL2*

**SCTLR_EL2.{A, SA}** Enable alignment checking:

- **A** On data accesses from EL2.
- **SA** On the SP, when executing at EL2.

**SCTLR_EL2,{M, C, I, WXN}**

Memory system control bits:

- **M** Enables EL2 stage 1 address translation.
- **C** When set to 0, makes data accesses to Normal memory from EL2, and Normal memory accesses to the EL2 translation tables, Non-cacheable.
- **I** When set to 0, makes instruction accesses to Normal memory from EL2 Non-cacheable.
- **WXN** For accesses from EL2, enables treating all writable memory regions as XN, execute never.

**SCTLR_EL2.EE** Defines the endianness of data accesses from EL2, including stage 1 translation table walks at EL2.

Also defines the endianness of stage 2 translation table walks at Non-secure EL1 and EL0.

**Note**

Instruction fetches are always little-endian.

### *Controls provided by HCR_EL2*

**HCR_EL2.RW** Determines the Execution state of the next-lower Exception level.

**HCR_EL2.{AMO, IMO, FMO}**

Route physical interrupts to EL2 and enable virtual interrupts:

- **AMO** Route physical SError interrupts to EL2 and enable virtual SError interrupts
- **IMO** Route physical IRQ interrupts to EL2 and enable virtual IRQ interrupts.
- **FMO** Route physical FIQ interrupts to EL2 and enable virtual FIQ interrupts.

**Note**

If a physical interrupt is routed to both EL3 and EL2, routing to EL3 takes precedence over routing to EL2.

**HCR_EL2.{VSE, VI, VF}**

Cause a virtual interrupt to be pending:

- **VSE** Virtual SError interrupt.
- **VI** Virtual IRQ interrupt.
- **VF** Virtual FIQ interrupt.

**HCR_EL2.VM** Enable bit for Non-secure EL1&0 stage 2 address translations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**HCR_EL2.{SWIO, PTW, FB, BSU, DC, CD, ID}**

Controls for memory system behavior for accesses made from Non-secure EL1 and EL0:

**SWIO** Set/Way Invalidate Override.

**PTW** Protect Table Walk.

**FB** Force broadcast of TLB and instruction cache maintenance instructions.

**BSU** Barrier Shareability Upgrade.

**DC** Default Cacheability control for the EL1&0 translation regime.

**CD** For the Non-secure EL1&0 translation regime, when set to 1, forces stage 2 translation table walks to Normal memory, and stage 2 translations for data accesses to Normal memory, to be Non-cacheable.

**ID** For the Non-secure EL1&0 translation regime, when set to 1, forces stage 2 translations for instruction accesses to Normal memory be Non-cacheable.

**HCR_EL2.HCD** Hypervisor Call Disable.

**Note**

If an implementation includes EL3, this bit is RES0.

**HCR_EL2.{TRVM, TDZ, TVM, TTLB, TPU, TPC, TSW, TACR, TIDCP, TSC, TID1, TID2, TID3, TWE, TWI}**

Trap operations performed at Non-secure EL1 or EL0 to EL2, as follows:

**TRVM** Trap Read of Virtual Memory controls.

**TDZ** Trap Data Cache Zero.

**TVM** Trap Virtual Memory controls.

**TTLB** Trap TLB maintenance instructions.

**TPU** Trap cache maintenance to the Point of Unification instructions.

**TPC** Trap data cache maintenance to the Point of Coherency instructions.

**TSW** Trap data cache maintenance by Set/Way instructions.

**TACR** Trap Auxiliary Control Register accesses.

**TIDCP** Trap Implementation-Dependent functionality.

**TSC** Trap Secure Monitor Call.

**TID0** Trap ID Group 0 register accesses.

**TID1** Trap ID Group 1 register accesses.

**TID2** Trap ID Group 2 register accesses.

**TID3** Trap ID Group 3 register accesses.

**TWI** Trap Wait-For-Interrupt.

**TWE** Trap Wait-For-Event.

**Note**

There are no AArch64 System registers in ID Group 0, therefore the TID0 trap is only relevant when Non-secure EL1 is using AArch32.

**HCR_EL2.TGE** Trap General Exceptions.

### *Controls provided by the HSTR_EL2*

When EL2 is using AArch64 and at least one of Non-secure EL1 or EL0 is using AArch32, HSTR_EL2 provides the following trap of Non-secure AArch32 operation to EL2:

**HSTR_EL2.T*n*, for values of *n* in the set {0-3, 5-13, 15}**

Trap accesses to System registers in the AArch32 conceptual coprocessor CP15, by the coprocessor primary register number.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Controls provided by the MDCR_EL2*

**MDCR_EL2.{TDRA, TDOSA, TDA}**

Trap controls, that control traps to EL2 of Non-secure EL1 and EL0 System register accesses to the following:

**TDRA** The Debug ROM registers.

**TDOSA** The OS-related debug registers.

**TDA** Those debug registers not included in either of the MDCR_EL2.TDRA or MDCR_EL2.TDOSA traps.

**MDCR_EL2.TDE** Routes all debug exceptions taken from Non-secure EL1 and EL0 to EL2.

**MDCR_EL2.{TPM, TPMCR}**

These are trap controls, that control traps to EL2 of Non-secure EL1 and EL0 accesses to the following registers:

**TPM** All Performance Monitors registers.

**TPMCR** The Performance Monitors Control Registers.

**MDCR_EL2.HPMN** Defines the number of Performance Monitors counters that are accessible from Non-secure EL1 and EL0.

### Controls provided at EL1

See:

- *Controls provided by the SCTLR_EL1*.
- *Controls provided by the MDSCR_EL1* on page D1-1561.

#### *Controls provided by the SCTLR_EL1*

**SCTLR_EL1.{A, SA}** Enable alignment checking:

**A** On data accesses from EL1 and EL0.

**SA** On the SP, when executing at EL1.

**SCTLR_EL1.SA0** Enable alignment checking on the SP when executing at EL0.

**SCTLR_EL1.{M, C, I, WXN}**

Memory system control bits:

**M** Enables EL1&0 stage 1 address translation.

**C** When set to 0, makes data accesses to Normal memory from EL0 and EL1, and Normal memory accesses to the EL1&0 stage 1 translation tables, Non-cacheable.

**I** When set to 0, makes instruction accesses to Normal memory from EL1 and EL0 Non-cacheable.

**WXN** For accesses from EL1 and EL0, enables treating all writable memory regions as XN, execute never.

**SCTLR_EL1.EE** Defines the endianness of data accesses from EL1, including stage 1 translation table walks at EL1 and EL0.

**Note**

Instruction fetches are always little-endian.

**SCTLR_EL1.E0E** EL0 Endianness. Defines the endianness used for explicit data accesses made from EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SCTLR_EL1.{UCI, UCT, DZE, nTWI, nTWE}**

Trap enables:

**UCI** Unprivileged Cache maintenance Instruction enable.

**UCT** Unprivileged Cache Type access enable.

**DZE** Data cache Zero Enable.

**nTWI** Not Trap Wait-For-Interrupt.

**nTWE** Not Trap Wait-For-Event.

**SCTLR_EL1.UMA** Unprivileged Mask Access.

**SCTLR_EL1.{SED, ITD, CP15BEN}**

These bits control the use, at EL0, of AArch32 functionality that is deprecated, or OPTIONAL and deprecated:

**SED** Disables use of the `SETEND` instruction.

**ITD** If supported, disables some uses of the `IT` instruction. The register field description identifies the uses that are disabled.

**CP15BEN** If supported, enables use of the CP15DMB, CP15DSB, and CP15ISB memory barrier instructions.

The deprecated uses of the `IT` instruction, and use of the CP15DMB, CP15DSB, and CP15ISB instructions, are deprecated for performance reasons. Implementation of the ITD and CP15BEN controls is optional, and if a control is not implemented then the associated AArch32 functionality cannot be disabled.

***Controls provided by the MDSCR_EL1***

**MDSCR_EL1.{MDE, SS}**

Enable controls for the debug exceptions:

**MDE** Enables Breakpoint exceptions, Watchpoint exceptions, and Vector Catch exceptions.

**SS** Enables Software Step exceptions.

There is no enable control for Software Breakpoint Instruction exceptions. Software Breakpoint Instruction exceptions are always enabled.

**MDSCR_EL1.KDE** Enables debug exceptions from $EL_D$ when $EL_D$ is using AArch64.

**MDSCR_EL1.TDCC** Enables a trap to EL1 of EL0 accesses to the Debug Communications Channel registers.

## D1.12.2 Control of SIMD, floating-point and trace functionality

In addition to the controls described in *The hierarchy of configuration and routing control* on page D1-1556, the following registers provide a hierarchy of control of access to SIMD and floating-point functionality, and to trace functionality that is accessible using the System registers:

**CPTR_EL3** Traps operation at lower Exception levels to EL3, if the operation is not trapped to EL2 by CPTR_EL2 and is not trapped to EL1 by CPACR_EL1.

**CPTR_EL2** Traps operation in Non-secure EL1 or EL0 to EL2, if the operation is not trapped to EL1 by CPACR_EL1. CPTR_EL2.{TTA, TFP} also trap operation in EL2.

The trap bits in the CPTR_EL3 and CPTR_EL2 are as follows:

**TCPAC** Traps accesses to the registers that control access to SIMD, floating-point, and trace functionality.

**TTA** Traps any System register access to trace functionality, unless that access is otherwise trapped to a lower Exception level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TFP** Traps any execution of an instruction that uses the SIMD and floating-point register bank, unless that access is otherwise trapped to a lower Exception level.

**CPACR_EL1** Traps operation from EL1 or EL0 to EL1. Traps set in the CPACR_EL1 take precedence over any traps set in the CPTR_EL2 or CPTR_EL3. The trap fields are as follows:

**TTA** Traps to EL1 any System register access from EL0 or EL1 to trace functionality.

**FPEN** Traps to EL1 execution of instructions that uses the SIMD and floating-point register bank.

### D1.12.3 Control of IMPLEMENTATION DEFINED features

*The hierarchy of configuration and routing control* on page D1-1556 and *Control of SIMD, floating-point and trace functionality* on page D1-1561 describe the controls of the trapping of architecturally-defined functionality. However, the architecture also defines registers that can be used to provide IMPLEMENTATION DEFINED traps of IMPLEMENTATION DEFINED functionality to the different Exception levels. Table D1-10 shows these control registers, for AArch64 state controls.

**Table D1-10 Control of traps of IMPLEMENTATION DEFINED functionality**

| Traps to EL3 | Traps to EL2 | Traps to EL1 | Notes |
|---|---|---|---|
| ACTLR_EL3 | ACTLR_EL2 | ACTLR_EL1 | Registers also provide IMPLEMENTATION DEFINED configuration controls for the appropriate Exception level. |
| - | HACR_EL2 | - | Provides traps of IMPLEMENTATION DEFINED Non-secure EL1 and EL0 functionality to EL2. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.13 Synchronous exception types, routing and priorities

Synchronous exceptions are:

- UNDEFINED exceptions generated by:
  - Attempts to execute instructions at an inappropriate Exception level.
  - Attempts to execute instruction bit patterns that have not been allocated.
- Illegal Execution state exceptions. These are caused by attempts to execute an instruction when the value of PSTATE.IL is 1, see *Illegal return events from AArch64 state* on page D1-1553.
- Exceptions caused by the use of a misaligned Stack Pointer.
- Exceptions caused by attempting to execute an instruction with a misaligned PC.
- Exceptions caused by the exception-generating instructions `SVC`, `HVC`, or `SMC`.
- Traps on attempts to execute instructions that the System registers define as instructions that are trapped to a higher Exception level. See *Configurable instruction enables and disables, and trap controls* on page D1-1575.
- Instruction Aborts generated by the memory address translation system that are associated with attempts to execute instructions from areas of memory that generate faults.
- Data Aborts generated by the memory address translation system that are associated with attempts to read or write memory that generate faults.
- Data Aborts caused by a misaligned address.
- All of the debug exceptions:
  - Software Breakpoint Instruction exceptions.
  - Breakpoint exceptions.
  - Watchpoint exceptions.
  - Vector Catch exceptions.
  - Software Step exceptions.
- In an implementation that supports the trapping of floating-point exceptions, exceptions caused by trapped IEEE floating-point exceptions, see *Floating-point exception traps* on page D1-1568.
- In some implementations, External aborts. External aborts are failed memory accesses, and include accesses to those parts of the memory system that occur during the address translation. The ARMv8 architecture permits, but does not require, implementations to treat such exceptions synchronously. See *External aborts* on page D3-1727.

This remainder of this section contains the following:

- *Routing exceptions to EL2* on page D1-1564.
- *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.
- *Effect of Data Aborts* on page D1-1567.
- *Floating-point exception traps* on page D1-1568.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D1.13.1 Routing exceptions to EL2

When HCR_EL2.TGE is 1, any exception taken from Non-secure EL0 that would be taken to Non-secure EL1 is, instead, routed to EL2. This means that an application can execute at Non-secure EL0 without using any functionality at Non-secure EL1.

**Note**

Implementations typically use the following Exception level and software hierarchy in Non-secure state:

**EL2** Hypervisor.

**EL1** Operating system.

**EL0** Application.

In such an implementation, setting HCR_EL2.TGE to 1 means that an application can run at Non-secure EL0 under the direct control of a hypervisor executing at EL2, with no operating system involvement.

### D1.13.2 Synchronous exception prioritization for exceptions taken to AArch64

In principle, any single instruction can generate a number of different synchronous exceptions, between the fetching of the instruction, its decode, and eventual execution. For exceptions taken to an Exception level that is using AArch64, these are prioritized as follows, where 1 is the highest priority.

**Note**

The priority numbering in this list only shows the relative priorities of exceptions taken to an Exception level that is using AArch64. This numbering has no global significance and, for example, does not correlate with the equivalent AArch32 list in *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778.

1. Software Step exceptions. See *Software Step exceptions* on page D2-1686.
2. Misaligned PC exceptions. See *PC alignment checking* on page D1-1531.
3. Instruction Abort exceptions. See *Exception from an Instruction abort* on page D1-1549 and *AArch64 prioritization of synchronous aborts from a single stage of address translation* on page D4-1830.
4. Breakpoint exceptions or Address Matching Vector Catch exceptions. See:
   - *Breakpoint exceptions* on page D2-1653.
   - *Vector Catch exceptions* on page D2-1685.

   Vector Catch exceptions are only taken from AArch32 state.

   **Note**

   An Exception Trapping Vector Catch exception is generated on exception entry for an exception that has been prioritized as described in *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778. This means that it is outside the scope of the description of this section.
5. Illegal Execution state exceptions. See *Illegal return events from AArch64 state* on page D1-1553.
6. Exceptions taken from EL1 to EL2 because of one of the following configuration settings:
   - HSTR_EL2.T*n*.
   - HCR_EL2.TIDCP.

   **Note**

   These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state the equivalent controls are HSTR.T*n* and HCR.TIDCP.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

7. Undefined Instruction exceptions that occur as a result of one or more of the following:
   - An attempt to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation.
   - An attempt to execute an instruction that is defined never to be accessible at the current Exception level regardless of any enables or traps.
   - Debug state execution of an instruction encoding that is unallocated in Debug state.
   - Non-debug state execution of an instruction encoding that is unallocated in Non-debug state.
   - Execution of an `HVC` instruction, when `HVC` instructions are disabled by SCR_EL3.HCE or HCR_EL2.HCD.
   - Execution of an `MSR` or `MRS` instruction to SP_EL0 when the value of SPSel is 0.
   - Execution of an `HLT` instruction when `HLT` instructions are disabled by EDSCR.HDE.
   - In Debug state:
     — Execution of a `DCPS1` instruction in Non-secure EL0 when HCR_EL2.TGE is 1.
     — Execution of a `DCPS2` instruction in EL1 or EL0 when SCR_EL3.NS is 0 or when EL2 is not implemented.
     — Execution of a `DCPS3` instruction when EDSCR.SDD is 1 or when EL3 is not implemented.
     — When the value of EDSCR.SDD is 1, execution in EL2, EL1, or EL0 of an instruction that is trapped to EL3.
   - When executing in AArch32 state, execution of an instruction that is UNDEFINED as a result of any of:
     — Being in an IT block when SCTLR_EL1.ITD is 1.
     — Executing a `SETEND` instruction executed SCTLR_EL1.SED.
     — Executing a CP15DMB, CP15DSB, or CP15ISB barrier instruction when SCTLR_EL1.CP15BEN is 0.

     **Note**

     These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state the equivalent controls are SCTLR.{ITD, SED, CP15BEN}, with additional controls HSCTLR.{ITD, SED, CP15BEN}.

     See *Disabling or enabling EL0 use of AArch32 deprecated functionality* on page D1-1579
   - When executing in AArch32 state, execution of an instruction that is UNDEFINED because at least one of FPCR.{Stride, Len} is nonzero, when programming these bits to nonzero values is supported. See *Floating-point exception traps* on page G1-3844.

     **Note**

     — This case applies only when EL0 is using AArch32 and EL1 is using AArch64. The exception generated by the attempted execution at EL0 of the UNDEFINED instruction is taken to EL1 using AArch64.
     — When EL1 is using AArch32, the corresponding controls are FPSCR.{Stride, Len}, and any exception generated by the attempted execution at EL0 or EL1 of an instruction that is UNDEFINED because of a nonzero {Stride, Len} value is taken to EL1 using AArch32.

8. Exceptions taken to EL1, or taken to EL2 because the value of HCR_EL2.TGE is 1, that are generated because of configurable access to instructions, and that are not covered by any of priorities 1-7.

   **Note**

   When EL2 is using AArch32, the equivalent control for routing exceptions to EL2 is HCR.TGE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

9. Exceptions taken from EL0 to EL2 because of one of the following configuration settings:
   - HSTR_EL2.T*n*.
   - HCR_EL2.TIDCP.

   **Note**

   These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state the equivalent controls are HSTR.T*n* and HCR.TIDCP.

10. Exceptions taken to EL2 because of configuration settings in the CPTR_EL2.

    **Note**

    These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state, the equivalent controls are in the HCPTR.

11. Exceptions taken to EL2 because of one of the following configuration settings:
    - Any setting in HCR_EL2, other than the TIDCP bit.
    - Any setting in CNTHCTL_EL2.
    - Any setting in MDCR_EL2.

    **Note**

    These are the controls for exceptions taken to AArch64 state. For exceptions taken to AArch32 state, the equivalent controls are:
    - Any setting in HCR, other than the TIDCP bit.
    - Any setting in CNTHCTL or HDCR.

12. Exceptions taken to EL2 because of configurable access to instructions, and that are not covered by any of priorities 1-11.
13. Exceptions caused by the `SMC` instruction being UNDEFINED because the value of SCR_EL3.SMD is 1.
14. Exceptions caused by the execution of an Exception generating instruction:
    - For exceptions taken from AArch64 state, *Branches, Exception generating, and System instructions* on page C3-138 defines these instructions.
    - When executing in AArch32 state, the exception-generating instructions are `SVC`, `HVC`, `SMC`, and `BKPT`.
15. Exceptions taken to EL3 because of configuration settings in the CPTR_EL3.
16. Exceptions taken to EL3 from Secure EL1 using AArch32, because of execution of the instructions listed in *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-1602.
17. Exceptions taken to EL3 from EL0, EL1, or EL2 because of configuration settings in the MDCR_EL3.
18. Exceptions taken to EL3 because of configurable access to instructions, and that are not covered by any of priorities 1-17.
19. Trapped floating-point exceptions, if supported. See *Floating-point exception traps* on page D1-1568.
20. Stack Pointer Alignment faults. See *Stack pointer alignment checking* on page D1-1531.
21. Data Abort exceptions other than a Data Abort exception generated by a Synchronous external abort that was not generated by a translation table walk. That is, any Data Abort exception that is not covered by item 23. See *Exception from a Data abort* on page D1-1549 and *AArch64 prioritization of synchronous aborts from a single stage of address translation* on page D4-1830. It is IMPLEMENTATION DEFINED whether Synchronous external aborts are prioritized here or as item 23.
22. Watchpoint exceptions. See *Watchpoint exceptions* on page D2-1671.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

23. Data Abort exception generated by a Synchronous external abort that was not generated by a translation table walk, see *External aborts* on page D3-1727. It is IMPLEMENTATION DEFINED whether Synchronous external aborts are prioritized here or as item 21.

For items 21-23, if an instruction results in more than one single-copy atomic memory access, the prioritization between synchronous exceptions generated on each of those different memory accesses is not defined by the architecture.

**Note**

Exceptions generated by a translation table walk are reported and prioritized as either an Instruction Abort exception, priority 3 in this list, or a Data Abort exception, priority 21 in this list. See also *AArch64 prioritization of synchronous aborts from a single stage of address translation* on page D4-1830.

### D1.13.3 Effect of Data Aborts

If an instruction that stores to memory generates a Data Abort, the value of each memory location that instruction stores to is either:

- Unchanged, if one of the following applies:
  - An MMU fault is generated.
  - A Watchpoint exception is generated.
  - An external abort is generated, if that external abort is taken synchronously.

    **Note**

    If an external abort is taken asynchronously, using the SError interrupt, it is outside the scope of the architecture to define the effect of the store on the memory location, because it depends on the system-specific nature of the external abort. However, in general, ARM recommends that such memory locations are not updated.
- UNKNOWN for any location for which no exception and no debug event is generated.

For external aborts and Watchpoint exceptions, the size of a memory location is defined as being the size for which a memory access is single-copy atomic.

**Note**

For the definition of a single-copy atomic access, see *Properties of single-copy atomic accesses* on page B2-81.

For Data Aborts from load or store instructions executed in AArch64 state, if the:

**Data Abort is taken synchronously**

- If the load or store instruction specifies writeback of a new base address, the base address is restored to the original value on taking the exception.
- If the instruction was a load to either the base address register or the offset register, that register is restored to the original value. Any other destination registers become UNKNOWN.
- If the instruction was a load that does not load the base address register or the offset register, then the destination registers become UNKNOWN.

**Data Abort is taken asynchronously, using the SError interrupt**

If the instruction was a load, the destination registers of the load take an UNKNOWN value if the SError interrupt is taken at a point in the instruction stream after the load.

**Note**

Data Aborts taken asynchronously are known as Asynchronous Aborts in AArch32 state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D1.13.4 Floating-point exception traps

Execution of a floating-point instruction can generate an exceptional condition, called a *floating-point exception*.

The ARMv8-A architecture supports synchronous exception generation in the event of any or all of the following floating-point exceptions:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

Whether an implementation includes synchronous exception generation for these floating-point exceptions is IMPLEMENTATION DEFINED:

- For an implementation that does provide this capability, FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} are the control bits that enable synchronous exception generation for each of the different floating-point exceptions.
- For an implementation that does not provide this capability, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits are RAZ/WI.

**Note**

The ARMv8-A architecture does not support asynchronous reporting of floating-point exceptions.

When generating synchronous exceptions for one or more floating-point exceptions is enabled, the synchronous exceptions generated by the floating-point exception traps are taken to the lowest Exception level that can handle such an exception, while adhering to the rule that an exception can never be taken to a lower Exception level. This means that trapped floating-point exceptions taken:

- From EL0 are taken to EL1, unless they are taken from Non-secure state when the value of HCR_EL2.TGE is 1, when they are taken to EL2 instead.
- From EL1 are taken to EL1.
- From EL2 are taken to EL2.
- From EL3 are taken to EL3.

The exception is reported in the ELR_ELx for the Exception level to which it is taken.

In an implementation that includes synchronous exception generation for floating-point exceptions:

- Synchronous exception generation applies to floating-point exceptions generated by scalar SIMD and floating-point instructions executed in AArch64 state.
- The registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception. An implementation is permitted not to restore the cumulative exception flags in the event of such an exception.
- When the execution of separate operations in separate SIMD elements causes multiple floating-point exceptions, the ESR_ELx reports one exception associated with one element that the instruction uses. The architecture does not specify which element is reported, however the element that is reported is identified in the ESR_ELx.

The `AArch64.FPTrappedException()` and `FPProcessException()` pseudocode functions describe the handling of trapped floating-point exceptions generated in AArch64 state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.14 Asynchronous exception types, routing, masking and priorities

In the ARMv8-A architecture, asynchronous exceptions that are taken to AArch64 state are also known as *interrupts*.

There are two types of interrupts:

**Physical interrupts** Are signals sent to the PE from outside the PE. They are:
- SError. System Error.
- IRQ.
- FIQ.

**Virtual interrupts** Are interrupts that a Hypervisor executing in EL2 can enable. A virtual interrupt is taken from Non-secure EL0 or Non-secure EL1 to a Guest OS running in Non-secure EL1.

Virtual interrupts have names that correspond to the physical interrupts:
- vSError.
- vIRQ.
- vFIQ.

**Note**

The AArch64 SError interrupt replaces the AArch32 asynchronous abort. The new name better describes the nature of the exception, and means that it is categorized as a unique exception class, with EC encoding `0x2F`.

An external abort generated by the memory system might be taken asynchronously using the SError interrupt. The effect of a failed memory access is described in *Effect of Data Aborts* on page D1-1567.

Each physical interrupt type can be assigned a target Exception level of EL1, EL2 or EL3, as shown in *Asynchronous exception routing*.

When an interrupt occurs:

- On taking an SError or a vSError interrupt to an Exception level using AArch64, the Exception Syndrome register for that Exception level is updated with the encoding for an SError interrupt. See *Exception classes and the ESR_ELx syndrome registers* on page D1-1539.
- On taking an IRQ, vIRQ, FIQ or vFIQ interrupt to an Exception level using AArch64, the Exception Syndrome register for that Exception level is not updated.

The remainder of this section contains the following:
- *Asynchronous exception routing*.
- *Asynchronous exception masking* on page D1-1570.
- *Virtual interrupts* on page D1-1572.
- *Prioritization and recognition of asynchronous exceptions* on page D1-1574.
- *Taking an interrupt or other exception during a multiple-register load or store* on page D1-1574.

### D1.14.1 Asynchronous exception routing

The following tables show the routing of physical interrupts when the highest implemented Exception level is using AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In the tables, C indicates that the interrupt is not taken, regardless of the Process state interrupt mask.

**Table D1-11 Routing when both EL3 and EL2 are implemented**

| SCR_EL3.EA<br>SCR_EL3.IRQ<br>SCR_EL3.FIQ | SCR_EL3.RW | AMO[a]<br>IMO[a]<br>FMO[a] | Target Exception level when executing in: | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | Non-secure | | | Secure | | |
| | | | EL0 | EL1 | EL2 | EL0 | EL1 | EL3 |
| 0 | 0 | 0 | EL1 | EL1 | EL2 | EL1 | EL1 | C |
| | X | 1 | EL2 | EL2 | EL2 | EL1 | EL1 | C |
| | 1 | 0 | EL1 | EL1 | C | EL1 | EL1 | C |
| 1 | X | X | EL3 | EL3 | EL3 | EL3 | EL3 | EL3 |

a. If EL2 is using AArch64, these are the HCR_EL2.{AMO, IMO, FMO} control bits. If EL2 is using AArch32, these are the HCR{AMO, IMO, FMO} control bits. If HCR_EL2.TGE or HCR.TGE is 1, these bits are treated as being 1 other than for a direct read.

**Table D1-12 Routing when EL3 is implemented and EL2 is not implemented**

| SCR_EL3.EA<br>SCR_EL3.IRQ<br>SCR_EL3.FIQ | Target Exception level when executing in: | | | | |
|---|---|---|---|---|---|
| | Non-secure | | Secure | | |
| | EL0 | EL1 | EL0 | EL1 | EL3 |
| 0 | EL1 | EL1 | EL1 | EL1 | C |
| 1 | EL3 | EL3 | EL3 | EL3 | EL3 |

**Table D1-13 Routing when EL3 is not implemented and EL2 is implemented**

| HCR_EL2.AMO[a]<br>HCR_EL2.IMO[a]<br>HCR_EL2.FMO[a] | Target Exception level when executing in: | | |
|---|---|---|---|
| | Non-secure | | |
| | EL0 | EL1 | EL2 |
| 1 | EL2 | EL2 | EL2 |
| 0 | EL1 | EL1 | C |

a. If HCR_EL2.TGE is 1, these bits are treated as being 1 other than for a direct read.

### D1.14.2 Asynchronous exception masking

When an interrupt is masked, it means that it cannot be taken. Instead, it remains pending.

When executing in AArch64 state, interrupts are masked implicitly when the target Exception level of the interrupt is lower than the current Exception level.

In addition, interrupts can be masked when the target Exception level is the current Exception level. The controls for this are:

**SError** PSTATE.A
**IRQ** PSTATE.I
**FIQ** PSTATE.F


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the target Exception level is higher than the current Exception level:

- If the target Exception level is EL2 or EL3, the interrupt cannot be masked by the PSTATE.{A, I, F} bits.
- If the target Exception level is EL1, the interrupt can be masked by the PSTATE.{A, I, F} bits.

**Note**

- The ability to execute in EL0 with interrupts to EL1 masked is required by some user level driver code.
- The PSTATE.{A, I, F} bits can mask both physical interrupts and virtual interrupts.
- The ARMv8-A architecture does not support *Non-maskable FIQ* (NMFI) operations. This means that it does not provide a configuration option to override the masking of FIQs by PSTATE.F.

On taking any exception to an Exception level using AArch64, all of PSTATE.{A, I, F} are set to 1, masking all interrupts that target that Exception level.

The following tables show the masking of physical interrupts when the highest implemented Exception level is using AArch64:

- For implementations that include both EL2 and EL3, see Table D1-14.
- For implementations that include EL3 but not EL2, see Table D1-15 on page D1-1572.
- For implementations that include EL2 but not EL3, see Table D1-16 on page D1-1572.

For the masking of virtual interrupts, see *Virtual interrupts* on page D1-1572.

In the tables:

**A** When the interrupt is asserted it is taken regardless of the value of the Process state interrupt mask.

**B** When the interrupt is asserted it is subject to the corresponding Process state mask. If the value of the mask is 1 then the interrupt is not taken. If the value of the mask is 0 the interrupt is taken.

**C** When the interrupt is asserted it is not taken, regardless of the value of the Process state interrupt mask.

**Table D1-14 Physical interrupt masking when both EL3 and EL2 are implemented**

| SCR_EL3.EA<br>SCR_EL3.IRQ<br>SCR_EL3.FIQ | SCR_EL3.RW | AMO[a]<br>IMO[a]<br>FMO[a] | Target Exception level | Effect of the interrupt mask when executing in: | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Non-secure | | | Secure | | |
| | | | | EL0 | EL1 | EL2 | EL0 | EL1 | EL3 |
| 0 | 0 | 0 | EL1 | B | B | B | B | B | C |
| | | 1 | EL2 | A | A | B | B | B | C |
| | 1 | 0 | EL1 | B | B | C | B | B | C |
| | | 1 | EL2 | A | A | B | B | B | C |
| 1 | X | X | EL3 | A | A | A | A | A | B |

a. If EL2 is using AArch64, these are the HCR_EL2.{AMO, IMO, FMO} control bits. If EL2 is using AArch32, these are the HCR{AMO, IMO, FMO} control bits. If HCR_EL2.TGE or HCR.TGE is 1, these bits are treated as being 1 other than a direct read.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D1-15 Physical interrupt masking when EL3 is implemented and EL2 is not implemented**

| SCR_EL3.EA<br>SCR_EL3.IRQ<br>SCR_EL3.FIQ | Target Exception level | Effect of the interrupt mask when executing in: | | | | |
|---|---|---|---|---|---|---|
| | | Non-secure | | Secure | | |
| | | EL0 | EL1 | EL0 | EL1 | EL3 |
| 0 | EL1 | B | B | B | B | C |
| 1 | EL3 | A | A | A | A | B |

**Table D1-16 Physical interrupt masking when EL3 is not implemented and EL2 is implemented**

| HCR_EL2.AMO[a]<br>HCR_EL2.IMO[a]<br>HCR_EL2.FMO[a] | Target Exception level | Effect of the interrupt mask when executing in: | | |
|---|---|---|---|---|
| | | Non-secure | | |
| | | EL0 | EL1 | EL2 |
| 0 | EL1 | B | B | C |
| 1 | EL2 | A | A | B |

a. If HCR_EL2.TGE is 1, these bits are treated as being 1 other than for a direct read.

### D1.14.3 Virtual interrupts

Setting an HCR_EL2.{FMO, IMO, AMO} routing control bit to 1 enables the corresponding virtual interrupt, unless HCR_EL2.TGE is 1. When the value of HCR_EL2.TGE is 1 all virtual interrupts are disabled.

Virtual interrupts can only be taken from Non-secure EL0 to EL1 or from Non-secure EL1 to EL1. When a virtual interrupt type is enabled, that type of interrupt can be generated by:

- Software setting the corresponding virtual interrupt pending bit, HCR_EL2.{VSE, VI, VF}, to 1.
- For a vIRQ or a vFIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller. See, for example, the *ARM Generic Interrupt Controller Architecture Specification*.

**Note**

For a usage model for virtual interrupts, see *Virtual interrupt usage model* on page D1-1521.

When a virtual interrupt is disabled:

- It cannot be taken.
- It cannot be seen in the ISR_EL1.

Each virtual interrupt type can be masked when execution is in Non-secure EL1 or EL0, by using the same Process State mask bits that mask the physical interrupts, PSTATE.{A, I, F}.

When execution is in Secure state, or in EL3 or EL2, all types of virtual interrupt are always masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-17 summarizes the bits that enable virtual interrupts and the bits that cause virtual interrupts to be pending.

**Table D1-17 HCR_EL2 interrupt control bits**

| Virtual interrupt type | Enable control | Cause a virtual interrupt to be pending |
|---|---|---|
| vSError | HCR_EL2.AMO | HCR_EL2.VSE |
| vIRQ | HCR_EL2.IMO | HCR_EL2.VI |
| vFIQ | HCR_EL2.FMO | HCR_EL2.VF |

On taking a vIRQ or a vFIQ interrupt, the corresponding virtual interrupt pending bit in the HCR_EL2 retains its state.

On taking a vSError interrupt, HCR_EL2.VSE is cleared to 0.

**Note**

This means that if the virtual interrupt pending bits are used, the vIRQ or vFIQ exception handler must cause software executing in EL2 or EL3 to set their corresponding virtual interrupt pending bits to 0.

As with physical interrupts:

- Taking a vSError interrupt to an Exception level using AArch64 updates ESR_EL1 with the dedicated encoding for an SError interrupt. For the encoding, see *Exception classes and the ESR_ELx syndrome registers* on page D1-1539.
- Taking a vIRQ or a vFIQ interrupt to an Exception level using AArch64 does not update the ESR_EL1.

The following table shows the masking of virtual interrupts when the highest implemented Exception level is using AArch64. In the table:

**B** When the interrupt is asserted it is subject to the corresponding Process state mask. If the value of the mask is 1 then the interrupt is not taken. If the value of the mask is 0 the interrupt is taken.

**C** When the interrupt is asserted it is not taken, regardless of the value of the Process state interrupt mask.

**Table D1-18 Virtual interrupt masking**

| SCR_EL3.EA<br>SCR_EL3.IRQ<br>SCR_EL3.FIQ | FMO[a]<br>IMO[a]<br>AMO[a] | TGE[a] | Effect of the interrupt mask when executing in: | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | Non-secure | | | Secure | | |
| | | | EL0 | EL1 | EL2 | EL0 | EL1 | EL3 |
| X | 0 | X | C | C | C | C | C | C |
| X | 1 | 0 | B | B | C | C | C | C |
| X | 1 | 1 | C | C | C | C | C | C |

a. If EL2 is using AArch64, these are the HCR_EL2.{TGE, AMO, IMO, FMO} control bits. If EL2 is using AArch32, these are the HCR{TGE, AMO, IMO, FMO} control bits.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D1.14.4 Prioritization and recognition of asynchronous exceptions

The ARMv8-A architecture does not define when interrupts are taken. The prioritization of interrupts, including virtual interrupts, is IMPLEMENTATION DEFINED.

Any interrupts that are pending prior to one of the following context synchronizing events, are taken before the first instruction after the context synchronizing event, provided that the interrupt is not masked:

- An `ISB` instruction.
- Exception entry.
- Exception return.
- Exit from Debug state.

**Note**

- If the first instruction after the context synchronizing event generates a synchronous exception, then the architecture does not define whether the PE takes the interrupt or the synchronous exception first.
- The ISR_EL1 register indicates whether an interrupt is pending.
- Interrupts are masked when the PE is in Debug state, so this list of context synchronizing events does not include the `DCPS` and `DRPS` instructions.

In the absence of a specific requirement to take an interrupt, the architecture only requires that unmasked pending interrupts are taken in finite time.

### D1.14.5 Taking an interrupt or other exception during a multiple-register load or store

In AArch64 state, interrupts can be taken during a sequence of memory accesses caused by a single load or store instruction. This is true regardless of the memory type being accessed.

If an interrupt, or another exception, is taken from AArch64 during the execution of an instruction that performs a sequence of memory accesses, rather than a single single-copy atomic access, then:

- For a load, any register being loaded by the instruction other than ones used in the generation of the address by the instruction, can contain an UNKNOWN value. Registers used in the generation of the address are restored to their initial value.
- For a store, any data location being stored to by the instruction can contain an UNKNOWN value.
- For either a load or a store, if the instruction specifies writeback of the base address, then that register is restored to its initial value.

**Note**

- This interrupt behavior is in contrast to behavior in AArch32 state, when interrupts cannot be taken during a sequence of memory access caused by a single load or store instruction.
- In both Execution states, synchronous data abort exceptions can be taken during the execution of an instruction that performs a sequence of memory accesses.
- Software must avoid using multiple-register load and store instructions for accesses to Device memory, particularly to Device memory with the non-Gathering attribute, because an exception taken during the load or store can result in repeated accesses.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.15 Configurable instruction enables and disables, and trap controls

This section describes the controls provided by AArch64 state for enabling, disabling, and trapping particular instructions. Each control is categorized as an *instruction enable*, an *instruction disable*, or a *trap control*:

**Instruction enables and instruction disables**

Enable or disable the use of one or more particular instructions at a particular Exception level and Security state.

When an instruction is disabled as a result of an instruction enable or disable, it is UNDEFINED.

**Trap controls**

Control whether one or more particular instructions, whenever executed at a particular Exception level, are *trapped*.

A trapped instruction generates a *Trap exception*.

For trap controls provided by:

**EL1** Trap exceptions are taken to EL1, unless routed from Non-secure EL0 to EL2 because HCR_EL2.TGE is 1 as described in *Routing exceptions to EL2* on page D1-1564.
For descriptions of these controls see *EL1 configurable controls* on page D1-1576.

**EL2** Trap exceptions are taken to EL2.
For descriptions of these controls see *EL2 configurable controls* on page D1-1584.

**EL3** Trap exceptions are taken to EL3.
For descriptions of these controls see *EL3 configurable controls* on page D1-1601.

An exception generated as a result of an instruction enable or disable, or a trap control, is only taken if both of the following apply:

- The instruction generating the exception does not also generate a higher priority exception. *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 defines the prioritization of different exceptions on the same instruction.
- The instruction is not UNPREDICTABLE in the PE state it is executed in. UNPREDICTABLE instructions can generate exceptions as a result of these controls, but the architecture does not require them to do so.

Exceptions generated as a result of these controls are synchronous exceptions.

For:

- Undefined Instruction exceptions, the PE reports EC value `0x00` in the ESR_ELx. This is the value for an exception for an unknown or uncategorized reason.
- Trap exceptions, the PE reports the applicable EC value and associated syndrome in the ESR_ELx. Table D1-8 on page D1-1540 includes the EC values that are used for Trap exceptions.

**Note**

- A particular control might have a mnemonic that suggests it is different type of control to the control type it is categorized as. For example, SCTLR_EL1.DZE is a trap control even though DZE means DC ZVA Enable.
- In addition to the controls described in this section, a *routing control*, HCR_EL2.TGE, can be used to route exceptions from EL0 to EL2. See *Routing exceptions to EL2* on page D1-1564.
- An implementation might provide additional controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.

This section is organized as follows:

- *Register access instructions* on page D1-1576.
- *EL1 configurable controls* on page D1-1576.
- *EL2 configurable controls* on page D1-1584.
- *EL3 configurable controls* on page D1-1601.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D1.15.1 Register access instructions

When an instruction is disabled or trapped, the exception is taken before execution of the instruction. This means that if the instruction is a register access instruction:

- No access is made before the exception is taken.
- Side-effects that are normally associated with the access do not occur before the exception is taken.

### D1.15.2 EL1 configurable controls

These controls are in _EL1 System registers. The resulting exceptions might be taken from either Execution state. SPSR_EL1.M[4] indicates which Execution state the exception was taken from.

Table D1-19 shows the _EL1 System registers that contain these controls.

**Table D1-19 _EL1 registers that contain instruction enables and disables, and trap controls**

| Register name | Register description |
|---|---|
| SCTLR_EL1 | System Control Register, EL1 |
| CPACR_EL1 | Architectural Feature Access Control Register |
| MDSCR_EL1 | Monitor System Debug Control Register |
| PMUSERENR_EL0 | Performance Monitors User Enable Register |

Table D1-20 summarizes the controls.

**Table D1-20 Instruction enables and disables, and trap controls, provided by EL1**

| Control | Control type[a] | Description |
|---|---|---|
| SCTLR_EL1.UCI | T | *Traps to EL1 of EL0 execution of cache maintenance instructions* on page D1-1577 |
| SCTLR_EL1.UCT | T | *Traps to EL1 of EL0 accesses to the CTR_EL0* on page D1-1577 |
| SCTLR_EL1.{nTWE, nTWI} | T | *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-1578 |
| SCTLR_EL1.DZE | T | *Traps to EL1 of EL0 execution of DC ZVA instructions* on page D1-1578 |
| SCTLR_EL1.UMA | T | *Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks* on page D1-1579 |
| SCTLR_EL1.{SED, ITD}<br>SCTLR_EL1.CP15BEN | D<br>E | *Disabling or enabling EL0 use of AArch32 deprecated functionality* on page D1-1579 |
| CPACR_EL1.TTA | T | *Traps to EL1 of EL0 and EL1 System register accesses to the trace registers* on page D1-1580 |
| CPACR_EL1.FPEN | T | *Traps to EL1 of EL0 and EL1 accesses to SIMD and floating-point functionality* on page D1-1581 |
| MDSCR_EL1.TDCC | T | *Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers* on page D1-1581 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D1-20 Instruction enables and disables, and trap controls, provided by EL1 (continued)**

| Control | Control type[a] | Description |
|---|---|---|
| CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} | T | *Traps to EL1 of EL0 accesses to the Generic Timer registers* on page D1-1582 |
| PMUSERENR_EL0.{ER, CR, SW, EN} | T | *Traps to EL1 of EL0 accesses to Performance Monitors registers* on page D1-1582 |

a. See Table D1-21.

**Table D1-21 Control types, for exceptions taken to EL1**

| Abbreviation | Type | See |
|---|---|---|
| D | Disable | *Instruction enables and instruction disables* on page D1-1575 |
| E | Enable | *Instruction enables and instruction disables* on page D1-1575 |
| T | Trap | *Trap controls* on page D1-1575 |

### Traps to EL1 of EL0 execution of cache maintenance instructions

SCTLR_EL1.UCI traps EL0 execution of cache maintenance instructions to EL1:

**1** EL0 execution of cache maintenance instructions is not trapped to EL1.

**0** Any attempt to execute a cache maintenance instruction at EL0 is trapped to EL1.

Table D1-22 shows the instructions that are trapped to EL1, and how the exceptions are reported in ESR_EL1.

**Table D1-22 Instructions trapped to EL1 when SCTLR_EL1.UCI is 0**

| Traps from | Trapped instructions | Syndrome reporting in ESR_EL1 |
|---|---|---|
| AArch64 state | DC CVAU, DC CIVAC, DC CVAC, IC IVAU | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18`[a] |
| AArch32 state | n/a | n/a |

a. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, these Trap exceptions are routed to EL2 and are reported in ESR_EL2 using the same EC values as shown in the table.

### Traps to EL1 of EL0 accesses to the CTR_EL0

SCTLR_EL1.UCT traps EL0 accesses to the CTR_EL0 to EL1:

**1** EL0 accesses to the CTR_EL0 are not trapped to EL1.

**0** EL0 accesses to the CTR_EL0 are trapped to EL1.

Table D1-23 shows how the exceptions are reported in ESR_EL1.

**Table D1-23 Register accesses trapped to EL1 when SCTLR_EL1.UCT is 0**

| Traps from | Register | Syndrome reporting in ESR_EL1 |
|---|---|---|
| AArch64 | CTR_EL0 | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18`[a] |
| AArch32 | n/a | n/a |

a. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, these Trap exceptions are routed to EL2 and are reported in ESR_EL2 using the same EC values as shown in the table.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Traps to EL1 of EL0 execution of WFE and WFI instructions

SCTLR_EL1.{nTWE, nTWI} trap EL0 execution of WFE and WFI instructions to EL1:

**SCTLR_EL1.nTWE**

**1** EL0 execution of WFE instructions is not trapped to EL1.

**0** Any attempt to execute a WFE instruction at EL0 is trapped to EL1, if the instruction would otherwise have caused the PE to enter a low-power state.

**SCTLR_EL1.nTWI**

**1** EL0 execution of WFI instructions is not trapped to EL1.

**0** Any attempt to execute a WFI instruction at EL0 is trapped EL1, if the instruction would otherwise have caused the PE to enter a low-power state.

Table D1-24 shows how the exceptions are reported in ESR_EL1.

**Table D1-24 Instructions trapped to EL1 when SCTLR_EL1.{nTWE, nTWI} are 0**

| Trap control | Traps from | Trapped instructions | Syndrome reporting in ESR_EL1 |
|---|---|---|---|
| SCTLR_EL1.nTWE | Both Execution states | WFE | Trapped WFI or WFE instruction, using EC value 0x01[a] |
| SCTLR_EL1.nTWI | | WFI | |

a. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, these Trap exceptions are routed to EL2 and are reported in ESR_EL2 using the same EC values as shown in the table.

In AArch32 state, the attempted execution of a conditional WFE or WFI instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait for Event mechanism and Send event* on page D1-1611.
- *Wait For Interrupt* on page D1-1614.

### Traps to EL1 of EL0 execution of DC ZVA instructions

SCTLR_EL1.DZE traps EL0 execution of DC ZVA instructions to EL1:

**1** EL0 execution of DC ZVA instructions is not trapped to EL1.

**0** Any attempt to execute a DC ZVA instruction at EL0 is trapped to EL1. Reading the DCZID_EL0 returns a value that indicates that DC ZVA instructions are not implemented.

Table D1-25 shows how the exceptions are reported in ESR_EL1.

**Table D1-25 Instruction trapped to EL1 when SCTLR_EL1.DZE is 0**

| Traps from | Trapped instruction | Syndrome reporting in ESR_EL1 |
|---|---|---|
| AArch64 state | DC ZVA | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18[a] |
| AArch32 state | n/a | n/a |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

a. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, these Trap exceptions are routed to EL2 and are reported in ESR_EL2 using the same EC values as shown in the table.

### Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks

SCTLR_EL1.UMA traps EL0 execution of MSR and MRS instructions that access the PSTATE.{D, A, I, F} masks to EL1:

**1** EL0 execution of MSR or MRS instructions that access the DAIF is not trapped to EL1.

**0** Any attempt at EL0 to execute an MSR or an MRS instruction that accesses the DAIF is trapped to EL1.

Table D1-26 shows how the exceptions are reported in ESR_EL1.

**Table D1-26 Instructions trapped to EL1 when SCTLR_EL1.UMA is 0**

| Taken from | Disabled instructions | Syndrome reporting in ESR_EL1 |
|---|---|---|
| AArch64 state | MRS, MSR (register), MSR (immediate), that access the DAIF | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18[a] |
| AArch32 state | n/a | n/a |

a. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, these Trap exceptions are routed to EL2 and are reported in ESR_EL2 using the same EC values as shown in the table.

### Disabling or enabling EL0 use of AArch32 deprecated functionality

Table D1-27 shows the deprecated AArch32 functionality that might have disable controls in the SCTLR_EL1:

- The SED control is always implemented.
- Whether each of the ITD, CP15BEN controls is implemented is IMPLEMENTATION DEFINED. If a control is not implemented then the associated functionality cannot be disabled.

These SCTLR_EL1 controls apply only to execution at EL0 using AArch32. When an instruction is disabled by one of these controls, it is UNDEFINED at EL0 using AArch32. The table shows how the exceptions are reported in ESR_EL1.

**Table D1-27 EL1 controls for disabling and enabling EL0 use of AArch32 deprecated functionality**

| Deprecated AArch32 functionality | Instruction enable or disable in the SCTLR_EL1 | Disabled instructions | Syndrome reporting in ESR_EL1[a] |
|---|---|---|---|
| SETEND instructions | SED[b] | SETEND instructions | Exception for an unknown reason, using EC value 0x00 |
| Some uses of IT instructions | ITD[c] | See the SCTLR_EL1.IT description | |
| Accesses to the CP15DMB, CP15DSB, and CP15ISB barrier instructions | CP15BEN[d] | MCR accesses to the CP15DMB, CP15DSB, and CP15ISB instructions | |

a. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, the exception is routed to EL2 and reported in ESR_EL2 using the EC value shown in the table.

b. SETEND instruction disable. SETEND instructions are disabled when the value of this field is 1.

c. IT instruction disable. If this control is implemented, some uses of IT instructions are disabled when the value of this field is 1.

d. CP15 memory barrier enable. If this control is implemented, the specified register accesses are disabled when the value of CP15BEN is 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- The uses of the IT instruction, and use of the CP15DMB, CP15DSB, and CP15ISB barrier instructions, are deprecated for performance reasons.
- The SCTLR provides similar controls that apply when EL1 is using AArch32, and the HSCTLR provides similar controls that apply when EL2 is using AArch32.

### Traps to EL1 of EL0 and EL1 System register accesses to the trace registers

CPACR_EL1.TTA traps EL0 and EL1 System register accesses to the trace registers to EL1.

**1** EL0 and EL1 System register accesses to the trace registers are trapped to EL1.

**0** EL0 and EL1 System register accesses to the trace registers are not trapped to EL1.

**Note**

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED. A resulting Undefined Instruction exception is higher priority than a CPACR_EL1.TTA Trap exception.
- The ARMv8-A architecture does not provide traps on trace register accesses through the optional Memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Register access instructions* on page D1-1576.

Table D1-28 shows the registers for which accesses are trapped to EL1 when CPACR_EL1.TTA is 1, and how the exceptions are reported in ESR_EL1.

**Table D1-28 Register accesses trapped to EL1 when CPACR_EL1.TTA is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL1 |
|---|---|---|
| AArch64 state | All implemented trace registers | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18.[a] |
| AArch32 state | All implemented trace registers | For accesses using:<br>• MCR or MRC instructions, trapped MCR or MRC CP14 access, using EC value 0x05.[a]<br>• MCRR or MRRC instructions, trapped MCRR or MRRC CP14 access, using EC value 0x0C.[a] |

a. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, these Trap exceptions are routed to EL2 and are reported in ESR_EL2 using the same EC values as shown in the table.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Traps to EL1 of EL0 and EL1 accesses to SIMD and floating-point functionality

CPACR_EL1.FPEN traps EL0 and EL1 accesses to the SIMD and floating-point registers to EL1.

**Note**

. CPACR_EL1.FPEN is a two-bit field. For the definition of when the trap is enabled, see the CPACR_EL1 register description.

Table D1-29 shows the registers for which accesses are trapped, and how the exceptions are reported in ESR_EL1.

**Table D1-29 Register accesses trapped to EL1 by CPACR_EL1.FPEN**

| Traps from | Registers | Syndrome reporting in ESR_EL1 |
|---|---|---|
| EL0 and EL1 using AArch64, or EL0 using AArch64 only[a]. | FPCR, FPSR, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-S31 registers. See *The SIMD and floating-point registers, V0-V31* on page D1-1524. | Trapped access to a SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP, using EC value `0x07`[b] |
| EL0 using AArch32 | FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers. See *Advanced SIMD and floating-point System registers* on page G1-3843. | Trapped access to a SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP, using EC value `0x07`[b] |

a. Depending on the value of CPACR_EL1.FPEN. See the register description for details.
b. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, these Trap exceptions are routed to EL2 and are reported in ESR_EL2 using EC value `0x00`.

### Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers

MDSCR_EL1.TDCC traps EL0 accesses to the DCC registers to EL1:

**1** EL0 accesses to the DCC registers are trapped to EL1.

**0** EL0 accesses to the DCC registers are not trapped to EL1.

Traps of AArch32 PL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

Table D1-30 shows the accesses that are trapped, and how the exceptions are reported in ESR_EL1.

**Table D1-30 Accesses trapped to EL1 when MDSCR_EL1.TDCC is 1**

| Traps from | Trapped accesses | Syndrome reporting in ESR_EL1 |
|---|---|---|
| AArch64 state | Accesses to the MDCCSR_EL0, DBGDTR_EL0, DBGDTRTX_EL0 and DBGDTRRX_EL0 | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18`[a] |
| AArch32 state | • `MRC` of DBGDSCRint, DBGDTRRXint, and, if implemented, DBGDIDR, DBGDSAR and DBGDRAR.<br>• `MCR` to DBGDTRTXint. | Trapped `MCR` or `MRC` CP14 access, using EC value `0x05`[a] |
| | • `LDC` of DBGDTRTXint.<br>• `STC` of DBGDTRRXint. | Trapped `LDC` or `STC` access to CP14, using EC value `0x06`[a] |
| | If implemented, `MRRC` of DBGDSAR and DBGDRAR. | Trapped `MCRR` or `MRRC` CP14 access, using EC value `0x0C`[a] |

a. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, these Trap exceptions are routed to EL2 and are reported in ESR_EL2 using the same EC values as shown in the table.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Traps to EL1 of EL0 accesses to the Generic Timer registers

CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} trap EL0 accesses to the Generic Timer registers to EL1, as follows:

- CNTKCTL_EL1.EL0PTEN traps EL0 accesses to the physical timer registers.
- CNTKCTL_EL1.EL0VTEN traps EL0 accesses to the virtual timer registers.
- CNTKCTL_EL1.EL0PCTEN traps EL0 accesses to the frequency register and physical counter register.
- CNTKCTL_EL1.EL0VCTEN traps EL0 accesses to the frequency register and virtual counter register.

For all of these controls:

**1** EL0 accesses are not trapped to EL1.

**0** EL0 accesses are trapped to EL1.

Accesses to the frequency register, CNTFRQ_EL0 or CNTFRQ, are only trapped if CNTKCTL_EL1.EL0PCTEN and CNTKCTL_EL1.EL0VCTEN are both 0.

Table D1-31 shows the registers for which accesses are trapped, and how the exceptions are reported in ESR_EL1.

**Table D1-31 Register accesses trapped from EL0 to EL1 by CNTKCTL_EL1 trap controls**

| Traps from | Trap control | Registers | Syndrome reporting in ESR_EL1 |
|---|---|---|---|
| AArch64 state | EL0PTEN | CNTP_CTL_EL0, CNTP_CVAL_EL0, CNTP_TVAL_EL0 | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18`[a] |
| | EL0VTEN | CNTV_CTL_EL0, CNTV_CVAL_EL0, CNTV_TVAL_EL0 | |
| | EL0PCTEN | CNTFRQ_EL0, CNTPCT_EL0 | |
| | EL0VCTEN | CNTFRQ_EL0, CNTVCT_EL0 | |
| AArch32 state | EL0PTEN | CNTP_CTL, CNTP_CVAL, CNTP_TVAL | For accesses using:<br>• `MCR` or `MRC` instructions, trapped `MCR` or `MRC` CP15 access, using EC value `0x03`[a]<br>• `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` CP15 access, using EC value `0x04`[a] |
| | EL0VTEN | CNTV_CTL, CNTV_CVAL, CNTV_TVAL | |
| | EL0PCTEN | CNTFRQ, CNTPCT | |
| | EL0VCTEN | CNTFRQ, CNTVCT | |

a. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, these Trap exceptions are routed to EL2 and are reported in ESR_EL2 using the same EC values as shown in the table.

### Traps to EL1 of EL0 accesses to Performance Monitors registers

PMUSERENR_EL0.{ER, CR, SW, EN} trap EL0 accesses to the Performance Monitors registers to EL1. For each of these controls:

**1** EL0 accesses are not trapped to EL1.

**0** EL0 accesses are trapped to EL1.

For those Performance Monitors registers that more than one PMUSERENR_EL0.{ER, CR, SW, EN} control applies to, accesses are only trapped if all controls that apply are set to 0.

The accesses that these trap controls trap might be reads, writes, or both.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-32 shows:

- The registers for which EL0 accesses are trapped. For each register, the table shows the type of access trapped.
- How the exceptions are reported in ESR_EL1.

**Table D1-32 Register accesses trapped to EL1 when disabled from EL0**

| Traps from | Trap control | Registers | Access type | Syndrome reporting in ESR_EL1 |
|---|---|---|---|---|
| AArch64 state | ER | PMXEVCNTR_EL0, PMEVCNTR<n>_EL0 | R | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18[a] |
| | | PMSELR_EL0 | RW | |
| | CR | PMCCNTR_EL0 | R | |
| | SW | PMSWINC_EL0 | W | |
| | EN | PMCNTENSET_EL0, PMCNTENCLR_EL0, PMCR_EL0, PMOVSCLR_EL0, PMSWINC_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0, PMXEVTYPER_EL0, PMXEVCNTR_EL0, PMOVSSET_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMCCFILTR_EL0. | RW | |
| AArch32 state | ER | PMXEVCNTR, PMEVCNTR<n> | R | Trapped MCR or MRC CP15 access, using EC value 0x03[a] |
| | | PMSELR | RW | |
| | CR | PMCCNTR, accessed using an MRC | R | |
| | CR | PMCCNTR, accessed using an MRRC | R | Trapped MCRR or MRRC CP15 access, using EC value 0x04[a] |
| | SW | PMSWINC | W | Trapped MCR or MRC CP15 access, using EC value 0x03[a] |
| | EN | PMCNTENSET, PMCNTENCLR, PMCR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR, accessed using an MCR or MRC | RW | |
| | EN | PMCCNTR, accessed using an MCRR or MRRC | RW | Trapped MCRR or MRRC CP15 access, using EC value 0x04[a] |

a. If HCR_EL2.TGE is 1 and the PE is in Non-secure state, these Trap exceptions are routed to EL2 and are reported in ESR_EL2 using the same EC values as shown in the table.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D1.15.3 EL2 configurable controls

These controls are in _EL2 System registers. The resulting exceptions might be taken from either Execution state. SPSR_EL2.M[4] indicates which Execution state the exception was taken from.

These controls are ignored in Secure state.

Table D1-33 shows the _EL2 System registers that contain these controls.

**Table D1-33 _EL2 registers that contain instruction disables and trap controls**

| Register name | Register description |
|---|---|
| HCR_EL2 | Hypervisor Configuration Register |
| HSTR_EL2 | Hypervisor System Trap Register |
| CPTR_EL2 | Architectural Feature Trap Register, EL2 |
| MDCR_EL2 | Monitor Debug Configuration Register, EL2 |

Table D1-34 summarizes the controls.

**Note**

- There are no instruction enables at EL2.
- For completeness, Table D1-34 includes the routing control described in *Routing exceptions to EL2* on page D1-1564.

**Table D1-34 Instruction disables and trap controls provided by EL2**

| Control | Control type[a] | Description |
|---|---|---|
| HCR_EL2.{TRVM, TVM} | T | *Traps to EL2 of Non-secure EL1 accesses to virtual memory control registers* on page D1-1585 |
| HCR_EL2.HCD | D | *Disabling Non-secure state execution of HVC instructions* on page D1-1586 |
| HCR_EL2.TDZ | T | *Traps to EL2 of Non-secure EL0 and EL1 execution of DC ZVA instructions* on page D1-1586 |
| HCR_EL2.TGE | R | *Routing exceptions to EL2* on page D1-1564 |
| HCR_EL2.TTLB | T | *Traps to EL2 of Non-secure EL1 execution of TLB maintenance instructions* on page D1-1587 |
| HCR_EL2.{TSW, TPC, TPU} | T | *Traps to EL2 of Non-secure EL0 and EL1 execution of cache maintenance instructions* on page D1-1587 |
| HCR_EL2.TACR | T | *Traps to EL2 of Non-secure EL1 accesses to the Auxiliary Control Register* on page D1-1589 |
| HCR_EL2.TIDCP | T | *Traps to EL2 of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page D1-1589 |
| HCR_EL2.TSC | T | *Traps to EL2 of Non-secure EL1 execution of SMC instructions* on page D1-1590 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D1-34 Instruction disables and trap controls provided by EL2 (continued)**

| Control | Control type[a] | Description |
|---|---|---|
| HCR_EL2.{TID0, TID1, TID2, TID3} | T | *Traps to EL2 of Non-secure EL0 and EL1 accesses to the ID registers* on page D1-1591 |
| HCR_EL2.{TWI, TWE} | T | *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page D1-1594 |
| CPTR_EL2.TCPAC | T | *Trapping to EL2 of Non-secure EL1 accesses to the CPACR_EL1 or CPACR* on page D1-1595 |
| CPTR_EL2.TFP | T | *General trapping to EL2 of Non-secure accesses to the SIMD and floating-point registers* on page D1-1595 |
| CPTR_EL2.TTA | T | *Traps to EL2 of Non-secure System register accesses to the trace registers* on page D1-1596 |
| HSTR_EL2.{T0-T3, T5-T13, T15} | T | *General trapping to EL2 of Non-secure EL0 and EL1 accesses to System registers, from AArch32 state only* on page D1-1596 |
| MDCR_EL2.{TDRA, TDOSA, TDA} | T | *Traps to EL2 of Non-secure EL0 and EL1 System register accesses to debug registers* on page D1-1597 |
| CNTHCTL_EL2.{EL1PCEN, EL1PCTEN} | T | *Traps to EL2 of Non-secure EL0 and EL1 accesses to the Generic Timer registers* on page D1-1600 |
| MDCR_EL2.{TPM, TPMCR} | T | *Traps to EL2 of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page D1-1600 |

a. See Table D1-35.

**Table D1-35 Control types, for exceptions taken to EL1**

| Abbreviation | Type | See |
|---|---|---|
| D | Disable | *Instruction enables and instruction disables* on page D1-1575 |
| R | Routing control | *Routing exceptions to EL2* on page D1-1564 |
| T | Trap | *Trap controls* on page D1-1575 |

Also see the following for more general information about traps to EL2:

- *Register access instructions* on page D1-1576.
- For traps from an Exception level using AArch32:
  - *Instructions that fail their condition code check* on page G1-3856.
  - *Instructions that are UNPREDICTABLE* on page G1-3857.

### Traps to EL2 of Non-secure EL1 accesses to virtual memory control registers

HCR_EL2.{TRVM, TVM} trap Non-secure EL1 accesses to the virtual memory control registers to EL2:

**HCR_EL2.TRVM, for read accesses:**

**1** Non-secure EL1 reads of the virtual memory control registers are trapped to EL2.

**0** Non-secure EL1 reads of the virtual memory control registers are not trapped to EL2.

**HCR_EL2.TVM, for write access:**

**1** Non-secure EL1 writes to the virtual memory control registers are trapped to EL2.

**0** Non-secure writes to the virtual memory control registers are not trapped to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-36 shows the registers for which:

- Reads are trapped to EL2 when HCR_EL2.TRVM is 1.
- Writes are trapped to EL2 when HCR_EL2.TVM is 1.

Table D1-36 also shows how the exceptions are reported in ESR_EL2.

**Table D1-36 Register read and write accesses trapped when HCR_EL2.{TRVM, TVM} are 1**

| Traps from | Registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, CONTEXTIDR_EL1. | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18. |
| AArch32 state | SCTLR, TTBR0, TTBR1, TTBCR, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR. | Trapped MCR or MRC CP15 access, using EC value 0x03.<br>Trapped MCRR or MRRC CP15 access, using EC value 0x04. |

**Note**

EL2 provides a second stage of address translation, that a hypervisor can use to remap the address map defined by a Guest OS. In addition, a hypervisor can trap attempts by a Guest OS to write to the registers that control the Non-secure memory system. A hypervisor might use this trap as part of its virtualization of memory management.

### Disabling Non-secure state execution of HVC instructions

HCR_EL2.HCD disables Non-secure state execution of HVC instructions:

**1** HVC instructions are UNDEFINED at EL2 and Non-secure EL1. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

**0** HVC instruction execution is enabled at EL2 and Non-secure EL1.

**Note**

HVC instructions are always UNDEFINED at EL0.

HCR_EL2.HCD is only implemented if EL3 is not implemented. Otherwise, it is RES0.

Table D1-37 shows how the exceptions are reported in ESR_ELx.

**Table D1-37 Instruction disabled when HCR_EL2.HCD is 1**

| Taken from | Disabled instruction | Syndrome reporting in ESR_ELx |
|---|---|---|
| AArch64 state | HVC | Exception for an unknown reason, using EC value 0x00 |
| AArch32 state | HVC | |

### Traps to EL2 of Non-secure EL0 and EL1 execution of DC ZVA instructions

HCR_EL2.TDZ traps Non-secure EL0 and EL1 execution of DC ZVA instructions to EL2:

**1** Any attempt to execute a DC ZVA instruction at Non-secure EL0 or EL1 is trapped to EL2. Reading the DCZID_EL0 returns a value that indicates that DC ZVA instructions are not implemented.

**0** Non-secure EL0 and EL1 execution of DC ZVA instructions is not trapped to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-38 shows how the exceptions are reported in ESR_EL2.

**Table D1-38 Instruction trapped to EL1 when HCR_EL2.TDZ is 0**

| Traps from | Trapped instruction | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | DC ZVA | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | n/a | n/a |

### Traps to EL2 of Non-secure EL1 execution of TLB maintenance instructions

In the ARMv8-A architecture, the system instruction encoding space includes TLB maintenance instructions.

HCR_EL2.TTLB traps Non-secure EL1 execution of TLB maintenance instructions to EL2:

**1** Any attempt to execute a TLBI instruction at Non-secure EL1 is trapped to EL2.

**0** Non-secure EL1 execution of TLBI instructions is not trapped to EL2.

Table D1-39 shows the instructions that are trapped, and how the exceptions are reported in ESR_EL2.

**Table D1-39 Instructions trapped to EL2 when HCR_EL2.TTLB is 1**

| Traps from | Trapped instructions | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | TLBI VMALLE1IS, TLBI VAE1IS, TLBI ASIDE1IS, TLBI VAAE1IS, TLBI VALE1IS, TLBI VAALE1IS, TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, TLBI VAALE1 | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS, ITLBIALL, ITLBIMVA, ITLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL. | Trapped MCR or MRC CP15 access, using EC value 0x03 |

**Note**

These instructions are always UNDEFINED at EL0.

For more information about these instructions, see:

- *TLB maintenance instructions* on page D4-1839, for the AArch64 state instructions.
- *The scope of TLB maintenance instructions* on page G4-4069, for the AArch32 state instructions.

### Traps to EL2 of Non-secure EL0 and EL1 execution of cache maintenance instructions

HCR_EL2.{TSW, TPC, TPU} trap cache maintenance instructions to EL2. When one of these controls is 1, any attempt to execute a corresponding cache maintenance instruction at Non-secure EL1, or at Non-secure EL0 if permitted by SCTLR_EL1.UCI, is trapped to EL2.

**Table D1-40 Controls for trapping cache maintenance instructions to EL2**

| Trap control | Trapped instructions |
|---|---|
| HCR_EL2.TSW | Data or unified cache maintenance by set/way |
| HCR_EL2.TPC | Data or unified cache maintenance to point of coherency |
| HCR_EL2.TPU | Cache maintenance to point of unification |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For:

- HCR_EL2.TSW == 1, Table D1-41 shows the instructions that are trapped, and how the exceptions are reported in ESR_EL2.
- HCR_EL2.TPC == 1, Table D1-42 shows the instructions that are trapped, and how the exceptions are reported in ESR_EL2.
- HCR_EL2.TPU == 1, Table D1-43 shows the instructions that are trapped, and how the exceptions are reported in ESR_EL2.

**Table D1-41 Instructions trapped to EL2 when HCR_EL2.TSW is 1**

| Traps from | Trapped instructions | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | DC ISW, DC CSW, DC CISW | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | DCISW, DCCSW, DCCISW | Trapped MCR or MRC CP15 access, using EC value 0x03 |

**Note**

These instructions are always UNDEFINED at EL0.

**Table D1-42 Instructions trapped to EL2 when HCR_EL2.TPC is 1**

| Traps from | Trapped instructions | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | DC IVAC, DC CVAC, DC CIVAC | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | DCIMVAC, DCCIMVAC, DCCMVAC | Trapped MCR or MRC CP15 access, using EC value 0x03 |

**Note**

DC IVAC is always UNDEFINED at EL0 using AArch64.

DCIMVAC, DCCIMVAC, and DCCMVAC are always UNDEFINED at EL0 using AArch32.

**Table D1-43 Instructions trapped to EL2 when HCR_EL2.TPU is 1**

| Traps from | Trapped instructions | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | IC IVAU, IC IALLU, IC IALLUIS, DC CVAU | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAU | Trapped MCR or MRC CP15 access, using EC value 0x03 |

**Note**

IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.

ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always UNDEFINED at EL0 using AArch32.

For more information about these instructions, see:

- *Cache maintenance instructions, and data cache zero* on page C5-272 for the AArch64 instructions.
- *Cache maintenance instructions, functional group* on page G4-4169 for the AArch32 instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Traps to EL2 of Non-secure EL1 accesses to the Auxiliary Control Register

HCR_EL2.TACR traps Non-secure EL1 accesses to the Auxiliary Control Registers to EL2:

**1** Non-secure EL1 accesses to the Auxiliary Control Registers are trapped to EL2.

**0** Non-secure EL1 accesses to the Auxiliary Control Registers are not trapped to EL2.

Table D1-44 shows the registers for which accesses are trapped, and how the exceptions are reported in ESR_EL2.

**Table D1-44 Register accesses trapped to EL2 when HCR_EL2.TACR is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | ACTLR_EL1 | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18` |
| AArch32 state | ACTLR and, if implemented, the ACTLR2. | Trapped `MCR` or `MRC` CP15 access, using EC value `0x03` |

**Note**

- The ACTLR_EL1, ACTLR, and ACTLR2 are not accessible at EL0.
- The Auxiliary Control Registers are IMPLEMENTATION DEFINED registers that might implement global control bits for the PE.

### Traps to EL2 of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations

The lockdown, DMA, and TCM features of the ARMv8-A architecture are IMPLEMENTATION DEFINED. The architecture reserves the encodings of a number of System registers for control of these features.

HCR_EL2.TIDCP traps the execution of System register access instructions that access these registers, as follows:

**1** At Non-secure EL1, any attempt to execute a System register access instruction with a reserved register encoding is trapped to EL2.

At Non-secure EL0, it is IMPLEMENTATION DEFINED whether attempts to execute System register access instructions with reserved register encodings are:

- Trapped to EL2.
- UNDEFINED, and the PE takes an Undefined Instruction exception to EL1.

**0** Non-secure EL0 and EL1 System register access instructions with reserved register encodings are not trapped to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-45 shows the register encodings for which accesses are trapped, and how the exceptions are reported in ESR_EL2.

**Table D1-45 Encodings trapped to EL2 when HCR_EL2.TIDCP is 1**

| Traps from | Register encodings | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | Any access to any of the encodings described in *Reserved encodings for IMPLEMENTATION DEFINED registers* on page C5-288. | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | An access to any of the following encodings:<br>• CRn==c9, opc1=={0-7}, CRm=={c0-c2, c5-c8}, opc2=={0-7}. See *VMSAv8-32 CP15 c9 register summary* on page G4-4144.<br>• CRn==c10, opc1=={0-7}, CRm=={c0, c1, c4, c8}, opc2=={0-7}. See *VMSAv8-32 CP15 c10 register summary* on page G4-4145.<br>• CRn==c11, opc1=={0-7}, CRm=={c0-c8, c15}, opc2=={0-7}. See *VMSAv8-32 CP15 c11 register summary* on page G4-4145. | Trapped MCR or MRC CP15 access, using EC value 0x03 |

An implementation can also include IMPLEMENTATION DEFINED registers that provide additional controls, to give finer-grained control of the trapping of IMPLEMENTATION DEFINED features.

**Note**

- ARM expects the trapping of Non-secure EL0 accesses to these functions to EL2 to be unusual, and used only when the hypervisor is virtualizing EL0 operation. ARM strongly recommends that unless the hypervisor must virtualize EL0 operation, a Non-secure EL0 access to any of these functions generates an Undefined Instruction exception, as it would if the implementation did not include EL2. The PE then takes this exception to Non-secure EL1.
- The trapping of accesses to these registers from Non-secure EL1 is higher priority than Undefined Instruction exceptions.

### Traps to EL2 of Non-secure EL1 execution of SMC instructions

HCR_EL2.TSC traps Non-secure EL1 execution of SMC instructions to EL2:

**1** Any attempt to execute an SMC instruction at Non-secure EL1 is trapped to EL2, regardless of the value of SCR_EL3.SMD.

**0** Non-secure EL1 execution of SMC instructions is not trapped to EL2.

If EL3 is not implemented, HCR_EL2.TSC is RES0.

Table D1-46 shows how the exceptions are reported in ESR_EL2:

**Table D1-46 SMC Instruction trapped to EL2 when HCR_EL2.TSC is 1**

| Traps from | Trapped instruction | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | SMC | Trapped SMC instruction execution in AArch64 state, using EC value 0x17 |
| AArch32 state | *SMC* on page F6-2979 | Trapped SMC instruction execution in AArch32 state, using EC value 0x13 |

In AArch32 state, the ARMv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

For more information about SMC instructions, see *SMC* on page C6-718.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- This trap is implemented only if the implementation includes EL3.
- SMC instructions are UNDEFINED at EL0.
- HCR_EL2.TSC traps execution of the SMC instruction. It is not a routing control for the SMC exception. Trap exceptions and SMC exceptions have different preferred return addresses.

### Traps to EL2 of Non-secure EL0 and EL1 accesses to the ID registers

Other than the MIDR_EL1, MPIDR_EL1, and PMCR_EL0.N, the ID registers are divided into groups, with a trap control in the HCR_EL2 for each group.

**Table D1-47 ID register groups**

| Trap control | Register group |
|---|---|
| HCR_EL2.TID0 | *ID group 0, Primary device identification registers* on page D1-1592 |
| HCR_EL2.TID1 | *ID group 1, Implementation identification registers* on page D1-1592 |
| HCR_EL2.TID2 | *ID group 2, Cache identification registers* on page D1-1593 |
| HCR_EL2.TID3 | *ID group 3, Detailed feature identification registers* on page D1-1593 |

These controls trap register accesses to EL2, as follows:

**HCR_EL2.TID0**

When 1, any attempt at Non-secure EL0 or EL1 to read any register in ID group 0 is trapped to EL2.

**HCR_EL2.TID1**

When 1, any attempt at Non-secure EL1 to read any register in ID group 1 is trapped to EL2.

**HCR_EL2.TID2**

When 1, any attempt at Non-secure EL0 or EL1 to read any register in ID group 2, and any attempt at Non-secure EL0 or EL1 to write to the CSSELR or CSSELR_EL1, is trapped to EL2.

**HCR_EL2.TID3**

When 1, any attempt at Non-secure EL1 to read any register in ID group 3 is trapped to EL2.

For the MIDR_EL1 and MPIDR_EL1, and for PMCR_EL0.N, the architecture provides read/write aliases. The original register becomes accessible only from EL2 or Secure state, and a Non-secure EL0 or EL1 read of the original register returns the value of the read/write alias. This substitution is invisible to the EL0 or EL1 software reading the register.

**Table D1-48 ID register substitution**

| Register | Original | Alias, EL2 using AArch64 |
|---|---|---|
| Main ID | MIDR_EL1 | VPIDR_EL2 |
| Multiprocessor Affinity | MPIDR_EL1 | VMPIDR_EL2 |
| Performance Monitors Control Register | PMCR_EL0.N | MDCR_EL2.HPMN |

Reads of the MIDR_EL1, MPIDR_EL1 or PMCR_EL0.N from EL2 or Secure state are unchanged by the implementation of EL2, and access the physical registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- If the optional Performance Monitors Extension is not implemented, MDCR_EL2.HPMN is RES0 and PMCR_EL0 is reserved.
- MDCR_EL2.HPMN also affects whether a Performance Monitors counter can be accessed from Non-secure EL0 or EL1. See the register description of MDCR_EL2 for more information.
- PMCR_EL0 contains other fields that identify the implementation. For more information about trapping accesses to the PMCR_EL0, see *Traps to EL2 of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page D1-1600.

### *ID group 0, Primary device identification registers*

In:

- AArch64 state, there are no ID group 0 registers.
- AArch32 state, these registers identify some top-level implementation choices.

Table D1-49 shows the registers that are in ID group 0 for traps to EL2, and how the exceptions are reported in ESR_EL2.

**Table D1-49 ID group 0 registers**

| Traps from | Group 0 registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | n/a | n/a |
| AArch32 state | FPSID | Trapped CP10 access, using EC value `0x08` |
| | JIDR | Trapped CP14 access, using EC value `0x05` |

**Note**

The FPSID is not accessible from EL0 using AArch32.

When the FPSID is accessible, a T32 or A32 `VMSR FPSID, <Rt>` instruction is permitted but is ignored. The execution of this `VMSR` instruction execution is not trapped by the ID group 0 trap.

### *ID group 1, Implementation identification registers*

These registers often provide coarse-grained identification mechanisms for implementation-specific features.

Table D1-50 shows the registers that are in ID group 1 for traps to EL2, and how the exceptions are reported in ESR_EL2.

**Table D1-50 ID group 1 registers**

| Traps from | Group 1 registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | REVIDR_EL1, AIDR_EL1 | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18` |
| AArch32 state | TCMTR, TLBTR, REVIDR, AIDR | Trapped `MCR` or `MRC` CP15 access, using EC value `0x03` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *ID group 2, Cache identification registers*

These registers describe and control the cache implementation.

Table D1-51 shows the registers that are in ID group 2 for traps to EL2, and how the exceptions are reported in ESR_EL2.

**Table D1-51 ID group 2 registers**

| Traps from | Group 2 registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | CTR_EL0, CCSIDR_EL1, CLIDR_EL1, CSSELR_EL1 | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18` |
| AArch32 state | CTR, CCSIDR, CLIDR, CSSELR | Trapped `MCR` or `MRC` CP15 access, using EC value `0x03` |

#### *ID group 3, Detailed feature identification registers*

These registers provide detailed information about the features of the implementation.

**Note**

In AArch32 state, these registers are called the CPUID registers. There is no requirement for this trap to apply to those registers that the CPUID Identification Scheme defines as reserved. See *The CPUID identification scheme* on page G4-4163.

Table D1-52 shows the registers that are in ID group 3 for traps to EL2, and how the exceptions are reported in ESR_EL2.

**Table D1-52 ID group 3 registers**

| Traps from | Group 3 registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | ID_PFR0_EL1, ID_PFR1_EL1, ID_DFR0_EL1.<br>ID_AFR0_EL1, ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, and ID_MMFR4_EL1, except that if ID_MMFR4_EL1 is implemented as RAZ/WI then it is IMPLEMENTATION DEFINED whether reads of the register are trapped.<br>ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, ID_ISAR5_EL1.<br>MVFR0_EL1, MVFR1_EL1, MVFR2_EL1.<br>ID_AA64PFR0_EL1, ID_AA64PFR1_EL1. ID_AA64DFR0_EL1, ID_AA64DFR1_EL1. ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1. ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1. ID_AA64AFR0_EL1, ID_AA64AFR1_EL1.<br>It is IMPLEMENTATION DEFINED whether HCR_EL2.TID3 traps `MRS` accesses to registers in the following range that are not already mentioned in this table:<br>• `Op0` == 3, `CRn` == c0, `op1` == 0, `CRm` == {c2-c7}, `op2` == {0-7}. | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D1-52 ID group 3 registers (continued)**

| Traps from | Group 3 registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch32 state | MVFR0, MVFR1, MVFR2. | Trapped CP10 access, using EC value 0x08 |
| | ID_PFR0, ID_PFR1, ID_DFR0, ID_AFR0.<br>ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3 and, ID_MMFR4, except that if ID_MMFR4 is implemented as RAZ/WI then it is IMPLEMENTATION DEFINED whether reads of the register are trapped.<br>ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5.<br>Any MRC access to any of the following CP15 encodings:<br>• CRn == c0, opc1 == 0, CRm == {c3-c7}, opc2 == {0, 1}.<br>• CRn == c0, opc1 == 0, CRm == c3, opc2 == 2.<br>• CRn == c0, opc1 == 0, CRm == c5, opc2 == {4, 5}.<br>It is IMPLEMENTATION DEFINED whether HCR_EL2.TID3 traps MRC accesses to CP15 encodings in the following range that are not already mentioned in this table:<br>• CRn == c0, opc1 == 0, CRm == {c2-c7}, opc2 == {0-7}. | Trapped MCR or MRC CP15 access, using EC value 0x03 |

### Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions

HCR_EL2.{TWE, TWI} trap Non-secure EL0 and EL1 execution of WFE and WFI instructions to EL2:

**HCR_EL2.TWE**

**1** Any attempt to execute a WFE instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state.

**0** Non-secure EL0 or EL1 execution of WFE instructions is not trapped to EL2.

**HCR_EL2.TWI**

**1** Any attempt to execute a WFI instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state.

**0** Non-secure EL0 or EL1 execution of WFI instructions is not trapped to EL2.

Table D1-53 shows how the exceptions are reported in ESR_EL2.

**Table D1-53 Instructions trapped to EL2 when HCR_EL2.{TWE, TWI} are 1**

| Trap control | Traps from | Trapped instructions | Syndrome reporting in ESR_EL2 |
|---|---|---|---|
| HCR_EL2.TWE | Both Execution states | WFE | Trapped WFI or WFE instruction, using EC value 0x01 |
| HCR_EL2.TWI | | WFI | |

In AArch32 state, the attempted execution of a conditional WFE or WFI instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait for Event mechanism and Send event* on page D1-1611.
- *Wait For Interrupt* on page D1-1614.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Trapping to EL2 of Non-secure EL1 accesses to the CPACR_EL1 or CPACR

CPTR_EL2.TCPAC traps Non-secure EL1 accesses to the CPACR_EL1 or CPACR to EL2:

**1** Non-secure EL1 accesses to the CPACR_EL1 or CPACR are trapped to EL2.

**0** Non-secure EL1 accesses to the CPACR_EL1 or CPACR are not trapped to EL2.

Table D1-54 shows how the exceptions are reported in ESR_EL2.

**Table D1-54 Register accesses trapped to EL2 when CPTR_EL2.TCPAC is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | CPACR_EL1 | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18` |
| AArch32 state | CPACR | Trapped `MCR` or `MRC` CP15 access, using EC value `0x03` |

**Note**

- The CPACR_EL1 or CPACR is not accessible at EL0.
- In ARMv7 and earlier versions of the ARM architecture, one function of the CPACR is as an ID register that identifies what coprocessor functionality is implemented. Legacy software might use this identification mechanism. A hypervisor can use this trap to emulate this mechanism.

### General trapping to EL2 of Non-secure accesses to the SIMD and floating-point registers

CPTR_EL2.TFP traps Non-secure accesses to SIMD and floating-point registers to EL2:

**1** Any attempt at EL2, or Non-secure EL0 or EL1, to execute an instruction that accesses the SIMD or floating-point registers is trapped to EL2.

**0** Non-secure execution of instructions that access the SIMD or floating-point registers is not trapped to EL2.

Table D1-55 shows the registers for which accesses are trapped, and how the exceptions are reported in ESR_EL2.

**Table D1-55 Register accesses trapped to EL2 when CPTR_EL2.TFP is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | FPCR, FPSR, FPEXC32_EL2, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-S31 registers. See *The SIMD and floating-point registers, V0-V31* on page D1-1524. | Trapped access to a SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP, using EC value `0x07` |
| AArch32 state | FPSID, MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers. See *Advanced SIMD and floating-point System registers* on page G1-3843. | Trapped access to a SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP, using EC value `0x07`[a] |

a. Permitted `VMSR` accesses to the FPSID are ignored, but for the purposes of this trap the architecture defines a `VMSR` access to the FPSID from EL1 or higher as an access to a SIMD and floating-point register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Traps to EL2 of Non-secure System register accesses to the trace registers

CPTR_EL2.TTA traps System register accesses to the trace registers to EL2. These traps are from Non-secure state, so are from all of:

- EL2.
- Non-secure EL0 and EL1.

When CPTR_EL2.TTA is:

**1** Non-secure System register accesses to the trace registers are trapped to EL2.

**0** Non-secure System register accesses to the trace registers are not trapped to EL2.

**Note**

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED. A resulting Undefined Instruction exception is higher priority than a CPTR_EL2.TTA Trap exception.
- EL2 does not provide traps on trace register accesses through the Memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Register access instructions* on page D1-1576.

Table D1-56 shows the registers for which accesses are trapped to EL2 when CPTR_EL2.TTA is 1, and how the exceptions are reported in ESR_EL2.

**Table D1-56 Register accesses trapped to EL2 when CPTR_EL2.TTA is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | All implemented trace registers | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18`. |
| AArch32 state | All implemented trace registers | For accesses using:<br>• `MCR` or `MRC` instructions, trapped `MCR` or `MRC` CP14 access, using EC value `0x05`.<br>• `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` CP14 access, using EC value `0x0C`. |

### General trapping to EL2 of Non-secure EL0 and EL1 accesses to System registers, from AArch32 state only

HSTR_EL2.{T0-T3, T5-T13, T15} trap accesses to the CP15 System registers, by the accessed primary CP15 register number, {c0-c3, c5-c13, c15}. These traps are from AArch32 state only. They are from both:

- Non-secure EL1 using AArch32.
- Non-secure EL0 using AArch32.

When an HSTR_EL2.Tx trap control is:

**1** Any AArch32 state Non-secure EL1 or EL0 access to the corresponding register is trapped to EL2.

**0** AArch32 state Non-secure EL1 or EL0 accesses to the corresponding register are not trapped to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-57 shows the accesses that are trapped, and how the exceptions are reported in ESR_EL2.

**Table D1-57 Accesses trapped to EL2 when an HSTR_EL2.Tx trap is enabled**

| Traps from | Trapped accesses | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | n/a | n/a |
| AArch32 state | MCR and MRC instructions, where CRn in the instruction corresponds to the trapped primary CP15 register | Trapped MCR or MRC CP15 access, using EC value 0x03 |
| | MCRR and MRRC instructions, where CRm in the instruction corresponds to the trapped primary CP15 register | Trapped MCRR or MRRC CP15 access, using EC value 0x04 |

**Note**

HSTR_EL2[4, 14] is reserved, RES0, although the Generic Timer control registers are implemented in CP15 c14. EL2 does not provide a trap on accesses to the Generic Timer CP15 registers.

#### *CP15 register with IMPLEMENTATION DEFINED access permission from EL0*

For a trapped primary CP15 register, if it is IMPLEMENTATION DEFINED whether, when the corresponding trap control is 0, an access from Non-secure User mode is UNDEFINED, then when the corresponding trap control is 1, it is IMPLEMENTATION DEFINED whether an access from Non-secure EL0 using AArch32 generates:

- A Trap exception taken to EL2.
- An Undefined Instruction exception taken to Non-secure EL1.

If the instruction generates an Undefined Instruction exception taken to Non-secure EL1, and Non-secure EL1 is using AArch64, the exception is reported in ESR_EL1 as an exception for an unknown reason, using EC value 0x00.

**Note**

ARM expects that trapping of Non-secure EL0 accesses to EL2 will be unusual, and used only when the hypervisor must virtualize EL0 operation. ARM recommends that, whenever possible, Non-secure EL0 accesses to the System registers behave as they would if the implementation did not include EL2.This means that, if the architecture does not support the Non-secure EL0 access, that access generates an Undefined Instruction exception that is taken to Non-secure EL1.

### Traps to EL2 of Non-secure EL0 and EL1 System register accesses to debug registers

MDCR_EL2.{TDRA, TDOSA, TDA} trap Non-secure System register accesses to the debug registers to EL2, as follows:

- MDCR_EL2.(TDRA, TDA} trap Non-secure EL0 and EL1 accesses.
- MDCR_EL2.TDOSA traps Non-secure EL1 accesses.

**Note**

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

System register accesses to the debug registers can have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2. See *Register access instructions* on page D1-1576.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-58 shows the subsections that list the accesses trapped. The subsections describe how the traps are reported in ESR_EL2.

**Table D1-58 Traps of Non-secure EL0 and EL1 accesses to debug registers**

| Trap control | Subsection |
|---|---|
| MDCR_EL2.TDRA | *Traps to EL2 of Non-secure EL0 and EL1 System register accesses to debug registers* on page D1-1597 |
| MDCR_EL2.TDOSA | *Trapping System register accesses to powerdown debug registers to EL2* |
| MDCR_EL2.TDA | *Trapping general System register accesses to debug registers to EL2* on page D1-1599 |

#### *Trapping System register accesses to Debug ROM registers to EL2*

MDCR_EL2.TDRA traps Non-secure EL0 and EL1 System register accesses to the Debug ROM registers to EL2:

**1** Non-secure EL0 and EL1 System register accesses to the Debug ROM registers are trapped to EL2.

**0** Non-secure EL0 and EL1 System register accesses to the Debug ROM registers are not trapped to EL2.

This trap applies to Non-secure EL0 only if it is using AArch32.

Table D1-59 shows the register accesses that are trapped, and how the exceptions are reported in ESR_EL2.

**Table D1-59 Register accesses trapped to EL2 when MDCR_EL2.TDRA is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | MDRAR_EL1 | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18`. |
| AArch32 state | DBGDRAR, DBGDSAR | For accesses using:<br>• `MCR` or `MRC` instructions, trapped `MCR` or `MRC` CP14 access, using EC value `0x05`.<br>• `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` CP14 access, using EC value `0x0C`. |

If MDCR_EL2.TDE or HCR_EL2.TGE is 1, behavior is as if MDCR_EL2.TDRA is 1 other than for the purpose of a direct read.

#### *Trapping System register accesses to powerdown debug registers to EL2*

MDCR_EL2.TDOSA traps Non-secure EL1 System register accesses to the powerdown debug registers to EL2:

**1** Non-secure EL1 System register accesses to the powerdown debug registers are trapped to EL2.

**0** Non-secure EL1 System register accesses to the powerdown debug registers are not trapped to EL2.

Table D1-60 shows the register accesses that are trapped, and how the exceptions are reported in ESR_EL2.

**Table D1-60 Register accesses trapped to EL2 when MDCR_EL2.TDOSA is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, DBGPRCR_EL1.<br>Any IMPLEMENTATION DEFINED integration registers.<br>Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by MDCR_EL2.TDOSA | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18`. |
| AArch32 state | DBGOSLSR, DBGOSLAR, DBGOSDLR, DBGPRCR.<br>Any IMPLEMENTATION DEFINED integration registers.<br>Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by HDCR.TDOSA | Trapped `MCR` or `MRC` CP14 access, using EC value `0x05`. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

These registers are not accessible at EL0.

If MDCR_EL2.TDE or HCR_EL2.TGE is 1, behavior is as if MDCR_EL2.TDOSA is 1 other than for the purpose of a direct read.

### *Trapping general System register accesses to debug registers to EL2*

MDCR_EL2.TDA traps Non-secure EL0 and EL1 System register accesses to those debug System registers that are not mentioned in either of the following:

- *Traps to EL2 of Non-secure EL0 and EL1 System register accesses to debug registers* on page D1-1597.
- *Trapping System register accesses to powerdown debug registers to EL2* on page D1-1598.

This means that MDCR_EL2.TDA traps Non-secure EL0 and EL1 System register accesses to all debug System registers to EL2, except the following:

- Any access from:
  - AArch64 state to the MDRAR_EL1.
  - AArch32 state to the DBGDRAR or DBGDSAR.

  MDCR_EL2.TDRA traps these accesses.
- Any access from:
  - AArch64 state to the OSLAR_EL1, OSLSR_EL1, OSDLR_EL1 or DBGPRCR_EL1.
  - AArch32 state to the DBGOSLSR, DBGOSLAR, OSDLR_EL1 or DBGPRCR.

  MDCR_EL2.TDOSA traps these accesses.

MDCR_EL2.TDA does not trap accesses to the DBGDTRRX_EL0, DBGDTRTX_EL0, or DBGDTR_EL0 when the PE is in Debug state.

When MDCR_EL2.TDA is:

**1** Non-secure EL0 or EL1 System register accesses to any of the registers shown in Table D1-61 are trapped to EL2.

**0** Non-secure EL0 or EL1 System register accesses to the registers shown in Table D1-61 are not trapped to EL2.

Table D1-61 shows how the exceptions are reported in ESR_EL2.

**Table D1-61 Accesses trapped to EL2 when MDCR_EL2.TDA is 1**

| Traps from | Trapped accesses | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | Accesses to the MDCCSR_EL0, MDCCINT_EL1, DBGDTR_EL0, DBGDTRRX_EL0, DBGDTRTX_EL0, OSDTRRX_EL1, MDSCR_EL1, OSDTRTX_EL1, OSECCR_EL1, DBGBVR<n>_EL1, DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_EL1, DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, and DBGAUTHSTATUS_EL1. | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18` |
| AArch32 state | Accesses to the DBGDIDR, DBGDSCRint, DBGDCCINT, DBGDTRRXint, DBGDTRTXint, DBGWFAR, DBGVCR, DBGDSCRext, DBGDTRTXext, DBGDTRRXext, DBGBVR<n>, DBGBCR<n>, DBGBXVR<n>, DBGWCR<n>, DBGWVR<n>, DBGCLAIMSET, DBGCLAIMCLR, DBGAUTHSTATUS, DBGDEVID, DBGDEVID1, DBGDEVID2, and DBGOSECCR. | For accesses using `MCR` or `MRC` instructions, trapped `MCR` or `MRC` CP14 access, using EC value `0x05` |
| | `STC` accesses to DBGDTRRXint.<br>`LDC` accesses to DBGDTRTXint. | Trapped `LDC` or `STC` access to CP14, using EC value `0x06` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If MDCR_EL2.TDE or HCR_EL2.TGE is 1, behavior is as if MDCR_EL2.TDA is 1 other than for the purpose of a direct read.

### Traps to EL2 of Non-secure EL0 and EL1 accesses to the Generic Timer registers

CNTHCTL_EL2.{EL1PCEN, EL1PCTEN} trap Non-secure EL0 and EL1 accesses to the Generic Timer registers to EL2, as follows:

- CNTHCTL_EL2.EL1PCEN traps Non-secure EL0 and EL1 accesses to the physical timer registers.
- CNTHCTL_EL2.EL1PCTEN traps Non-secure EL0 and EL1 accesses to the physical counter register.

For each of these controls:

**1** Non-secure EL0 and EL1 accesses are not trapped to EL2.

**0** Non-secure EL0 and EL1 accesses are trapped to EL2.

Table D1-62 shows the registers for which accesses are trapped, and how the exceptions are reported in ESR_EL2.

**Table D1-62 Register accesses trapped to EL2 by CNTHCTL_EL2 trap controls**

| Traps from | Trap control | Registers | Syndrome reporting in ESR_EL2 |
|---|---|---|---|
| AArch64 state | EL1PCEN | CNTP_CTL_EL0, CNTP_CVAL_EL0, CNTP_TVAL_EL0 | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| | EL1PCTEN | CNTPCT_EL0 | |
| AArch32 state | EL1PCEN | CNTP_CTL, CNTP_CVAL, CNTP_TVAL | For accesses using:<br>• MCR or MRC instructions, trapped MCR or MRC CP15 access, using EC value 0x03<br>• MCRR or MRRC instructions, trapped MCRR or MRRC CP15 access, using EC value 0x04 |
| | EL1PCTEN | CNTPCT | Trapped MCRR or MRRC CP15 access, using EC value 0x04 |

### Traps to EL2 of Non-secure EL0 and EL1 accesses to Performance Monitors registers

MDCR_EL2.{TPM, TPMCR} trap Non-secure EL0 and EL1 accesses to the Performance Monitors registers to EL2:

**MDCR_EL2.TPM**

**1** Non-secure EL0 and EL1 accesses to all Performance Monitors registers are trapped to EL2.

**0** Non-secure EL0 and EL1 accesses to any Performance Monitors register is not trapped to EL2.

**MDCR_EL2.TPMCR**

**1** Non-secure EL0 and EL1 accesses to the Performance Monitors Control Registers are trapped to EL2.

**0** Non-secure EL0 and EL1 accesses to the Performance Monitors Control Registers are not trapped to EL2.

**Note**

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For:

- MDCR_EL2.TPM == 1, Table D1-63 shows the registers for which accesses are trapped, and how the exceptions are reported in ESR_EL2.
- MDCR_EL2.TPMCR == 1, Table D1-64 shows the registers for which accesses are trapped, and how the exceptions are reported in ESR_EL2.

**Table D1-63 Register accesses trapped to EL2 when MDCR_EL2.TPM is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | PMCR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMOVSCLR_EL0, PMSWINC_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0, PMXEVTYPER_EL0, PMXEVCNTR_EL0, PMUSERENR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, PMOVSSET_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMCCFILTR_EL0. | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | PMCR, PMCNTENSET, PMCNTENCLR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMUSERENR, PMINTENSET, PMINTENCLR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR. | For accesses using:<br>• MCR or MRC instructions, trapped MCR or MRC CP15 access, using EC value 0x03<br>• MCRR or MRRC instructions, trapped MCRR or MRRC CP15 access, using EC value 0x04 |

**Table D1-64 Register accesses trapped to EL2 when MDCR_EL2.TPMCR is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL2 |
|---|---|---|
| AArch64 state | PMCR_EL0 | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | PMCR | Trapped MCR or MRC CP15 access, using EC value 0x03 |

**Note**

MDCR_EL2.HPMN affects whether a counter can be accessed from Non-secure EL0 or EL1. See the register description of MDCR_EL2 for more information.

### D1.15.4 EL3 configurable controls

These controls are in _EL3 System registers. The resulting exceptions might be taken from either Execution state. SPSR_EL3.M[4] indicates which Execution state the exception was taken from.

Table D1-65 shows the _EL3 System registers that contain these controls.

**Table D1-65 _EL3 registers that contain instruction enables and disables, and trap controls**

| Register description | Register name |
|---|---|
| Secure Configuration Register | SCR_EL3 |
| Architectural Feature Trap Register, EL3 | CPTR_EL3 |
| Monitor Debug Configuration Register, EL3 | MDCR_EL3 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-66 summarizes the controls.

**Table D1-66 Instruction enables and disables, and trap controls, provided by EL3**

| Control | Control type[a] | Description |
|---|---|---|
| SCR_EL3.{TWE, TWI} | T | *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-1603 |
| SCR_EL3.ST | T | *Traps to EL3 of Secure EL1 accesses to the Counter-timer Physical Secure timer registers* on page D1-1604 |
| SCR_EL3.HCE | E | *Enabling EL3, EL2, and Non-secure EL1 execution of HVC instructions* on page D1-1604 |
| SCR_EL3.SMD | D | *Disabling EL3, EL2, and EL1 execution of SMC instructions* on page D1-1605 |
| CPTR_EL3.TCPAC | T | *Trapping to EL3 of EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR* on page D1-1605 |
| CPTR_EL3.TTA | T | *Traps to EL3 of all System register accesses to the trace registers* on page D1-1606 |
| CPTR_EL3.TFP | T | *Traps to EL3 of all accesses to the SIMD and floating-point registers* on page D1-1606 |
| MDCR_EL3.{TDOSA, TDA} | T | *Traps to EL3 of EL2, EL1, and EL0 System register accesses to debug registers* on page D1-1607 |
| MDCR_EL3.TPM | T | *Traps to EL3 of EL2, EL1, and EL0 accesses to Performance Monitors registers* on page D1-1609 |

a. See Table D1-67.

**Table D1-67 Control types, for exceptions taken to EL1**

| Abbreviation | Type | See |
|---|---|---|
| D | Disable | *Instruction enables and instruction disables* on page D1-1575 |
| E | Enable | *Instruction enables and instruction disables* on page D1-1575 |
| T | Trap | *Trap controls* on page D1-1575 |

Also see the following for more general information about traps to EL3:

- *Register access instructions* on page D1-1576.
- *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32*.

### Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32

If EL1 is using AArch32, all of the following are trapped to EL3:

- Secure EL1 reads and writes to any of the SCR, NSACR, MVBAR or SDCR.
- Any attempt at Secure EL1 to execute any of the following:
  - `ATS12NSOxx` instructions.
  - `SRS` instructions that use the R13_mon banked register.
  - `MRS` or `MSR` instructions that access any of the SPSR_mon, R13_mon or R14_mon banked registers.

In addition, if EL1 is using AArch32:

- Secure EL1 write accesses to the CNTFRQ register are UNDEFINED. They are not trapped to EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Any attempt at Secure EL1 to change the mode to Monitor mode, by using a CPS or an MSR instruction, or by performing an exception return, is treated as an illegal change of the CPSR.M field. See *Illegal changes to PSTATE.M* on page G1-3771.

Table D1-68 shows the accesses that are trapped to EL3, and how the exceptions are reported in ESR_EL3.

**Table D1-68 Accesses trapped to EL3 from Secure EL1 using AArch32**

| Taken from | Trapped instructions, or trapped accesses | Syndrome reporting in ESR_EL3 |
|---|---|---|
| Secure EL1 using AArch32 | Reads and writes to any of the SCR, NSACR, MVBAR or SDCR | Trapped MCR or MRC CP15 access, using EC value 0x03 |
| | ATS12NSOxx instructions | |
| | SRS instructions that use the R13_mon banked register | Exception for an unknown reason, using EC value 0x00 |
| | MRS or MSR instructions that accesses any of the SPSR_mon, R13_mon or R14_mon banked registers | |

**Note**

- Reads of the NSACR from either Non-secure EL1 using AArch32 or Non-secure EL2 using AArch32 return the value 0x00000C00. See *Restricted access System registers* on page G4-4124.
- These operations are not available at EL0.

### Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions

SCR_EL3.{TWE, TWI} trap EL2, EL1, and EL0 execution of WFE and WFI instructions to EL3:

**SCR_EL3.TWE**

**1** Any attempt to execute a WFE instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state.

**0** EL2, EL1, and EL0 execution of WFE instructions is not trapped to EL3.

**SCR_EL3.TWI**

**1** Any attempt to execute a WFI instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state.

**0** EL2, EL1, and EL0 execution of WFI instructions is not trapped.

For EL0 and EL1, these traps apply to WFE and WFI execution in both Security states.

Table D1-69 shows how the exceptions are reported in ESR_EL3.

**Table D1-69 Instructions trapped to EL3 when SCR_EL3.{TWE, TWI} are 1**

| Trap control | Traps from | Trapped instructions | Syndrome reporting in ESR_EL3 |
|---|---|---|---|
| SCR_EL3.TWE | Both Execution states | WFE | Trapped WFI or WFE instruction, using EC value 0x01 |
| SCR_EL3.TWI | | WFI | |

In AArch32 state, the attempted execution of a conditional WFE or WFI instruction is only trapped if the instruction passes its condition code check.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait for Event mechanism and Send event* on page D1-1611.
- *Wait For Interrupt* on page D1-1614.

### Traps to EL3 of Secure EL1 accesses to the Counter-timer Physical Secure timer registers

SCR_EL3.ST traps Secure EL1 accesses to the Counter-timer Physical Secure timer registers to EL3:

**1** Secure EL1 accesses to the Counter-timer Physical Secure timer registers are not trapped to EL3.

**0** Secure EL1 accesses to the Counter-timer Physical Secure timer registers are trapped to EL3.

**Note**

Accesses to the Counter-timer Physical Secure timer registers are always enabled at EL3.

Table D1-70 shows the registers for which accesses are trapped to EL3, and how the exceptions are reported in ESR_EL3.

**Table D1-70 Register accesses trapped to EL3 when SCR_EL3.ST is 0**

| Traps from | Registers | Syndrome reporting in ESR_EL3 |
|---|---|---|
| AArch64 state | CNTPS_TVAL_EL1<br>CNTPS_CTL_EL1<br>CNTPS_CVAL_EL1 | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | n/a | n/a |

**Note**

These registers are not accessible at EL0.

### Enabling EL3, EL2, and Non-secure EL1 execution of HVC instructions

SCR_EL3.HCE enables HVC instruction execution at EL1 and above:

**1** HVC instruction execution is enabled at EL1 and above.

**0** HVC instructions are UNDEFINED at EL1, EL2, and EL3. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

For EL1, this enable control applies to HVC instructions in Non-secure state only.

If EL2 is not implemented, this bit is RES0.

**Note**

HVC instructions are always UNDEFINED at EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-71 shows how the exceptions are reported in ESR_ELx.

**Table D1-71 Instruction disabled when SCR_EL3.HCE is 0**

| Taken from | Disabled instruction | Syndrome reporting in ESR_ELx |
|---|---|---|
| AArch64 state | HVC | Exception for an unknown reason, using EC value 0x00 |
| AArch32 state | | |

### Disabling EL3, EL2, and EL1 execution of SMC instructions

SCR_EL3.SMD disables SMC instruction execution at EL1 and above:

**1** SMC instructions are UNDEFINED at EL1 and above. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

**0** SMC instruction execution is enabled at EL1 and above.

For EL1, this disable control applies to SMC instructions in both Security states.

**Note**

SMC instructions are always UNDEFINED at EL0.

Table D1-72 shows how the exceptions are reported in ESR_ELx.

**Table D1-72 Exceptions generated when SCR_EL3.SMD is 1**

| Taken from | Disabled Instruction | Syndrome reporting in ESR_ELx |
|---|---|---|
| AArch64 state | SMC | Exception for an unknown reason, using EC value 0x00 |
| AArch32 state | | |

**Note**

If HCR_EL2.TSC or HCR.TSC traps attempted EL1 execution of SMC instructions to EL2, that trap has priority over this disable.

### Trapping to EL3 of EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR

CPTR_EL3.TCPAC traps all of the following to EL3:

- EL2 accesses to the CPTR_EL2 or HCPTR.
- EL2 and EL1 accesses to the CPACR_EL1 or CPACR.

When CPTR_EL3.TCPAC is:

**1** EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR, are trapped to EL3.

**0** EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR, are not trapped to EL3.

For EL1, this trap control applies to accesses from both Security states.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-73 shows how the exceptions are reported in ESR_EL3.

**Table D1-73 Register accesses trapped to EL3 when CPTR_EL3.TCPAC is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL3 |
|---|---|---|
| AArch64 state | CPTR_EL2<br>CPACR_EL1 | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18` |
| AArch32 state | HCPTR<br>CPACR | Trapped `MCR` or `MRC` CP15 access, using EC value `0x03` |

### Traps to EL3 of all System register accesses to the trace registers

CPTR_EL3.TTA traps System register accesses to the trace registers, from all Exception levels, to EL3:

**1** All System register accesses to the trace registers are trapped to EL3.

**0** System register accesses to the trace registers are not trapped to EL3.

**Note**

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED. A resulting Undefined Instruction exception is higher priority than a CPTR_EL3.TTA Trap exception.
- EL3 does not provide traps on trace register accesses through the Memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Register access instructions* on page D1-1576.

For EL0 and EL1, this trap control applies to accesses from both Security states.

Table D1-74 shows the registers for which accesses are trapped to EL3, and how the exceptions are reported in ESR_EL3.

**Table D1-74 Register accesses trapped to EL3 when CPTR_EL3.TTA is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL3 |
|---|---|---|
| AArch64 state | All implemented trace registers | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18` |
| AArch32 state | All implemented trace registers | For accesses using:<br>• `MCR` or `MRC` instructions, trapped `MCR` or `MRC` CP14 access, using EC value `0x05`.<br>• `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` CP14 access, using EC value `0x0C`. |

### Traps to EL3 of all accesses to the SIMD and floating-point registers

CPTR_EL3.TFP traps all accesses to SIMD and floating-point registers, from all Exception levels, to EL3:

**1** Any attempt at any Exception level to execute an instruction that accesses the SIMD or floating-point registers is trapped to EL3

**0** Execution of instructions that access the SIMD or floating-point registers is not trapped to EL3.

For EL0 and EL1, this trap control applies to accesses from both Security states.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-75 shows the registers for which accesses are trapped to EL3, and how the exceptions are reported in ESR_EL3.

**Table D1-75 Register accesses trapped to EL3 when CPTR_EL3.TFP is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL3 |
|---|---|---|
| AArch64 state | FPCR, FPSR, FPEXC32_EL2, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-S31 registers. See *The SIMD and floating-point registers, V0-V31* on page D1-1524. | Trapped access to a SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP, using EC value `0x07` |
| AArch32 state | FPSID, MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers. See *Advanced SIMD and floating-point System registers* on page G1-3843. | Trapped access to a SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP, using EC value `0x07`[a] |

a. Permitted `VMSR` accesses to the FPSID are ignored, but for the purposes of this trap the architecture defines a `VMSR` access to the FPSID from EL1 or higher is an access to a SIMD and floating-point register.

**Note**

- FPEXC32_EL2 is not accessible from EL0 using AArch64.
- FPSID, MVFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

## Traps to EL3 of EL2, EL1, and EL0 System register accesses to debug registers

MDCR_EL3.{TDOSA, TDA} trap EL2, EL1, and EL0 System register accesses to the debug registers to EL3, from both Security states.

**Note**

EL3 does not provide traps on debug register accesses through the Memory-mapped or External debug interfaces.

System register accesses to the debug registers can have side-effects. When a System register access is trapped to EL3, no side-effects occur before the exception is taken to EL3. See *Register access instructions* on page D1-1576.

Table D1-76 shows the subsections that list the accesses trapped.

**Table D1-76 Traps of EL2, EL1, and EL0 accesses to debug registers**

| Trap control | Subsection |
|---|---|
| MDCR_EL3.TDOSA | *Trapping System register accesses to powerdown debug registers to EL3* |
| MDCR_EL3.TDA | *Trapping general System register accesses to debug registers to EL3* on page D1-1608 |

### *Trapping System register accesses to powerdown debug registers to EL3*

MDCR_EL3.TDOSA traps EL2 and EL1 accesses to the powerdown debug registers to EL3:

**1** EL2 and EL1 System register accesses to the powerdown debug registers are trapped to EL3.

**0** EL2 and EL1 System register accesses to the powerdown debug registers are not trapped to EL3.

For EL1, this trap control applies to accesses from both Security states.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-77 shows the registers for which accesses are trapped, and how the exceptions are reported in ESR_EL3.

**Table D1-77 Register accesses trapped to EL3 when MDCR_EL3.TDOSA is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL3 |
|---|---|---|
| AArch64 state | OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, DBGPRCR_EL1.<br>Any IMPLEMENTATION DEFINED integration registers.<br>Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by MDCR_EL3.TDOSA. | Trapped AArch64 `MSR`, `MRS`, or system instruction, using EC value `0x18`. |
| AArch32 state | DBGOSLSR, DBGOSLAR, DBGOSDLR, DBGPRCR. | For accesses using:<br>• `MCR` or `MRC` instructions, trapped `MCR` or `MRC` CP14 access, using EC value `0x05`.<br>• `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` CP14 access, using EC value `0x0C`. |

**Note**

These registers are not accessible at EL0.

### *Trapping general System register accesses to debug registers to EL3*

MDCR_EL3.TDA traps EL2, EL1, and EL0 System register accesses to the debug System registers that are not mentioned in *Trapping System register accesses to powerdown debug registers to EL3* on page D1-1607.

This means that MDCR_EL3.TDA traps EL2, EL1, and EL0 System register accesses to all debug System registers, except the following:

- Accesses from AArch64 state to the OSLAR_EL1, OSLSR_EL1, OSDLR_EL1 or DBGPRCR_EL1.
- Accesses from AArch32 state to the DBGOSLSR, DBGOSLAR, OSDLR_EL1 or DBGPRCR.

When MDCR_EL3.TDA is:

**1** EL2, EL1, and EL0 System register accesses to any of the registers shown in Table D1-78 on page D1-1609 are trapped to EL3.

**0** EL2, EL1, and EL0 System register accesses to the registers shown in Table D1-78 on page D1-1609 are not trapped to EL3.

For EL0 and EL1, this trap control applies to accesses from both Security states.

MDCR_EL3.TDA does not trap accesses to the DBGDTRRX_EL0, DBGDTRTX_EL0, or DBGDTR_EL0 when the PE is in Debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D1-78 shows the registers for which accesses are trapped, and how the exceptions are reported in ESR_EL3.

**Table D1-78 Accesses trapped to EL3 when MDCR_EL3.TDA is 1**

| Traps from | Trapped accesses | Syndrome reporting in ESR_EL3 |
|---|---|---|
| AArch64 state | Accesses to the MDCR_EL2, MDRAR_EL1, MDCCSR_EL0, MDCCINT_EL1, DBGDTR_EL0, DBGDTRRX_EL0, DBGDTRTX_EL0, OSDTRRX_EL1, MDSCR_EL1, OSDTRTX_EL1, OSECCR_EL1, DBGBVR<n>_EL1, DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_EL1, DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, and DBGAUTHSTATUS_EL1. | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | Accesses to the HDCR, DBGDRAR, DBGDSAR, DBGDIDR, DBGDSCRint, DBGDCCINT, DBGDTRRXint, DBGDTRTXint, DBGWFAR, DBGVCR, DBGDSCRext, DBGDTRTXext, DBGDTRRXext, DBGBVR<n>, DBGBCR<n>, DBGBXVR<n>, DBGWCR<n>, DBGWVR<n>, DBGCLAIMSET, DBGCLAIMCLR, DBGAUTHSTATUS, DBGDEVID, DBGDEVID1, DBGDEVID2 and DBGOSECCR. | For accesses using:<br>• MCR or MRC instructions, trapped MCR or MRC CP14 access, using EC value 0x05.<br>• MRRC instructions, trapped MCRR or MRRC CP14 access, using EC value 0x0C. |
| | STC accesses to DBGDTRRXint.<br>LDC accesses to DBGDTRTXint. | LDC or STC, trapped LDC or STC access to CP14, using EC value 0x06 |

### Traps to EL3 of EL2, EL1, and EL0 accesses to Performance Monitors registers

MDCR_EL3.TPM traps EL2, EL1, and EL0 accesses to the Performance Monitors registers to EL3:

**1** EL2, EL1, and EL0 System register accesses to all Performance Monitors registers are trapped to EL3.

**0** EL2, EL1, and EL0 System register accesses to Performance Monitors registers are not trapped to EL3.

For EL0 and EL1, this trap control applies to accesses from both Security states.

Table D1-79 shows the registers for which accesses are trapped, and how the exceptions are reported in ESR_EL3.

**Table D1-79 Register accesses trapped to EL3 when MDCR_EL3.TPM is 1**

| Traps from | Registers | Syndrome reporting in ESR_EL3 |
|---|---|---|
| AArch64 state | PMCR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMOVSCLR_EL0, PMSWINC_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0, PMXEVTYPER_EL0, PMXEVCNTR_EL0, PMUSERENR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, PMOVSSET_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMCCFILTR_EL0. | Trapped AArch64 MSR, MRS, or system instruction, using EC value 0x18 |
| AArch32 state | PMCR, PMCNTENSET, PMCNTENCLR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMUSERENR, PMINTENSET, PMINTENCLR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR. | For accesses using:<br>• MCR or MRC instructions, trapped MCR or MRC CP15 access, using EC value 0x03<br>• MCRR or MRRC instructions, trapped MCRR or MRRC CP15 access, using EC value 0x04 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.16 System calls

A system call is generated by the execution of an SVC, HVC, or SMC instruction:

- By default, the execution of an SVC instruction generates a Supervisor Call, a synchronous exception that targets EL1. This provides a mechanism for software executing at EL0 to make a call to an operating system or other software executing at EL1.
- In an implementation that includes EL2, the execution of an HVC instruction generates a Hypervisor Call, a synchronous exception that targets EL2 by default.

  The HVC instruction is UNDEFINED:
  — At EL0.
  — At EL1 in Secure state.

  **Note**

  Software executing at EL0 cannot directly generate a Hypervisor Call.
- In an implementation that includes EL3, by default the execution of an SMC instruction generates a Secure Monitor Call, a synchronous exception that targets EL3.

  The SMC instruction is UNDEFINED at EL0, meaning software executing at EL0 cannot directly generate a Secure Monitor Call.

The default behavior applies when the instruction is not UNDEFINED and both of the following are true:
- The instruction is executed at an Exception level that is the same as or lower than the target Exception level.
- The instruction is not trapped to a different Exception level.

If an SVC or HVC instruction is executed at an Exception level that is higher than the target Exception then it generates a synchronous exception that is taken to the current Exception level.

EL2 and EL3 can disable Hypervisor Call exceptions, see:
- *Disabling Non-secure state execution of HVC instructions* on page D1-1586.
- *Enabling EL3, EL2, and Non-secure EL1 execution of HVC instructions* on page D1-1604.

EL2 can trap use of the SMC instruction, see *Traps to EL2 of Non-secure EL1 execution of SMC instructions* on page D1-1590.

EL3 can disable Secure Monitor Call exceptions, see *Disabling EL3, EL2, and EL1 execution of SMC instructions* on page D1-1605.

### D1.16.1 Pseudocode description of system calls

The `AArch64.CallSupervisor()` pseudocode function performs an SVC call in AArch64 state.

The `AArch64.CallHypervisor()` pseudocode function performs an HVC call in AArch64 state.

The `AArch64.CallSecureMonitor()` pseudocode function performs an HVC call in AArch64 state.

The `AArch64.CallSupervisor()`, `AArch64.CallHypervisor()`, and `AArch64.CallSecureMonitor()` functions are described in Chapter J1 *ARMv8 Pseudocode*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.17 Mechanisms for entering a low-power state

The ARM architecture provides mechanisms that software can use to indicate that the PE can enter a low-power state, if it supports that state. The following sections describe those mechanisms:

- *Wait for Event mechanism and Send event*.
- *Wait For Interrupt* on page D1-1614.

### D1.17.1 Wait for Event mechanism and Send event

A PE can use the *Wait for Event* (WFE) mechanism to enter a low-power state, depending on the value of an Event Register for that PE. To enter the low-power state, the PE executes a Wait For Event instruction, `WFE`, and if the Event Register is clear, the PE can enter the low-power state.

If the PE does enter the low-power state, it remains in that low-power state until it receives a *WFE wake-up event*.

The architecture does not define the exact nature of the low-power state, except that the execution of a `WFE` instruction must not cause a loss of memory coherency.

WFE mechanism behavior depends on the interaction of all of the following, that are described in the subsections that follow:

- The Event Register for the PE. See subsection *The Event Register* on page D1-1612.
- The Wait For Event instruction, `WFE`. See subsection *The Wait For Event instruction* on page D1-1612.
- *WFE wake-up events*. See subsection *WFE wake-up events in AArch64 state* on page D1-1613
- The Send Event instructions, `SEV` and `SEVL` that can cause WFE wake-up events. See subsection *The Send Event instructions* on page D1-1613.

**Note**

Because the Wait for Event mechanism is associated with suspending execution on a PE for the purpose of power saving, ARM recommends that the Event Register is set only infrequently. However, software must only use the setting of the Event Register as a hint, and must not assume that any particular message is sent as a result of the setting of the Event Register.

Example D1-2 describes how a spinlock implementation might use the WFE mechanism to save energy.

**Example D1-2 Spinlock as an example of using Wait For Event and Send Event**

A multiprocessor operating system requires locking mechanisms to protect data structures from being accessed simultaneously by multiple PEs. These mechanisms prevent the data structures becoming inconsistent or corrupted if different PEs try to make conflicting changes. If a lock is busy, because a data structure is being used by one PE, it might not be practical for another PE to do anything except wait for the lock to be released. For example, if a PE is handling an interrupt from a device, it might need to add data received from the device to a queue. If another PE is removing data from the same queue, it will have locked the memory area that holds the queue. The first PE cannot add the new data until the queue is in a consistent state and the second PE has released the lock. The first PE cannot return from the interrupt handler until the data has been added to the queue, so it must wait.

Typically, a spin-lock mechanism is used in these circumstances:

- A PE requiring access to the protected data attempts to obtain the lock using single-copy atomic synchronization primitives such as the Load-Exclusive and Store-Exclusive operations described in *Synchronization and semaphores* on page B2-107.
- If the PE obtains the lock it performs its memory operation and then releases the lock.
- If the PE cannot obtain the lock, it reads the lock value repeatedly in a tight loop until the lock becomes available. When the lock becomes available, the PE again attempts to obtain it.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A spin-lock mechanism is not ideal for all situations:

- In a low-power system the tight read loop is undesirable because it uses energy to no effect.
- In a multiprocessor system the execution of spin-locks by multiple waiting PEs can degrade overall performance.

Using the Wait For Event and Send Event mechanism can improve the energy efficiency of a spinlock:

- A PE that fails to obtain a lock executes a `WFE` instruction to request entry to a low-power state, at the time when the exclusive monitor is set holding the address of the location holding the lock.
- When a PE releases a lock, the write to the lock location causes the exclusive monitor of any PE monitoring the lock location to be cleared. This clearing of the exclusive monitors generates a WFE wake-up event for each of those PEs. Then, these PEs can attempt to obtain the lock again.

For large systems, more advanced locking systems, such as ticket locks, can avoid unfairness caused by having multiple PEs simultaneously reading the lock. In such systems, the WFE mechanism can be used in a similar way to monitor the next ticket value.

---

### The Event Register

The Event Register is a single bit register for each PE. When set, an Event Register indicates that an event has occurred since the register was last cleared, that might require some action by the PE. Therefore, when the Event Register is set, the PE must not suspend operation on executing a `WFE` instruction.

The reset value of the Event Register is UNKNOWN.

The Event Register for a PE is set by any of the following:

- A Send Event instruction, `SEV`, executed by any PE in the system.
- A Send Event Local instruction, `SEVL`, executed by the PE.
- An exception return.
- The clearing of the global monitor for the PE.
- An event from a Generic Timer event stream, see *Event streams* on page D6-1901.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

The Event Register is cleared only by a Wait For Event instruction.

**Note**

Software cannot read or write the value of the Event Register directly.

---

### The Wait For Event instruction

The action of the Wait For Event instruction, `WFE`, depends on the state of the Event Register:

- If the Event Register is set, the instruction clears the register and completes immediately.
- If the Event Register is clear the PE can suspend execution and enter a low-power state. It remains in that state until the PE detects a WFE wake-up event, or earlier if the implementation chooses, or a until a reset. When the PE detects a WFE wake-up event, or earlier if chosen, the WFE instruction completes. If the wake-up event sets the Event Register, it is IMPLEMENTATION DEFINED whether on restarting execution, the Event Register is cleared.

The `WFE` instruction is available at all Exception levels. Attempts to enter a low-power state made by software executing at EL0, EL1, or EL2 might be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-1578.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page D1-1594.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-1603.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Software using the Wait For Event mechanism must tolerate spurious wake-up events, including multiple wake-ups.

### WFE wake-up events in AArch64 state

The following are *WFE wake-up events*:

- The execution of an SEV instruction on any PE in the multiprocessor system.
- Any physical SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE, that is not disabled by EDSCR.INTdis and:
  - Is marked as A in the tables in *Asynchronous exception masking* on page D1-1570, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
  - Is marked as B in the tables in *Asynchronous exception masking* on page D1-1570, if the value of the corresponding PSTATE.{A, I, F} mask bit is 0.
- In Non-secure EL1 or EL0, any virtual SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE, that is not disabled by EDSCR.INTdis and is marked as B in Table D1-18 on page D1-1573 in *Virtual interrupts* on page D1-1572, if the value of the corresponding PSTATE.{A, I, F} mask bit is 0.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed* see *Halting allowed and halting prohibited* on page H2-4821.

  See also *External Debug Request debug event* on page H3-4878.
- An event sent by the timer event stream for the PE. See *Event streams* on page D6-1901.
- An event caused by the clearing of the global monitor for the PE.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

Not all of these wake-up events set the Event Register.

**Note**

The disabling of interrupts, and WFE wake-up events, by EDSCR.INTdis is possible only when external debug is enabled.

### The Send Event instructions

The Send Event instructions are:

**SEV, Send Event** This causes an event to be signaled to all PEs in the multiprocessor system.

**SEVL, Send Event Local**

This must set the local Event Register.

**Note**

It might signal an event to other PEs by some IMPLEMENTATION DEFINED mechanism, but is not required to do so.

The mechanism that signals an event to other PEs is IMPLEMENTATION DEFINED. The PE is not required to guarantee the ordering of this event with respect to the completion of memory accesses by instructions before the SEV instruction. Therefore, ARM recommends that software includes a DSB instruction before any SEV instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

A DSB instruction ensures that no instructions, including any SEV instructions, that appear in program order after the DSB instruction, can execute until the DSB instruction has completed. See *Data Synchronization Barrier (DSB)* on page B2-88.

The SEVL instruction appears to execute in program order relative to any subsequent WFE instruction executed on the same PE, without the need for any explicit insertion of barrier instructions.

The receipt of a signaled SEV or SEVL event by a PE sets the Event Register on that PE.

The SEV and SEVL instructions are available at all Exception levels.

### Pseudocode description of the Wait For Event mechanism

This section identifies pseudocode functions that describe the behavior of the Wait For Event mechanism.

The `ClearEventRegister()` pseudocode function clears the Event Register of the current PE.

The `EventRegistered()` pseudocode function returns TRUE if the Event Register of the current PE is set and FALSE if it is clear.

The `WaitForEvent()` pseudocode function optionally suspends execution until a WFE wake-up event or reset occurs, or until some earlier time if the implementation chooses. It is IMPLEMENTATION DEFINED whether restarting execution after the period of suspension causes `ClearEventRegister()` to be called.

The `SendEvent()` pseudocode function sets the Event Register of every PE in the multiprocessor system.

The `EventRegisterSet()` pseudocode function sets the event register for the local PE.

## D1.17.2 Wait For Interrupt

Software can use the *Wait for Interrupt* (WFI) instruction to cause the PE to enter a low-power state. The PE then remains in that low-power state until it receives a *WFI wake-up event*, or until some other IMPLEMENTATION DEFINED reason causes it to leave the low-power state. The architecture permits a PE to leave the low-power state for any reason, but requires that it must leave the low-power state on receipt of any architected WFI wake-up event.

**Note**

Because the architecture permits a PE to leave the low-power state for any reason, it is permissible for a PE to treat WFI as a NOP, but this is not recommended for lowest power operation.

When the PE leaves a low-power state that was entered as a result of a WFI instruction, that WFI instruction completes.

The architecture does not define the exact nature of the low-power state, except that the execution of a WFI instruction must not cause a loss of memory coherency.

Attempts to enter a low-power state made by software executing at EL0, EL1, or EL2 might be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-1578.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page D1-1594.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-1603.

### WFI wake-up events

The following are *WFI wake-up events*:

- Any physical SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE, that is marked as A as B in the tables in *Asynchronous exception masking* on page D1-1570, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In Non-secure EL1 or EL0, any virtual SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE, that is marked as B in Table D1-18 on page D1-1573 in *Virtual interrupts* on page D1-1572, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed* see *Halting allowed and halting prohibited* on page H2-4821.

  See also *External Debug Request debug event* on page H3-4878.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

**Note**

- WFI wake-up events are never disabled by EDSCR.INTdis, and are never masked by the PSTATE.{A, I, F} mask bits. If wake-up is invoked by an interrupt that is disabled or masked the interrupt is not taken.
- Because debug events are WFI wake-up events, ARM recommends that Wait For Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures that the intervention of debug while waiting does not significantly change the function of the program being debugged.
- Some implementations of the WFI mechanism drain down any pending memory activity before suspending execution. This increases power saving, by increasing the area over which clocks can be stopped. The architecture does not require this operation, therefore software must not rely on the WFI mechanism operating in this way.

### Using WFI to indicate an idle state on bus interfaces

Software can use the WFI mechanism to force quiescence on a PE, and, combined with preventing any possible WFI wakeup events, this can be used to complete an entry into a powerdown state.

Because mechanisms for entering powerdown states are inherently IMPLEMENTATION DEFINED, whether an implementation uses the WFI mechanism is IMPLEMENTATION DEFINED. If it does, the `WFI` instruction forces the suspension of execution, and of all associated bus activity.

The control logic that does this also tracks the activity on the bus interfaces of the PE, so that when the PE has completed all current operations and any associated bus activity has completed, it can signal to an external power controller that there is no ongoing bus activity.

However, the PE must continue to process memory-mapped and external debug interface accesses to debug registers when in the WFI state. The indication of idle state to the system normally only applies to the non-debug functional interfaces used by the PE, not the debug interfaces.

When the OS Double Lock control, OSDLR_EL1.DLK, is 1, the PE must not signal this idle state to the control logic unless it can also guarantee that the debug interface is idle. For more information about the OS Double Lock, see *Debug behavior when the OS Double Lock is locked* on page H6-4932.

**Note**

In a PE that implements separate core and debug power domains, the debug interface referred to in this section is the interface between the core and debug power domains, since the signal to the power controller indicates that the core power domain is idle. For more information about the power domains see *Power domains and debug* on page H6-4925.

The exact nature of this interface is IMPLEMENTATION DEFINED, but the use of Wait For Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it very suitable as the preferred powerdown entry mechanism.

### Pseudocode description of Wait For Interrupt

The `WaitForInterrupt()` pseudocode function optionally suspends execution until a WFI wake-up event or reset occurs, or until some earlier time if the implementation chooses.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.18 Self-hosted debug

The ARMv8-A architecture supports both of the following:

**Self-hosted debug**

The PE itself hosts a debugger. The debugger programs the PE to generate *debug exceptions*. Debug exceptions are accommodated in the ARMv8-A Exception model.

**External debug**

The PE is controlled by an external debugger. The debugger programs the PE to generate *Halting debug events*, that cause the PE to enter *Debug state*. In Debug state, the PE is halted.

This section describes self-hosted debug. It includes:

- *Debug exceptions*.
- *The PSTATE debug mask bit, D*.

For external debug, see part E.

### D1.18.1 Debug exceptions

Debug exceptions occur during normal program flow, if a debugger has programmed the PE to generate them.

For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger might enable one or more debug exceptions.

The possible debug exceptions are:

- Software Breakpoint Instruction exceptions.
- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.
- Software Step exceptions.

Chapter D2 *AArch64 Self-hosted Debug* describes these in detail for AArch64.

For the PE to generate a debug exception requires that:

- The debug exception is enabled. *The debug exception enable controls* on page D2-1640 gives the controls for the different debug exceptions.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level and Security state* on page D2-1643.

Debug exceptions are synchronous exceptions, and are accommodated in the ARMv8 Exception model.

**Note**

Breakpoints and Watchpoints can cause entry to Debug state instead of causing debug exceptions. See Chapter H1 *Introduction to External Debug*.

### D1.18.2 The PSTATE debug mask bit, D

As with all other exceptions, when a debug exception is taken, software must take care to avoid generating another instance of an exception within the exception handler, to avoid recursive entry into the exception handler and loss of return state.

To help avoid this, the ARMv8 architecture provides a debug exception mask bit, PSTATE.D, that can mask Watchpoint, Breakpoint, and Software Step exceptions when the target Exception level is the current Exception level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PSTATE.D is set to 1 on taking an exception. This means that while handling an exception in AArch64 state, Watchpoint, Breakpoint, and Software Step exceptions are masked. This prevents recursive entry at the Exception level that debug exceptions are targeted to.

When execution is in AArch64 state, debug exceptions are also masked implicitly when the target Exception level is lower than the current Exception level.

When the target Exception level is higher than the current Exception level, debug exceptions cannot be masked by PSTATE.D.

Because debug exceptions are synchronous, the architecture requires that debug exceptions are not generated when PSTATE.D is 1. By preventing debug exception generation, debug exceptions cannot be taken at a subsequent time when the Process state D mask bit is cleared to 0.

**Note**

This differs from the behavior for interrupts, where the PSTATE.{A, I, F} mask has the effect of preventing the interrupt from being taken, but instead the interrupt remains pending.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.19 The Performance Monitors Extension

The System registers provide access to a Performance Monitors Unit (PMU), defined as the OPTIONAL Performance Monitors Extension to the architecture, a non-invasive debug resource that provides information about the operation of the PE. The PMU provides:

- A 64-bit cycle counter.
- An IMPLEMENTATION DEFINED number of 32-bit event counters. Each event counter can be configured to count occurrences of a specified event. The events that can be counted are:
  - Architectural and microarchitectural events that are likely to be consistent across many microarchitectures. The PMU architecture uses event numbers to identify an event, and the PMU specification defines which event number must be used for each of these architectural and microarchitectural events.
  - Implementation-specific events. The PMU specification reserves event numbers for implementation-specific events. See Appendix K3 *Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events*.

For more information, see Chapter D5 *The Performance Monitors Extension*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.20 Interprocessing

*Interprocessing* is the term used to describe moving between the AArch64 and AArch32 Execution states.

The Execution state can change only on a change of Exception level. This means that the Execution state can change only on taking an exception to a higher Exception level, or returning from an exception to a lower Exception level.

On taking an exception to a higher Exception level, the Execution state either:

- Remains unchanged.
- Changes from AArch32 state to AArch64 state.

On returning from an exception to a lower Exception level, the Execution state either:

- Remains unchanged.
- Changes from AArch64 state to AArch32 state.

**Note**

If, on taking or returning from an exception, the Exception level remains the same, the Execution state cannot change.

For the description of:

- Exception entry to an Exception level using AArch64, see *Exception entry* on page D1-1537.
- Exception return from an Exception level using AArch64 state, see *Exception return* on page D1-1552.
- Exception return to AArch32 state, see *Exception return to an Exception level using AArch32* on page G1-3796.

  **Note**

  The description in *Handling exceptions that are taken to an Exception level using AArch32* on page G1-3774 is outside the scope of interprocessing, because such exceptions must have been taken from an Exception level that is using AArch32, and therefore there is no change of Execution state.

The following sections describe the behavior associated with interprocessing.

- *Register mappings between AArch32 state and AArch64 state*.
- *State of the general-purpose registers on taking an exception to AArch64 state* on page D1-1628.
- *SPSR, ELR, and AArch64 SP relationships on changing Execution state* on page D1-1629.

### D1.20.1 Register mappings between AArch32 state and AArch64 state

This section defines the architectural mappings between AArch32 state registers and AArch64 state registers.

The mappings describe:

- For exceptions taken from AArch32 state to AArch64 state, where the AArch32 register content is found.
- For exception returns from AArch64 state to AArch32 state, how the AArch32 register content is derived.

The general model is:

- The AArch32 register contents are situated in the bottom 32 bits of the AArch64 registers.
- In AArch32 state, the upper 32 bits of AArch64 registers are inaccessible and are ignored.

**Note**

System software that executes in AArch64 state, such as an OS or Hypervisor, can use these mappings for context save and restore, or to interpret and modify the AArch32 registers of an application or virtual machine.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information see the following subsections:

- *Mapping of the general-purpose registers between the Execution states*.
- *Mapping of the SIMD and floating-point registers between the Execution states* on page D1-1621.
- *Mapping of the System registers between the Execution states* on page D1-1622.

### Mapping of the general-purpose registers between the Execution states

Table D1-80 shows how each of the AArch32 general-purpose registers, R0-R12, SP, and LR, including the banked copies of these registers, maps to an AArch64 general-purpose register. A register in the *AArch64 register* column of the table provides the AArch64 view of the corresponding register in the *AArch32 register* column.

**Note**

For some exceptions, the exception syndrome given in the ESR_ELx identifies one or more register numbers from the issued instruction that generated the exception. Where the exception is taken from an Exception level using AArch32 these register numbers give the AArch64 view of the register. For example, if an exception is taken from AArch32 Abort mode, and the faulting instruction specified R14, the ESR_ELx.ISS field would report this using the value `0b10100`, because register X20 provides the AArch64 view of LR_abt. which is the copy of R14 used in Abort mode.

**Table D1-80 General-purpose register mapping between AArch32 state and AArch64 state**

| AArch32 register | AArch64 register |
|---|---|
| R0 | X0 |
| R1 | X1 |
| R2 | X2 |
| R3 | X3 |
| R4 | X4 |
| R5 | X5 |
| R6 | X6 |
| R7 | X7 |
| R8_usr | X8 |
| R9_usr | X9 |
| R10_usr | X10 |
| R11_usr | X11 |
| R12_usr | X12 |
| SP_usr | X13 |
| LR_usr | X14 |
| SP_hyp | X15 |
| LR_irq | X16 |
| SP_irq | X17 |
| LR_svc | X18 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D1-80 General-purpose register mapping between AArch32 state and AArch64 state**

| AArch32 register | AArch64 register |
|---|---|
| SP_svc | X19 |
| LR_abt | X20 |
| SP_abt | X21 |
| LR_und | X22 |
| SP_und | X23 |
| R8_fiq | X24 |
| R9_fiq | X25 |
| R10_fiq | X26 |
| R11_fiq | X27 |
| R12_fiq | X28 |
| SP_fiq | X29 |
| LR_fiq | X30 |

**Note**

For a description of the banking of AArch32 general-purpose registers R8-R12, SP, and LR, see *AArch32 general-purpose registers, the PC, and the Special-purpose registers* on page G1-3763.

### Mapping of the SIMD and floating-point registers between the Execution states

Table D1-81 shows the mapping between the AArch64 V registers and the AArch32 Q registers.

**Table D1-81 SIMD and floating-point register mapping between AArch64 state and AArch32 state**

| AArch64 register | AArch32 register |
|---|---|
| V0 | Q0 |
| V1 | Q1 |
| V2 | Q2 |
| . | . |
| . | . |
| . | . |
| V15 | Q15 |

The AArch64 registers V16-V31 are not accessible from AArch32 state.

The mapping between the V, D, and S registers in AArch64 state is not the same as the mapping between the Q, D, and S registers in AArch32 state:

- In AArch64 state, there are:
  - 32 128-bit V registers, V0-V31.
  - 32 64-bit D registers, D0-D31.
  - 32 32-bit S registers, S0-S31.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A smaller register occupies the least-significant bytes of the corresponding larger register. For example, S5 is the least-significant word of D5 and V5. Figure D1-3 shows this mapping.

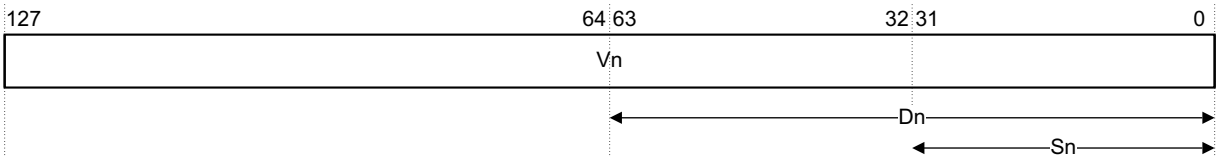



**Figure D1-3 AArch64 state SIMD and floating-point register mappings**

- In AArch32 state, there are:
  - 16 128-bit Q registers, Q0-Q15.
  - 32 64-bit D registers, D0-D31.
  - 32 32-bit S registers, S0-S31.

  Smaller registers are packed into larger registers. Figure D1-4 shows this mapping.

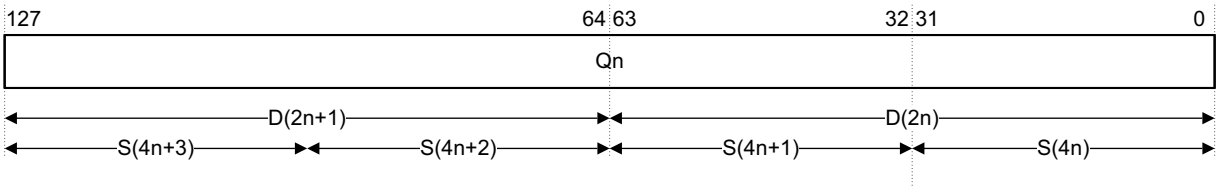



**Figure D1-4 AArch32 state SIMD and floating-point register mappings**

In AArch32 state:

- There are no S registers that correspond to Q8-Q15.
- D16-D31 pack into Q8-Q15. For example, D16 and D17 pack into Q8.

**Note**

A consequence of this mapping is that if software executing in AArch64 state interprets D or S registers from AArch32 state, it must unpack the D or S registers from the V registers before it uses them.

### Mapping of the System registers between the Execution states

ARMv8 architecturally defines the relationship between the AArch64 System registers and the AArch32 System registers, to allow supervisory code such as a hypervisor, that is executing in AArch64 state, to save, restore, and interpret the System registers belonging to a lower Exception level that is using AArch32.

Any modifications made to AArch32 System registers affects only those parts of those AArch64 registers that are mapped to the AArch32 System registers. Bits[63:32] of AArch64 registers, where they are not mapped to AArch32 registers, are unchanged by AArch32 state execution.

**Note**

This model is different to the model for the general-purpose registers described in *Mapping of the general-purpose registers between the Execution states* on page D1-1620. In this model, there are several cases where two AArch32 System registers are packed into a single AArch64 System register.

When EL3 is implemented and is using AArch32, some System registers are banked between the two Security states. When a register is banked in this way, there is an instance of the register in Secure state, and another instance of the register in Non-secure state. In Table D1-82 on page D1-1623 these banked registers are identified by footnote[a]. This banking is not supported when EL3 is using AArch64 or if EL3 is not implemented. This means that when EL3 is implemented and is using AArch64, exactly the same registers are accessed in the following states:

- Secure EL1 with EL1 using AArch32.
- Non-secure EL1 with EL1 using AArch32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This means that, architecturally, it is not possible to determine whether an AArch64 register is mapped onto the Secure instance of the corresponding AArch32 register, or onto the Non-secure instance of that register. When EL3 is using AArch64, the interrupt asserted by the AArch64 CNTP_* timer is the same interrupt as is asserted by the Non-secure AArch32 CNTP_* timer when EL3 is using AArch32.

**Note**

Although the architecture does not require this, because it is not architecturally visible, ARM expects that implementations will map many of the AArch64 registers for use by EL3 to the Secure instances of the banked AArch32 registers, and will map many of the AArch64 registers for use by EL1 to the Non-secure instances of the banked AArch32 registers. However, if EL2 and EL3 are implemented and both support use of AArch32, this is not possible for the following registers:

**IFAR** This is because when EL3 is using AArch32, HIFAR is an alias of the Secure IFAR.

**DFAR** This is because when EL3 is using AArch32, HDFAR is an alias of the Secure DFAR.

Table D1-82 shows the mappings between the writable AArch64 System registers and the AArch32 System registers.

**Table D1-82 Mapping of writable AArch64 System registers to the AArch32 System registers**

| AArch64 register | AArch32 register |
|---|---|
| ACTLR_EL1 | ACTLR[a], and, if implemented, ACTLR2[a]. |
| AFSR0_EL1 | ADFSR[a] |
| AFSR1_EL1 | AIFSR[a] |
| AMAIR_EL1[31:0] | AMAIR0[a] |
| AMAIR_EL1[63:32] | AMAIR1[a] |
| CONTEXTIDR_EL1 | CONTEXTIDR[a] |
| CPACR_EL1 | CPACR |
| CSSELR_EL1 | CSSELR[a] |
| DACR32_EL2 | DACR[a] |
| FAR_EL1[31:0] | DFAR[a] |
| ESR_EL1 | DFSR[a] |
| HACR_EL2 | HACR |
| ACTLR_EL2 | HACTLR and, if implemented, HACTLR2. |
| AFSR0_EL2 | HADFSR |
| AFSR1_EL2 | HAIFSR |
| AMAIR_EL2[31:0] | HAMAIR0 |
| AMAIR_EL2[63:32] | HAMAIR1 |
| CPTR_EL2 | HCPTR |
| HCR_EL2[31:0] | HCR |
| HCR_EL2[63:32] | HCR2 |
| MDCR_EL2 | HDCR |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D1-82 Mapping of writable AArch64 System registers to the AArch32 System registers**

| AArch64 register | AArch32 register |
|---|---|
| FAR_EL2[31:0] | HDFAR |
| FAR_EL2[63:32] | HIFAR |
| MAIR_EL2[31:0] | HMAIR0 |
| MAIR_EL2[63:32] | HMAIR1 |
| HPFAR_EL2[31:0] | HPFAR |
| SCTLR_EL2 | HSCTLR |
| ESR_EL2 | HSR |
| HSTR_EL2 | HSTR |
| TCR_EL2 | HTCR |
| TPIDR_EL2[31:0] | HTPIDR |
| TTBR0_EL2 | HTTBR |
| VBAR_EL2[31:0] | HVBAR |
| FAR_EL1[63:32] | IFAR[a] |
| IFSR32_EL2 | IFSR[a] |
| MAIR_EL1[63:32] | NMRR or MAIR1[a] |
| PAR_EL1 | PAR[a] |
| MAIR_EL1[31:0] | PRRR or MAIR0[a] |
| RMR_EL1 | RMR (at EL1) |
| RMR_EL2 | HRMR |
| RMR_EL3 | RMR (at EL3) |
| SCTLR_EL1 | SCTLR[a] |
| SDER32_EL3 | SDER |
| TPIDR_EL1[31:0] | TPIDRPRW[a] |
| TPIDRRO_EL0[31:0] | TPIDRURO[a] |
| TPIDR_EL0[31:0] | TPIDRURW[a] |
| TCR_EL1[31:0] | TTBCR[a] |
| TTBR0_EL1 | TTBR0[a] |
| TTBR1_EL1 | TTBR1[a] |
| VBAR_EL1[31:0] | VBAR[a] |
| VMPIDR_EL2[31:0] | VMPIDR |
| VPIDR_EL2 | VPIDR |
| VTCR_EL2 | VTCR |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D1-82 Mapping of writable AArch64 System registers to the AArch32 System registers**

| AArch64 register | AArch32 register |
|---|---|
| VTTBR_EL2 | VTTBR |
| **Timer registers** | |
| CNTFRQ_EL0 | CNTFRQ |
| CNTHCTL_EL2 | CNTHCTL |
| CNTHP_CTL_EL2 | CNTHP_CTL |
| CNTHP_CVAL_EL2[63:0] | CNTHP_CVAL |
| CNTHP_TVAL_EL2 | CNTHP_TVAL |
| CNTKCTL_EL1 | CNTKCTL |
| CNTP_CTL_EL0 | CNTP_CTL[a] |
| CNTP_CVAL_EL0[63:0] | CNTP_CVAL[a] |
| CNTP_TVAL_EL0 | CNTP_TVAL[a] |
| CNTPCT_EL0[63:0] | CNTPCT |
| CNTV_CTL_EL0 | CNTV_CTL |
| CNTV_CVAL_EL0[63:0] | CNTV_CVAL |
| CNTV_TVAL_EL0 | CNTV_TVAL |
| CNTVCT_EL0[63:0] | CNTVCT |
| CNTVOFF_EL2[63:0] | CNTVOFF |
| **Debug System registers** | |
| DBGAUTHSTATUS_EL1 | DBGAUTHSTATUS |
| DBGBCR<n>_EL1 | DBGBCR<n> |
| DBGBVR<n>_EL1[31:0] | DBGBVR<n> |
| DBGBVR<n>_EL1[63:32] | DBGBXVR<n> |
| DBGCLAIMCLR_EL1 | DBGCLAIMCLR |
| DBGCLAIMSET_EL1 | DBGCLAIMSET |
| DBGDTR_EL0 | DBGDTRRXint or the DBGDTRTXint |
| DBGDTRRX_EL0 | DBGDTRRXint |
| DBGDTRTX_EL0 | DBGDTRRXint |
| DBGPRCR_EL1 | DBGPRCR |
| DBGVCR32_EL2 | DBGVCR |
| DBGWCR<n>_EL1 | DBGWCR<n> |
| DBGWVR<n>_EL1[31:0] | DBGWVR<n> |
| ID_DFR0_EL1 | ID_DFR0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D1-82 Mapping of writable AArch64 System registers to the AArch32 System registers**

| AArch64 register | AArch32 register |
|---|---|
| MDCCSR_EL0[b] | DBGDSCRint[b] |
| MDCR_EL2 | HDCR |
| MDRAR_EL1 | DBGDRAR |
| MDSCR_EL1[b] | DBGDSCRext[b] |
| OSDLR_EL1 | DBGOSDLR |
| OSDTRRX_EL1[b] | DBGDTRRXext[b] |
| OSDTRTX_EL1[b] | DBGDTRTXext[b] |
| OSECCR_EL1 | DBGOSECCR |
| OSLAR_EL1 | DBGOSLAR |
| OSLSR_EL1 | DBGOSLSR |
| SDER32_EL3 | SDER |
| **Performance Monitors System registers** | |
| PMCCNTR_EL0[31:0] | PMCCNTR (MRC/MCR) |
| PMCEID0_EL0 | PMCEID0 |
| PMCEID1_EL0 | PMCEID1 |
| PMCNTENCLR_EL0 | PMCNTENCLR |
| PMCNTENSET_EL0 | PMCNTENSET |
| PMCR_EL0 | PMCR |
| PMEVCNTR<n>_EL0 | PMEVCNTR<n> |
| PMEVTYPER<n>_EL0 | PMEVTYPER<n> |
| PMINTENCLR_EL1 | PMINTENCLR |
| PMINTENSET_EL1 | PMINTENSET |
| PMOVSCLR_EL0 | PMOVSR |
| PMOVSSET_EL0 | PMOVSSET |
| PMSELR_EL0 | PMSELR |
| PMSWINC_EL0 | PMSWINC |
| PMUSERENR_EL0 | PMUSERENR |
| PMXEVCNTR_EL0 | PMXEVCNTR |
| PMXEVTYPER_EL0 | PMXEVTYPER |

a. AArch32 registers that are banked if EL3 is using AArch32.

b. These registers have overlapping register content. One or more bits of one register appear in the other register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

There are a small number of AArch32 System registers that are not mapped to any AArch64 System registers. The AArch64 registers listed in Table D1-83 can be used to access these from a higher Exception level that is using AArch64. The registers shown in the table are UNDEFINED if EL1 cannot use AArch32.

**Table D1-83 AArch64 registers for accessing registers that are only used in AArch32 state**

| AArch32 register | AArch64 register provided for accessing the AArch32 register | Short description |
|---|---|---|
| DACR | DACR32_EL2 | Domain Access Control Register |
| DBGVCR | DBGVCR32_EL2 | Debug Vector Catch Register |
| FPEXC | FPEXC32_EL2 | Floating-Point Exception Control Register |
| IFSR | IFSR32_EL2 | Instruction Fault Status Register |
| SDER | SDER32_EL3 | AArch32 Secure Debug Enable Register |

Table D1-84 shows the AArch64 System registers that allow access from AArch64 state to the AArch32 ID registers. These registers are RAZ if no Exception level can use AArch32.

**Table D1-84 AArch64 registers that access the AArch32 ID registers**

| AArch32 register | AArch64 register for access to the AArch32 register | Short description |
|---|---|---|
| ID_AFR0 | ID_AFR0_EL1 | AArch32 Auxiliary Feature Register 0 |
| ID_DFR0 | ID_DFR0_EL1 | AArch32 Debug Feature Register 0 |
| ID_ISAR0 | ID_ISAR0_EL1 | EL1, AArch32 Instruction Set Attribute Register 0 |
| ID_ISAR1 | ID_ISAR1_EL1 | EL1, AArch32 Instruction Set Attribute Register 1 |
| ID_ISAR2 | ID_ISAR2_EL1 | EL1, AArch32 Instruction Set Attribute Register 2 |
| ID_ISAR3 | ID_ISAR3_EL1 | EL1, AArch32 Instruction Set Attribute Register 3 |
| ID_ISAR4 | ID_ISAR4_EL1 | EL1, AArch32 Instruction Set Attribute Register 4 |
| ID_ISAR5 | ID_ISAR5_EL1 | EL1, AArch32 Instruction Set Attribute Register 5 |
| ID_MMFR0 | ID_MMFR0_EL1 | AArch32 Memory Model Feature Register 0 |
| ID_MMFR1 | ID_MMFR1_EL1 | AArch32 Memory Model Feature Register 1 |
| ID_MMFR2 | ID_MMFR2_EL1 | AArch32 Memory Model Feature Register 2 |
| ID_MMFR3 | ID_MMFR3_EL1 | AArch32 Memory Model Feature Register 3 |
| ID_MMFR4 | ID_MMFR4_EL1 | AArch32 Memory Model Feature Register 4 |
| ID_PFR0 | ID_PFR0_EL1 | AArch32 PE Feature Register 0 |
| ID_PFR1 | ID_PFR1_EL1 | AArch32 PE Feature Register 1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D1.20.2 State of the general-purpose registers on taking an exception to AArch64 state

When an exception is taken from AArch32 state to AArch64 state, the state of a general-purpose register depends on whether, immediately before the exception, the register was accessible from AArch32 state, as follows:

**If the general-purpose register was accessible from AArch32 state**

The upper 32 bits either become zero, or hold the value that the same architectural register held before any AArch32 execution. The choice between these two options is IMPLEMENTATION DEFINED, and might vary dynamically within an implementation. Correspondingly, software must regard the value as being a CONSTRAINED UNPREDICTABLE choice between these two values.

This behavior applies regardless of whether any execution occurred at the Exception level that was using AArch32. That is, this behavior applies even if AArch32 state was entered by an exception return from AArch64 state, and another exception was immediately taken to AArch64 state without any instruction execution in AArch32 state.

Which general-purpose registers have their upper 32 bits affected in this way depends on both:

- The AArch64 state target Exception level.
- The values of both:
  - SCR_EL3.RW.
  - HCR_EL2.RW or HCR.RW, where HCR.RW is a notional bit that is RES0.

Table D1-85 shows which general-purpose registers can have their upper 32 bits set to zero.

**Table D1-85 General-purpose registers that can have their upper 32 bits set to zero on taking an exception to AArch64 state from AArch32 state**

| SCR_EL3.RW | HCR_EL2.RW or HCR.RW[a] | Registers when the target Exception level is: | | |
|---|---|---|---|---|
| | | EL3 | EL2 | EL1 |
| 0 | 0 | X0-X30 | -[b] | -[b] |
| 0 | 1 | -[c] | -[c] | -[c] |
| 1 | 0 | X0-X14, X16-X30 | X0-X14, X16-X30 | -[b] |
| 1 | 1 | X0-X14 | X0-X14 | X0-X14 |

a. HCR.RW is a notional bit that is RES0.
b. The RW bit values are not valid for the targeted Exception level.
c. Not valid because the RW bit values would imply that EL2 is AArch32 and EL1 is AArch64.

**Note**

If EL2 is not implemented, or the SCR_EL3.NS or SCR.NS bit prevents its use, then as described in *The effects of supporting fewer than four Exception levels* on page D1-1633, the behavior is consistent with HCR_EL2.RW taking the value of SCR_EL3.RW.

**If the general-purpose register was not accessible from AArch32 state**

The general rule is that the register retains the state it had before any AArch32 execution.

There is one exception to this rule, that is when taking an exception to EL3 using AArch64 when either EL2 is not implemented or EL1 is in Secure state. In these cases, the X15 register must be treated as if it is accessible when the value of SCR_EL3.RW is 0, and therefore the upper bits of X15 might either be set to zero or retain their previous value.

Which general-purpose registers retain their state depends on both:

- The AArch64 state target Exception level.
- The values of both:
  - SCR_EL3.RW.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— HCR_EL2.RW or HCR.RW, where HCR.RW is a notional bit that is RES0.

Table D1-86 shows which general-purpose registers can retain their state.

**Table D1-86 General-purpose registers that can retain their state on taking an exception to AArch64 from AArch32**

| SCR_EL3.RW | HCR_EL2.RW or HCR.RW[a] | Registers when the target Exception level is: | | |
|---|---|---|---|---|
| | | EL3 | EL2 | EL1 |
| 0 | 0 | None | -[b] | -[b] |
| 0 | 1 | -[c] | -[c] | -[c] |
| 1 | 0 | X15 | X15 | -[b] |
| 1 | 1 | X15-X30 | X15-X30 | X15-X30 |

a. HCR.RW is a notional bit that is RES0.
b. The RW bit values are not valid for the targeted Exception level.
c. Not valid because the RW bit values would imply that EL2 is AArch32 and EL1 is AArch64.

**Note**

If EL2 is not implemented, or the SCR_EL3.NS bit prevents its use, then as described in *The effects of supporting fewer than four Exception levels* on page D1-1633, the behavior is consistent with HCR_EL2.RW taking the value of SCR_EL3.RW.

### D1.20.3 SPSR, ELR, and AArch64 SP relationships on changing Execution state

Table D1-87 shows the SPSR and ELR registers that are architecturally mapped between AArch32 state and AArch64 state.

**Table D1-87 SPSR and ELR mappings between AArch32 state and AArch64 state**

| AArch32 register | AArch64 register |
|---|---|
| SPSR_svc | SPSR_EL1 |
| SPSR_hyp | SPSR_EL2 |
| ELR_hyp | ELR_EL2 |

On exception entry to EL3 using AArch64 state from an Exception level using AArch32 state, when EL2 has been using AArch32 state, the upper 32-bits of ELR_EL2 are either set to zero or they retain the value before the AArch32 state execution. The implementation determines the choice between these two options, and the choice might vary dynamically within an implementation. Therefore, software must regard the upper 32-bits as being UNKNOWN.

On exception entry to an Exception level using AArch64 state from an Exception level using AArch32 state, the AArch64 Stack Pointers and Exception Link Registers associated with an Exception level that are not accessible during execution in AArch32 state at that Exception level, retain the state that they had before the execution in AArch32 state.

The following AArch32 registers are used only during execution in AArch32 state. However, they retain their state when there is execution at EL1 with EL1 using AArch64 state:

- SPSR_abt.
- SPSR_und.
- SPSR_irq.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- SPSR_fiq.

**Note**

- These registers are accessible during execution in AArch64 state at Exception levels higher than EL1, for context switching.
- If EL1 does not support execution in AArch32 state then these registers are RES0.

On exception entry to an Exception level using AArch64 from an Exception level using AArch32, the AArch64 Stack Pointers and Exception Link Registers associated with an Exception level that are not accessible during AArch32 execution at that Exception level retain the state that they had before AArch32 execution. This applies to the following registers:

- SP_EL0.
- SP_EL1.
- SP_EL2.
- ELR_EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D1.21 The effect of implementation choices on the programmers' model

Three of the implementation choices in ARMv8 are:

- The number of Exception levels implemented.
- Which Exception levels support AArch32 and which Exception levels support AArch64.
- Whether SIMD and floating-point support is implemented.

The following subsections give more information about how these choices affect the programmers' model:

- *Implication of Exception levels implemented*.
- *Support for Exception levels and Execution states* on page D1-1632.
- *Implementations not including Advanced SIMD and floating-point instructions* on page D1-1633.
- *The effects of supporting fewer than four Exception levels* on page D1-1633.

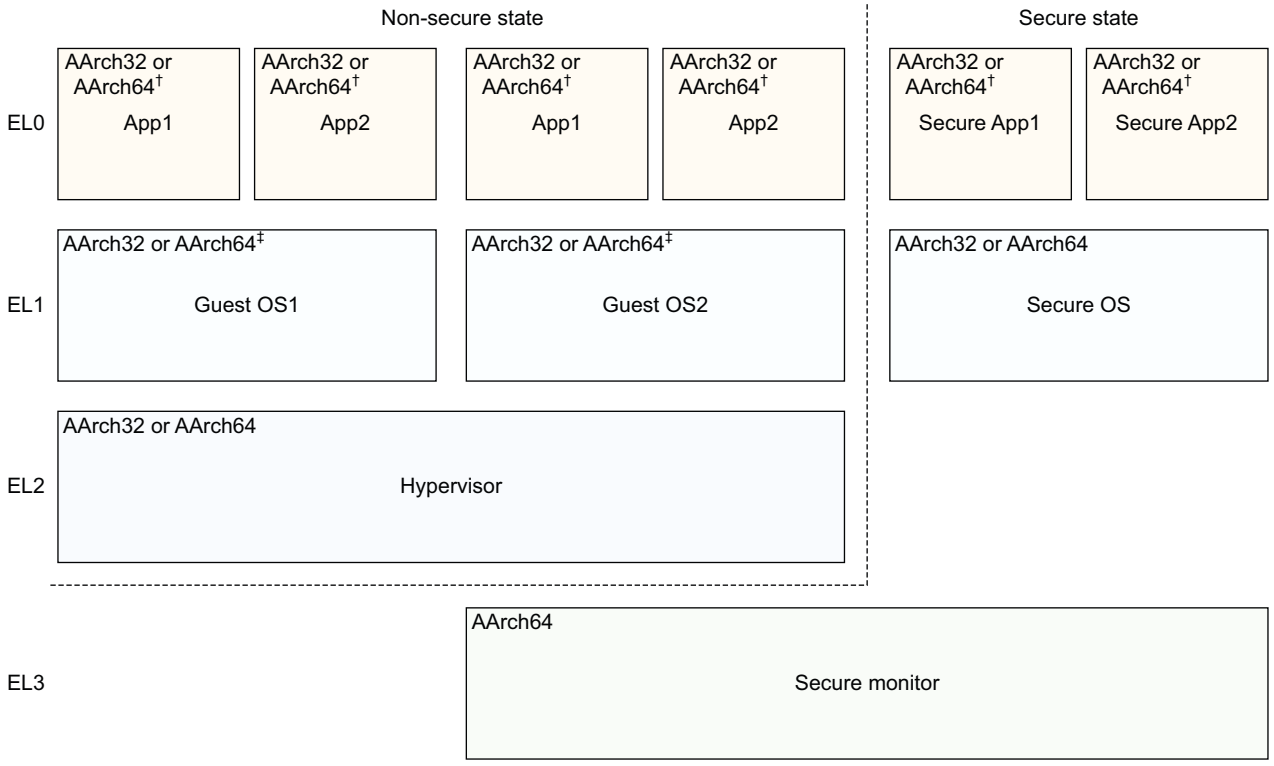
### D1.21.1 Implication of Exception levels implemented

All implementations must include EL0 and EL1.

EL2 and EL3 are optional. The architecture permits all combinations of EL2 and EL3.

See also *Implementations not including Advanced SIMD and floating-point instructions* on page D1-1633 and *The effects of supporting fewer than four Exception levels* on page D1-1633.

For an implementation that includes all of the Exception levels Figure D1-5 shows the implemented Exception levels and the possible Execution states at lower Exception levels when EL3 is using AArch64. Figure D1-5 applies regardless of whether EL3 also supports use of AArch32.




† AArch64 permitted only if EL1 is using AArch64
‡ AArch64 permitted only if EL2 is using AArch64

**Figure D1-5 ARMv8-A security model when EL3 is using AArch64**

The possible combinations of Exception levels are as follows:

- EL0, EL1, and EL2. The implementation supports only Non-secure state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- EL0, EL1, and EL3. The implementation does not support Virtualization. The Exception levels and Execution states depend on whether EL3 is using AArch64 state or AArch32 state, as follows:
  - If EL3 is using AArch64, the Exception levels and Execution states are as shown in Figure D1-5 on page D1-1631 with EL2 removed and no Non-secure state virtualization of EL1 and EL0.
  - If EL3 is using AArch32, the Exception levels and Execution states are as shown in Figure G1-1 on page G1-3754 with EL2 removed and no Non-secure state virtualization of EL1 and EL0.
- EL0 and EL1 only. The implementation supports only a single Security state. This might be either Secure state or Non-secure state, see *Behavior when only EL1 and EL0 are implemented* on page D1-1634.
- EL0, EL1, EL2, and EL3, as described in this section.

For more information, see *The effects of supporting fewer than four Exception levels* on page D1-1633.

### D1.21.2 Support for Exception levels and Execution states

Subject to the interprocessing rules defined in *Interprocessing* on page D1-1619, an implementation of the ARM architecture could support:

- AArch64 state only.
- AArch64 and AArch32 states.
- AArch32 state only.

This means the ARMv8-A architecture can, potentially, support implementations with very large number of combinations of Execution state and Exception level. ARM intends to license only a subset of the possible combinations. Table D1-88 shows the combinations of Exception levels and Execution states that are currently licensed.

**Table D1-88 Supported combinations of Exception levels and Execution state**

| Number of Exception levels | Supported Security states | Exception levels, AArch64 state | | | | Exception levels, AArch32 state | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | EL3 | EL2 | EL1 | EL0 | EL3 | EL2 | EL1 | EL0 |
| Four | Both | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| | | Yes | Yes | Yes | Yes | No | No | Yes | Yes |
| | | Yes | Yes | Yes | Yes | No | No | No | Yes |
| | | Yes | Yes | Yes | Yes | No | No | No | No |
| | | No | No | No | No | Yes | Yes | Yes | Yes |
| Three | Both | Yes | No | Yes | Yes | No | No | Yes | Yes |
| | | Yes | No | Yes | Yes | No | No | No | Yes |
| | | Yes | No | Yes | Yes | No | No | No | No |
| | | No | No | No | No | Yes | No | Yes | Yes |
| | Non-secure only | No | Yes | Yes | Yes | No | No | Yes | Yes |
| | | No | Yes | Yes | Yes | No | No | No | No |
| Two | Either | No | No | Yes | Yes | No | No | No | Yes |
| | | No | No | Yes | Yes | No | No | No | No |
| | | No | No | No | No | No | No | Yes | Yes |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D1.21.3 Implementations not including Advanced SIMD and floating-point instructions

In general, ARMv8-A requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets. Exceptionally, for implementations targeting specialized markets, ARM might produce or license an ARMv8-A implementation that does not provide any support for floating-point and Advanced SIMD instructions. In such an implementation:

**In AArch64 state**

- The CPACR_EL1.FPEN field is RES0.
- The CPTR_EL2.TFP bit is RES1.
- The CPTR_EL3.TFP bit is RES1.
- Each of the ID_AA64PFR0_EL1.{AdvSIMD, FP} fields is `0b1111`.
- The FPEXC32_EL2, FPCR, and FPSR registers are not implemented, and their encodings are UNDEFINED.
- Attempted accesses to floating-point and Advanced SIMD functionality are UNDEFINED. This means:
  - All floating-point and Advanced SIMD instructions are UNDEFINED.
  - Attempts to access the floating-point and Advanced SIMD System registers are UNDEFINED.

**In AArch32 state**

See *AArch32 implications of not including support for Advanced SIMD and floating-point* on page G1-3841.

### D1.21.4 The effects of supporting fewer than four Exception levels

*The effect of implementation choices on the programmers' model* on page D1-1631 defines the permitted combinations of Exception levels in an ARMv8-A implementation.

In every implementation that supports the highest Exception level using either AArch64 state or AArch32 state, an IMPLEMENTATION DEFINED mechanism determines whether the highest implemented Exception level uses AArch64 state or AArch32 state from a Cold reset. Typically, this mechanism is a configuration input. When the highest level is configured to be AArch64 state, then after a Cold reset execution starts at the reset vector in that Exception level.

The unimplemented Exception levels have no effect on execution:

- No interrupts are routed to these Exception levels.
- No traps that target these Exception levels are active
- All systems calls to unimplemented Exception levels from lower Exception levels are treated as UNDEFINED.
- There is no support for address translation from these Exception levels.
- Any exception return that targets an unimplemented Exception level is treated as an illegal exception return as described in *Illegal return events from AArch64 state* on page D1-1553.
- Every accessible register associated with an unimplemented Exception level is RES0 unless the register is associated with the Exception level only to provide the ability to transfer execution to a lower Exception level.

**Note**

If, for example, EL3 is not implemented and EL2 is the highest implemented Exception level, then because none of the EL3 registers are accessible from EL2, the content of those registers is not architecturally visible.

The following subsections give more information about each of the permitted combinations of Exception levels that do not include all Exception levels.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Behavior when EL2 is not implemented

If EL2 is not implemented and EL3 is implemented:

- If EL1 can use AArch32 then the following registers are not RES0:
  - DACR32_EL2.
  - IFSR32_EL2.
  - FPEXC32_EL2.
  - DBGVCR32_EL2.
- The VMPIDR_EL2 and VPIDR_EL2 behave as follows:
  - Reads of VMPIDR_EL2 return the value of MPIDR_EL1, writes to VMPIDR_EL2 are ignored.
  - Reads of VPIDR_EL2 return the value of MIDR_EL1, writes to VPIDR_EL2 are ignored.
- Behavior is consistent with the HCR_EL2.RW bit taking the value of the SCR_EL3.RW bit for all purposes other than reading the HCR_EL2.
- Virtual interrupts are disabled.
- The following address translation and TLB invalidation instructions are UNDEFINED:
  - AT S1E2R and AT S1E2W.
  - TLBI VAE2, TLBI VALE2, TLBI VAE2IS, TLBI VALE2IS, TLBI ALLE2, TLBI ALLE2IS.

  **Note**

  No other TLB or address translation instructions become UNDEFINED with this combination of Exception levels.
- The SCR_EL3.HCE bit is RES0.
- The CNTHCTL_EL2[1:0] bits are treated as if they have the value `0b11` for all purposes other than reading the CNTHCTL_EL2 register.

### Behavior when EL3 is not implemented

If EL3 is not implemented and EL2 is implemented, then:

- All memory transactions can only access a single physical memory address space.
- The PE behaves as if the value of the SCR_EL3.NS bit is 1, even though the SCR_EL3 is not accessible.

This means that if the PE is part of a system that supports two Security states, it behaves as if it is in Non-secure state, and can only access Non-secure memory.

### Behavior when only EL1 and EL0 are implemented

If EL3 and EL2 are not implemented, it is IMPLEMENTATION DEFINED whether the PE behaves as if the value of the SCR_EL3.NS bit is 1 or the PE behaves as if the value of the SCR_EL3.NS bit is 0.

This means that if the PE is part of a system that supports two Security states:

- If it behaves as if the value of the SCR_EL3.NS bit is 1, it can only access Non-secure memory.
- If it behaves as if the value of the SCR_EL3.NS bit is 0, it can access both Secure memory and Non-secure memory.

**Note**

- The behavior described in this subsection still applies if EL1 is configured to use AArch32.
- The implementation can provide a configuration input that determines, from reset, whether the it behaves as if the value of the SCR_EL3.NS bit is 1, or as if the value of the SCR_EL3.NS bit is 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter D2
# AArch64 Self-hosted Debug

When the PE is using self-hosted debug, it generates *debug exceptions*. This chapter describes the AArch64 self-hosted debug exception model. It is organized as follows:

**Introductory information**


- *About debug exceptions* on page D2-1637.
- *The debug exception enable controls* on page D2-1640.


**The debug Exception model**


- *Routing debug exceptions* on page D2-1641.
- *Enabling debug exceptions from the current Exception level and Security state* on page D2-1643.
- *The effect of powerdown on debug exceptions* on page D2-1645.
- *Summary of the routing and enabling of debug exceptions* on page D2-1646.
- *Pseudocode description of debug exceptions* on page D2-1648.


**The debug exceptions**


- *Software Breakpoint Instruction exceptions* on page D2-1650.
- *Breakpoint exceptions* on page D2-1653.
- *Watchpoint exceptions* on page D2-1671.
- *Vector Catch exceptions* on page D2-1685.
- *Software Step exceptions* on page D2-1686.


**Synchronization requirements**

The behavior of self-hosted debug after changes to System registers, or after changes to the authentication interface, but before a *Context Synchronization Operation* (CSO) guarantees the effects of the changes:


- *Synchronization and debug exceptions* on page D2-1701.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Note**

**Definition of a debugger**

Within this chapter, *debugger* means that part of an operating system, or higher level of system software, that handles debug exceptions and programs the debug System registers. An operating system with rich application environments might provide debug services that support a debugger user interface executing at EL0. From the architectural perspective the debug services are the debugger.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D2.1 About debug exceptions

Debug exceptions occur during normal program flow, if a debugger has programmed the PE to generate them. For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger enables one or more debug exceptions. The debug exceptions that can be generated in an AArch64 stage 1 translation regime are:

- *Software Breakpoint Instruction exceptions*.
- *Breakpoint exceptions*, generated by hardware breakpoints.
- *Watchpoint exceptions* on page D2-1638, generated by hardware watchpoints.
- *Software Step exceptions* on page D2-1638.

In addition, debug exceptions generated in an AArch32 translation regime might be routed to EL2 using AArch64. See *Routing debug exceptions* on page D2-1641. Chapter G2 describes the debug exceptions that can be generated in an AArch32 translation regime.

*Vector Catch exceptions* are exceptions that are never generated in an AArch64 translation regime but can be generated in stage 1 of an AArch32 translation regime and routed to EL2 using AArch64. *Vector Catch exceptions* on page D2-1685 describes the behavior for this case.

The PE can only generate a particular debug exception when both:

1. Debug exceptions are enabled from the current Exception level and Security state.

   See *Enabling debug exceptions from the current Exception level and Security state* on page D2-1643. Software Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.

2. A debugger has enabled that particular debug exception.

   All of the debug exceptions except for Software Breakpoint Instruction exceptions have an enable control contained in the MDSCR_EL1. See *The debug exception enable controls* on page D2-1640.

**Note**

If *halting is allowed* and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause entry to Debug state instead of causing debug exceptions. In Debug state, the PE is halted.

For the definition of halting is allowed, see *Halting allowed and halting prohibited* on page H2-4821.

The following list summarizes each of the debug exceptions:

**Software Breakpoint Instruction exceptions**

*Breakpoint instructions* generate these. Breakpoint instructions are instructions that software developers can use to cause exceptions at particular points in the program flow.

The breakpoint instruction in the A64 instruction set is `BRK #<immediate>`. Whenever one of these is committed for execution, the PE takes a Software Breakpoint Instruction exception.

**PE behavior**

Software Breakpoint Instruction exceptions cannot be masked. The PE takes Software Breakpoint Instruction exceptions regardless of both of the following:

- The current Exception level.
- The current Security state.

For more information, see *Software Breakpoint Instruction exceptions* on page D2-1650.

**Breakpoint exceptions**

The ARMv8-A architecture provides 2-16 hardware breakpoints. These can be programmed to generate Breakpoint exceptions based on particular instruction addresses, or based on particular PE contexts, or both.

For example, a software developer might program a hardware breakpoint to generate a Breakpoint exception whenever the instruction with address `0x1000` is committed for execution.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The ARMv8-A architecture supports the following types of hardware breakpoint for use in stage 1 of an AArch64 translation regime:

- Address.
  - Comparisons are made with the virtual address of each instruction in the program flow.
- Context:
  - Context ID Match. Matches with the Context ID held in the CONTEXTIDR_EL1.
  - VMID Match. Matches with the VMID value held in the VTTBR_EL2.
  - Context ID and VMID Match. Matches with both the Context ID and the VMID value.

An Address breakpoint can link to a Context breakpoint, so that the Address breakpoint only generates a Breakpoint exception if the PE is in a particular context when the address match occurs.

A breakpoint generates a Breakpoint exception whenever an instruction that causes a match is committed for execution.

**PE behavior**

If halting is allowed and EDSCR.HDE is 1, hardware breakpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 *Debug State*.

Otherwise:

- If debug exceptions are enabled, hardware breakpoints cause Breakpoint exceptions.
- If debug exceptions are disabled, hardware breakpoints are ignored.

For more information, see *Breakpoint exceptions* on page D2-1653.

### Watchpoint exceptions

The ARMv8-A architecture provides 2-16 hardware watchpoints. These can be programmed to generate Watchpoint exceptions based on accesses to particular data addresses, or based on accesses to any address in a data address range.

For example, a software developer might program a hardware watchpoint to generate a Watchpoint exception on an access to any address in the data address range `0x1000 - 0x101F`.

A hardware watchpoint can link to a hardware breakpoint, if the hardware breakpoint is a *Linked Context* type. In this case, the watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs.

The smallest data address size that a watchpoint can be programmed to match on is a byte. A single watchpoint can be programmed to match on one or more bytes.

A watchpoint generates a Watchpoint exception whenever an instruction that initiates an access that causes a match is committed for execution.

**PE behavior**

If halting is allowed and EDSCR.HDE is 1, hardware watchpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 *Debug State*.

Otherwise:

- If debug exceptions are enabled, hardware watchpoints cause Watchpoint exceptions.
- If debug exceptions are disabled, hardware watchpoints are ignored.

For more information, see *Watchpoint exceptions* on page D2-1671.

### Vector Catch exceptions

These are not generated in an AArch64 translation regime. They can only be generated in an AArch32 translation regime. See *Vector Catch exceptions* on page D2-1685.

### Software Step exceptions

Software step is a resource that a debugger can use to make the PE single-step instructions.

For example, by using software step, debugger software executing at a higher Exception level can debug software executing at a lower Exception level, by making it single-step instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

After the software being debugged has single-stepped an instruction, the PE takes a Software Step exception.

**PE behavior**

Software step can only be used by a debugger executing in an Exception level that is using AArch64. However, the instruction stepped might be executed in either Execution state, and therefore Software Step exceptions can be taken from either Execution state.

If debug exceptions are enabled, Software Step exceptions can be generated.

If debug exceptions are disabled, software step is inactive.

For more information, see *Software Step exceptions* on page D2-1686.

Table D2-1 summarizes PE behavior and shows the location of the pseudocode for each of the debug exceptions.

**Table D2-1 PE behavior and pseudocode for each of the debug exceptions**

| Debug exception | PE behavior if debug exceptions are: | | Pseudocode |
|---|---|---|---|
| | Enabled | Disabled | |
| Software Breakpoint Instruction exceptions | Takes the exception | Takes the exception | page D2-1651 |
| Breakpoint exceptions | Takes the exception[a] | Ignored | page D2-1666 |
| Watchpoint exceptions | Takes the exception[a] | Ignored | page D2-1682 |
| Vector Catch exceptions | Takes the exception | Ignored | page G2-3943 |
| Software Step exceptions | Takes the exception | Not applicable[b] | page D2-1699 |

a. If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause the PE to enter Debug state instead of causing debug exceptions. See Chapter H2 *Debug State*.

b. Software Step is inactive if debug exceptions are disabled. No Software Step exceptions can be generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D2.2 The debug exception enable controls

The enable controls for each debug exception are as follows:

**Software Breakpoint Instruction exceptions**

None. Software Breakpoint Instruction exceptions are always enabled.

**Breakpoint exceptions**

MDSCR_EL1.MDE, plus an enable control for each breakpoint, DBGBCR<n>_EL1.E.

**Watchpoint exceptions**

MDSCR_EL1.MDE, plus an enable control for each watchpoint, DBGWCR<n>_EL1.E.

**Vector Catch exceptions**

MDSCR_EL1.MDE.

**Software Step exceptions**

MDSCR_EL1.SS.

In addition, for all debug exceptions other than Software Breakpoint Instruction exceptions, software must configure the controls that enable debug exceptions from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level and Security state* on page D2-1643.

The PE cannot take a debug exception if debug exceptions are disabled from either the current Exception level or the current Security state.

Software Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D2.3 Routing debug exceptions

Debug exceptions are usually routed to EL1. However:

- If EL2 is implemented, the routing of debug exceptions taken from Non-secure state depends on MDCR_EL2.TDE:

  **1** Debug exceptions taken from Non-secure state are routed to EL2.

  **0** Debug exceptions behave as follows:
  - Debug exceptions taken from Non-secure EL1 and EL0 are routed to Non-secure EL1.
  - Software Breakpoint Instruction exceptions taken from EL2 are routed to EL2.
  - All other debug exceptions are disabled from EL2 using AArch64.

**Note**

If HCR_EL2.TGE is 1, MDCR_EL2.TDE is treated as being 1 except for a direct read of MDCR_EL2.

Table D2-2 shows this.

**Table D2-2 The effect of the TGE and TDE controls on debug exception routing**

| HCR_EL2.TGE | MDCR_EL2.TDE | Debug exceptions taken from Non-secure state are routed to: |
|---|---|---|
| 0 | 0 | Non-secure EL1[a] |
| 0 | 1 | EL2 |
| 1 | X | EL2 |

a. Software Breakpoint Instruction exceptions taken from EL2 are routed to EL2.

**Note**

If EL2 is not implemented, the PE behaves as if both HCR_EL2.TGE and MDCR_EL2.TDE are 0.

- If EL3 is implemented, Software Breakpoint Instruction exceptions taken from EL3 are routed to EL3.

  All other debug exceptions are disabled from EL3 using AArch64.

Either EL1 or EL2 is the *debug target exception level*, $EL_D$. That is, $EL_D$ is EL1 unless EL2 is implemented and MDCR_EL2.TDE is 1 and the debug exception is taken from Non-secure state, when $EL_D$ is EL2.

The following tables show the routing of debug exceptions:

**Table D2-3 Routing when both EL3 and EL2 are implemented**

| MDCR_EL2.TDE[a] | $EL_D$ when executing in: | | | | | |
|---|---|---|---|---|---|---|
| | Non-secure: | | | Secure: | | |
| | EL0 | EL1 | EL2 | EL0 | EL1 | EL3 |
| 0 | EL1 | EL1 | EL1[b] | EL1 | EL1 | EL1[b] |
| 1 | EL2 | EL2 | EL2 | | | |

a. If HCR_EL2.TGE is 1, this bit is treated as being 1 other than for a direct read of MDCR_EL2.

b. $EL_D$ is EL1. However, all debug exceptions other than Software Breakpoint Instruction exceptions are disabled, and Software Breakpoint Instruction exceptions taken from EL2 are routed to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D2-4 Routing when EL3 is implemented and EL2 is not implemented**

| $EL_D$ when executing in: | | | |
|---|---|---|---|
| Non-secure: | | Secure: | |
| EL0 | EL1 | EL1 | EL3 |
| EL1 | EL1 | EL1 | EL1[a] |

a. $EL_D$ is EL1. However, all debug exceptions other than Software Breakpoint Instruction exceptions are disabled, and Software Breakpoint Instruction exceptions taken from EL2 are routed to EL2.

**Table D2-5 Routing when EL3 is not implemented and EL2 is implemented**

| MDCR_EL2.TDE[a] | $EL_D$ when executing in Non-secure: | | |
|---|---|---|---|
| | EL0 | EL1 | EL2 |
| 0 | EL1 | EL1 | EL1[b] |
| 1 | EL2 | EL2 | EL2 |

a. If HCR_EL2.TGE is 1, this bit is treated as being 1 other than for a direct read of MDCR_EL2.

b. $EL_D$ is EL1. However, all debug exceptions other than Software Breakpoint Instruction exceptions are disabled, and Software Breakpoint Instruction exceptions taken from EL2 are routed to EL2.

### D2.3.1 Pseudocode description of routing debug exceptions

`DebugTarget()` returns the current debug target Exception level.

```
// DebugTarget()
// =============

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

`DebugTargetFrom()` returns the debug target Exception level for the specified Security state.

```
// DebugTargetFrom()
// =================
// Returns the debug exception target Exception level

bits(2) DebugTargetFrom(boolean secure)

    if HaveEL(EL2) && !secure then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D2.4 Enabling debug exceptions from the current Exception level and Security state

A debug exception can only be taken if all of the following are true:

- The OS lock is unlocked.
- EDPRSR.DLK is 0.
- The debug exception is enabled from the current Exception level.
- The debug exception is enabled from the current Security state.

Table D2-6 shows when debug exceptions are enabled from the current Exception level. In the table, $EL_D$ is the Exception level that Table D2-3 on page D2-1641 defines.

**Table D2-6 Whether debug exceptions are enabled from the current Exception level**

| Current Exception level | Software Breakpoint Instruction exceptions | All other debug exceptions |
|---|---|---|
| Any Exception level that is higher than $EL_D$[a] | Enabled | Disabled |
| $EL_D$ | Enabled | Disabled if either of the following is true:<br>• The Local (kernel) Debug Enable bit, MDSCR_EL1.KDE, is 0.<br>• The Debug exception mask bit, PSTATE.D, is 1.<br>Otherwise enabled.<br>This means that a debugger must explicitly enable these debug exceptions from $EL_D$ by setting MDSCR_EL1.KDE to 1 and PSTATE.D to 0. |
| Any Exception level that is lower than $EL_D$ | Enabled | Enabled |

a. This includes EL3. EL3 is always higher than $EL_D$.

**Note**

PSTATE.D is set to 1 at reset and on exception entry.

Table D2-7 shows when debug exceptions are enabled from the current Security state.

**Table D2-7 Whether debug exceptions are enabled from the current Security state**

| Current Security state | Software Breakpoint Instruction exceptions | All other debug exceptions |
|---|---|---|
| Non-secure | Enabled | Enabled |
| Secure | Enabled | Disabled if MDCR_EL3.SDD is 1. See *Disabling debug exceptions from Secure state* on page D2-1644.<br>Otherwise enabled. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D2.4.1 Disabling debug exceptions from Secure state

If EL3 is implemented, software executing at EL3 can set the *Secure Debug Disable* bit, MDCR_EL3.SDD, to 1 to disable all debug exceptions taken from AArch64 Secure state other than Software Breakpoint Instruction exceptions.

The ARMv8-A architecture does not support disabling debug in Non-secure state.

**Note**

- If the boot software executed when reset is deasserted sets MDCR_EL3.SDD to 1, software operating at EL3 never has to switch the debug registers between Secure state and Non-secure state.
- The PE cannot take a debug exception unless it is enabled from the current Exception level. See Table D2-6 on page D2-1643.
- If either the OS lock or the OS double-lock is locked, debug exceptions other than Software Breakpoint Instruction exceptions are disabled.
- If EL3 and EL2 are not implemented, and the implementation is a Secure state only implementation, the PE behaves as if MDCR_EL3.SDD is 0.

### D2.4.2 Pseudocode description of enabling debug exceptions

`AArch64.GenerateDebugExceptions()` determines whether debug exceptions other than Software Breakpoint Instruction exceptions are enabled from the current Exception level and Security state.

```
// AArch64.GenerateDebugExceptions()
// =================================

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);
```

`AArch64.GenerateDebugExceptionsFrom()` determines whether debug exceptions other than Software Breakpoint Instruction exceptions are enabled from the specified Exception level and Security state.

```
// AArch64.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = HaveEL(EL2) && !secure && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    if HaveEL(EL3) && secure then
        enabled = MDCR_EL3.SDD == '0' && from != EL3;
    else
        enabled = TRUE;

    target = if route_to_el2 then EL2 else EL1;
    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';

    return enabled;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D2.5 The effect of powerdown on debug exceptions

*Debug OS Save and Restore sequences* on page H6-4930 describes the *powerdown save routine* and the *restore routine*.

When executing either routine, software must use the OS Lock to disable generation of all of the following:

- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.
- Software Step exceptions.

This is because the generation of these exceptions depends on the state of the debug registers, and the state of the debug registers might be lost over these routines.

Debug exceptions other than Software Breakpoint Instruction exceptions are enabled only if both the OS Lock is unlocked and EDPRSR.DLK is 0.

Software Breakpoint Instruction exceptions are enabled regardless of the state of the OS Lock and EDPRSR.DLK.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D2.6 Summary of the routing and enabling of debug exceptions

Behavior is as follows:

**Software Breakpoint Instruction exceptions**

These are always enabled, regardless of the current Exception level and Security state. Table D2-8 shows the routing of these. In the table, n/a means not applicable.

**Table D2-8 Routing of Software Breakpoint Instruction exceptions**

| Current Security state | MDCR_EL2.TDE[a] | $EL_D$ when enabled from: | | | |
|---|---|---|---|---|---|
| | | EL0 | EL1 | EL2 | EL3 |
| Secure | X | Secure EL1 | Secure EL1 | n/a | EL3 |
| Non-secure | 0 | Non-secure EL1 | Non-secure EL1 | EL2 | n/a |
| | 1 | EL2 | EL2 | EL2 | n/a |

a. If EL2 is not implemented, behavior is as if this is 0. Otherwise, if the value of HCR_EL2.TGE is 1, MDCR_EL2.TDE is treated as being 1 other than for a direct read of MDCR_EL2.

**All other debug exceptions**

Table D2-9 shows the valid combinations of MDCR_EL3.SDD, MDCR_EL2.TDE, MDSCR_EL1.KDE, and PSTATE.D, and for each combination shows where these exceptions are enabled from and where they are taken to.

In the table, n/a means not applicable and a dash, -, means that debug exceptions are disabled from that Exception level.

**Table D2-9 Routing of Breakpoint, Watchpoint, Software Step, and Vector Catch exceptions**

| Debug state | Lock[a] | Current Security state | SDD[b] | TDE[c] | KDE | D | $EL_D$ when enabled from: | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | EL0 | EL1 | EL2 | EL3 |
| Yes | X | X | X | X | X | X | - | - | - | - |
| No | 1 | X | X | X | X | X | - | - | - | - |
| No | 0 | Secure | 1 | X | X | X | - | - | n/a | - |
| No | 0 | Secure | 0 | X | 0 | X | Secure EL1 | - | n/a | - |
| No | 0 | Secure | 0 | X | 1 | 1 | Secure EL1 | - | n/a | - |
| No | 0 | Secure | 0 | X | 1 | 0 | Secure EL1 | Secure EL1 | n/a | - |
| No | 0 | Non-secure | X | 0 | 0 | X | Non-secure EL1 | - | - | n/a |
| No | 0 | Non-secure | X | 0 | 1 | 1 | Non-secure EL1 | - | - | n/a |
| No | 0 | Non-secure | X | 0 | 1 | 0 | Non-secure EL1 | Non-secure EL1 | - | n/a |
| No | 0 | Non-secure | X | 1 | 0 | X | EL2 | EL2 | - | n/a |
| No | 0 | Non-secure | X | 1 | 1 | 1 | EL2 | EL2 | - | n/a |
| No | 0 | Non-secure | X | 1 | 1 | 0 | EL2 | EL2 | EL2 | n/a |

a. The value of (OSLSR_EL1.OSLK OR EDPRSR.DLK).

b. If EL3 is not implemented, behavior is as if this is 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

c. If HCR_EL2.TGE is 1, this bit is treated as being 1 other than for a direct read of MDCR_EL2. If EL2 is not implemented, behavior is as if TDE is 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D2.7 Pseudocode description of debug exceptions

`DebugFault()` returns a `FaultRecord` object that indicates that a memory access has generated a debug exception:

```
// AArch64.DebugFault()
// ====================

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)

    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_Debug, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

The `Abort()` function processes `FaultRecord` objects, as described in *Abort exceptions* on page D3-1736, and generates a debug exception.

`Abort()` calls one of the following:

```
// AArch64.BreakpointException()
// =============================

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

// AArch64.VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert HaveEL(EL2) && !IsSecure() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

// AArch64.SoftwareStepException()
// ===============================

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D2.8 Software Breakpoint Instruction exceptions

This section describes Software Breakpoint Instruction exceptions in an AArch64 translation regime.

For software executing in an Exception level that is using AArch64, a Software Breakpoint Instruction exception results from the execution of an A64 `BRK` instruction. However, within the AArch64 EL1&0 translation regime, executing a T32 or A32 `BKPT` instruction at EL0 using AArch32 generates a Software Breakpoint Instruction exception.

For more information about the T32 and A32 `BKPT` instructions see:

- *Breakpoint instruction in the A32 and T32 instruction sets* on page G2-3895.
- *BKPT instructions as the first instruction in an IT block* on page G2-3896.

The following subsections describe Software Breakpoint Instruction exceptions in an AArch64 translation regime:

- *About Software Breakpoint Instruction exceptions*.
- *Breakpoint instruction in the A64 instruction set*.
- *Exception syndrome information and preferred return address*.
- *Pseudocode description of Software Breakpoint Instruction exceptions* on page D2-1651.

### D2.8.1 About Software Breakpoint Instruction exceptions

A *breakpoint* is a debug event that results from the execution of an instruction, based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

*Software Breakpoint Instruction exceptions*, that this section describes, are software breakpoints. *Breakpoint exceptions* on page D2-1653 describes hardware breakpoints.

There is no enable control for Software Breakpoint Instruction exceptions. They are always enabled, and cannot be masked.

A Software Breakpoint Instruction exception is generated whenever a breakpoint instruction is committed for execution, regardless of all of the following:

- The current Exception level.
- The current Security state.
- Whether the *debug target Exception level*, $EL_D$, is using AArch64 or AArch32.

**Note**

- The debug target exception level, $EL_D$, is the Exception level that debug exceptions are targeting. *Routing debug exceptions* on page D2-1641 describes how $EL_D$ is derived.
- Debuggers using breakpoint instructions must be aware of the ARMv8 rules for concurrent modification and execution of instructions. See *Concurrent modification and execution of instructions* on page B2-82.

### D2.8.2 Breakpoint instruction in the A64 instruction set

The breakpoint instruction is `BRK #<immediate>`. It is unconditional.

For details of the instruction encoding, see *BRK* on page C6-475.

### D2.8.3 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information* on page D2-1651.
- *Preferred return address* on page D2-1651.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Exception syndrome information

On taking a Software Breakpoint Instruction exception, the PE records information about the exception in the *Exception Syndrome Register* (ESR) at the Exception level the exception is taken to. The ESR used is one of:

- ESR_EL1.
- ESR_EL2.
- ESR_EL3.

**Note**

Software Breakpoint Instruction exceptions are the only debug exception that can be taken to EL3 using AArch64.

Table D2-10 shows the information that the PE records.

**Table D2-10 Information recorded in the ESR_ELx**

| ESR_ELx field | Information recorded in ESR_EL1, ESR_EL2, or ESR_EL3. | |
|---|---|---|
| *Exception Class*, EC | Whether the breakpoint instruction was executed in AArch64 state or AArch32 state. The PE sets this to:<br>• `0x3C` for an A64 `BRK` instruction.<br>• `0x38` for an A32 or T32 `BKPT` instruction. | |
| *Instruction Length*, IL | The PE sets this to:<br>• 0 for a 16-bit T32 `BKPT` instruction.<br>• 1 for an A64 `BRK` instruction, or an A32 `BKPT` instruction. | |
| *Instruction Specific Syndrome*, ISS | **ISS[24:16]** | RES0. |
| | **ISS[15:0]** | The PE copies the instruction Comment field value into here, zero extended as necessary. |

**Note**

- If debug exceptions are routed to EL2, it is the exception that is routed, not the instruction that is trapped. Therefore, if a Software Breakpoint Instruction exception is routed to EL2, ESR_EL2.EC is set to the same value as if the exception was taken to EL1.
- For information about how debug exceptions can be routed to EL2, see *Routing debug exceptions* on page D2-1641.

### Preferred return address

The preferred return address is the address of the breakpoint instruction, not the next instruction. This is different to the behavior of other exception-generating instructions, like `SVC`.

## D2.8.4 Pseudocode description of Software Breakpoint Instruction exceptions

`AArch64.SoftwareBreakpoint()` generates a Software Breakpoint Instruction exception that is taken to AArch64 state.

```
// AArch64.SoftwareBreakpoint()
// ============================

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
exception.syndrome<15:0> = immediate;

if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D2.9 Breakpoint exceptions

This section describes Breakpoint exceptions in stage 1 of an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing at either:

- An Exception level that is using AArch64.
- EL0 using AArch32 when EL1 is using AArch64.

It contains the following subsections:

- *About Breakpoint exceptions*.
- *Breakpoint types and linking of breakpoints* on page D2-1654.
- *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-1660.
- *Breakpoint instruction address comparisons* on page D2-1661.
- *Breakpoint context comparisons* on page D2-1663.
- *Breakpoint usage constraints* on page D2-1663.
- *Exception syndrome information and preferred return address* on page D2-1665.
- *Pseudocode description of Breakpoint exceptions taken from AArch64 state* on page D2-1666.

### D2.9.1 About Breakpoint exceptions

A *breakpoint* is a debug event that results from the execution of an instruction, based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

*Breakpoint exceptions* are generated by *Breakpoint debug events*. Breakpoint debug events are generated by hardware breakpoints. Software breakpoints are described in *Software Breakpoint Instruction exceptions* on page D2-1650.

An implementation can include between 2-16 hardware breakpoints. ID_AA64DFR0_EL1.BRPs shows how many are implemented.

To use an implemented hardware breakpoint, a debugger programs the following registers for the breakpoint:

- The *Breakpoint Control Register*, DBGBCR<n>_EL1. This contains controls for the breakpoint, for example an enable control.
- The *Breakpoint Value Register*, DBGBVR<n>_EL1. This holds the value used for breakpoint matching, that is one of:
  - An instruction virtual address.
  - A Context ID.
  - A VMID value.
  - A concatenation of both a Context ID value and a VMID value.

These registers are numbered, so that:

- DBGBCR1_EL1 and DBGBVR1_EL1 are for breakpoint number one.
- DBGBCR2_EL1 and DBGBVR2_EL1 are for breakpoint number two.
- …
- …
- DBGBCR<n>_EL1 and DBGBVR<n>_EL1 are for breakpoint number n.

A debugger can link a breakpoint that is programmed with an address and a breakpoint that is programmed with anything other than an address together, so that a Breakpoint debug event is only generated if both breakpoints match.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For each instruction in the program flow, all of the breakpoints are tested. When a breakpoint is tested, it generates a Breakpoint debug event if all of the following are true:

- The breakpoint is enabled. That is, the breakpoint enable control for it, DBGBCR<n>_EL1.E, is 1.
- The conditions specified in the DBGBCR<n>_EL1 are met.
- The comparison with the value held in the DBGBVR<n>_EL1 is successful.
- If the breakpoint is linked to another breakpoint, the comparisons made by that other breakpoint are also successful.
- The instruction is committed for execution.

If all of these conditions are met, the breakpoint generates the Breakpoint debug event regardless of the following:

- Whether the instruction passes its condition code check.
- The instruction type.

If halting is allowed and EDSCR.HDE is 1, Breakpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:

- Enabled, Breakpoint debug events generate Breakpoint exceptions.
- Disabled, Breakpoint debug events are ignored.

**Note**

The remainder of this Breakpoint exceptions section, including all subsections, describes breakpoints as generating Breakpoint exceptions.

However, the behavior described also applies if breakpoints are causing entry to Debug state.

*The debug exception enable controls* on page D2-1640 describes the enable controls for Breakpoint debug events.

### D2.9.2 Breakpoint types and linking of breakpoints

Each implemented breakpoint is one of the following:

- A *context-aware* breakpoint. This is a breakpoint that can be programmed to generate a Breakpoint exception on any one of the following:
  - An instruction address match.
  - A Context ID match, with the value held in the CONTEXTIDR_EL1.
  - A VMID match, with the VMID value held in the VTTBR_EL2.
  - Both a Context ID match and a VMID match.
- A breakpoint that is not context-aware. These can only be programmed to generate a Breakpoint exception on an instruction address match.

ID_AA64DFR0_EL1.CTX_CMPs shows how many of the implemented breakpoints are context-aware breakpoints. At least one implemented breakpoint must be context-aware. The context-aware breakpoints are the highest numbered breakpoints.

Any breakpoint that is programmed to generate a Breakpoint exception on an instruction address match is categorized as an *Address breakpoint*. Breakpoints that are programmed to match on anything else are categorized as *Context breakpoints*.

When a debugger programs a breakpoint to be an Address or a Context breakpoint, it must also program that breakpoint so that it is either:

- Used in isolation. In this case the breakpoint is called an *Unlinked breakpoint*.
- Enabled for linking to another breakpoint. In this case the breakpoint is called a *Linked breakpoint*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

By linking an Address breakpoint and a Context breakpoint together, the debugger can create a breakpoint pair that only generates a Breakpoint exception if the PE is in a particular context when an instruction address match occurs. For example, a debugger might:

1. Program breakpoint number one to be a *Linked Address Match breakpoint*.
2. Program breakpoint number five to be a *Linked Context ID Match breakpoint*.
3. Link these two breakpoints together. A Breakpoint exception is only generated if both the instruction address matches and the Context ID matches.

The *Breakpoint Type* field for a breakpoint, DBGBCR<n>_EL1.BT, controls the breakpoint type and whether the breakpoint is enabled for linking. If BT[0] is 1, the breakpoint is enabled for linking.

Figure D2-1 shows all of the possible breakpoint types that stage 1 of an AArch64 translation regime supports, and their associated BT field values.

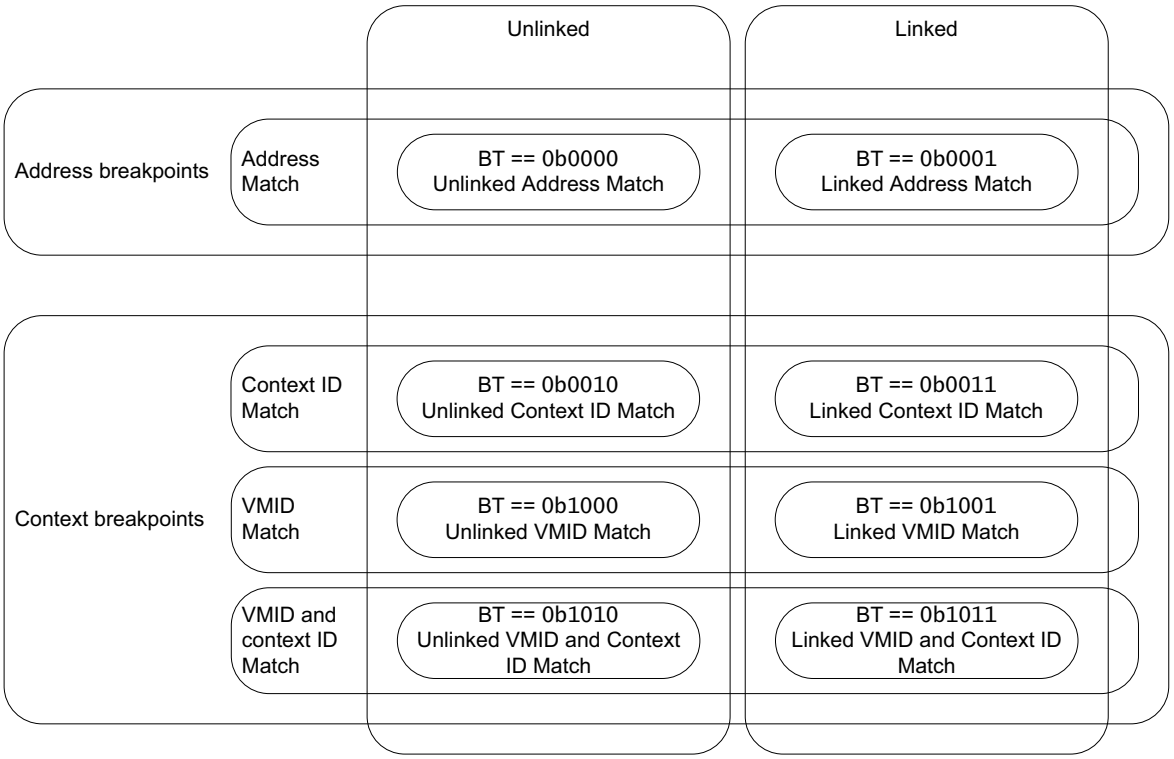



**Figure D2-1 Breakpoint types and their associated BT field values**

If AArch32 state is implemented, Address breakpoints can be programmed to generate Breakpoint exceptions on addresses that are halfword-aligned but not word-aligned. This makes it possible to breakpoint on T32 instructions. See *Specifying the halfword-aligned address that an Address breakpoint matches on* on page D2-1662.

**Note**

Stage 1 of an AArch32 translation regimes supports two additional breakpoint types, Unlinked and Linked Address Mismatch breakpoints, BT == `0b0100` and BT == `0b0101`. For information about these, see Chapter G2 *AArch32 Self-hosted Debug*. These types are reserved in stage 1 of an AArch64 translation regime. See *Reserved BT values* on page D2-1663.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Rules for linking breakpoints

The rules for breakpoint linking are as follows:

- Only Linked breakpoint types can be linked.
- Any type of Linked Address breakpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, DBGBCR<n>_EL1.LBN, for the Linked Address breakpoint specifies the particular Linked Context breakpoint that the Linked Address breakpoint links to, and:
  - DBGBCR<n>_EL1.{SSC, HMC, PMC} for the Linked Address breakpoint define the execution conditions that the breakpoint pair generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-1660.
  - DBGBCR<n>_EL1.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- Linked Context breakpoint types can only be linked to. The LBN field for Context breakpoints is therefore ignored.
- Linked Address breakpoints cannot link to watchpoints. The LBN field can therefore only specify another breakpoint.
- If a Linked Address breakpoint links to a breakpoint that is not context-aware, the behavior of the Linked Address breakpoint is CONSTRAINED UNPREDICTABLE. See *Other usage constraints for Address breakpoints* on page D2-1665.
- If a Linked Address breakpoint links to an Unlinked Context breakpoint, the Linked Address breakpoint never generates any Breakpoint exceptions.
- Multiple Linked Address breakpoints can link to a single Linked Context breakpoint.

  **Note**

  Multiple Linked watchpoints can also link to a single Linked Context breakpoint. *Watchpoint exceptions* on page D2-1671 describes watchpoints.

These rules mean that a single Linked Context breakpoint might be linked to by all, or any combination of, the following:

- Multiple Linked Address Match breakpoints.
- Multiple Linked watchpoints.

It is also possible that a Linked Context breakpoint might have no breakpoints or watchpoints linked to it.

Figure D2-2 on page D2-1657 shows an example of permitted breakpoint and watchpoint linking.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

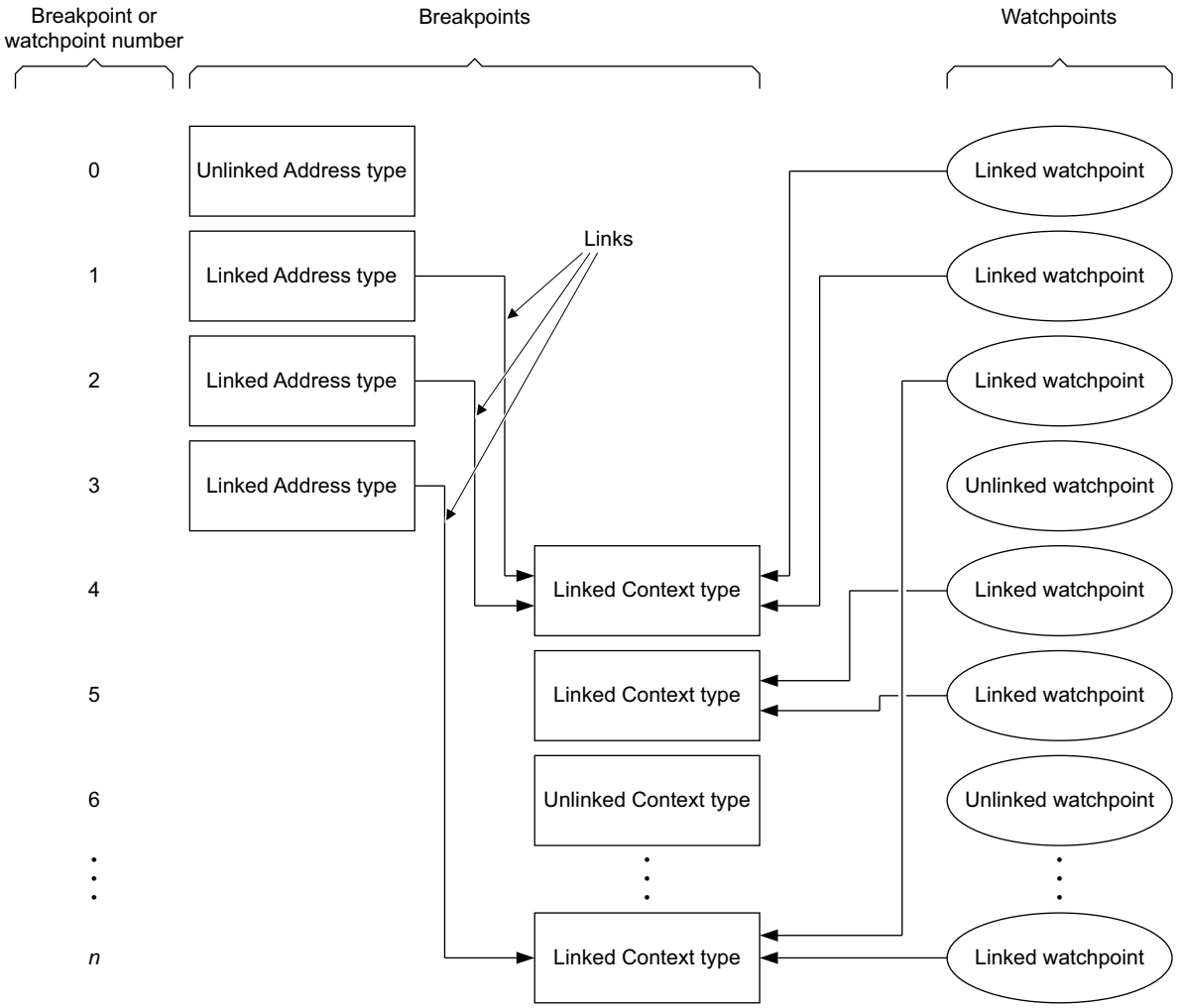



**Figure D2-2 The role of linking in Breakpoint and Watchpoint exception generation**

In Figure D2-2, each Linked Address breakpoint can only generate a Breakpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful. Similarly, each Linked watchpoint can only generate a Watchpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful.

## Breakpoint types defined by DBGBCRn_EL1.BT

The following list provides more detail about each breakpoint type:

**`0b0000`, Unlinked Address Match breakpoint**

Generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-1660.
- A successful address match, as described in *Breakpoint instruction address comparisons* on page D2-1661.

DBGBCR<n>_EL1.LBN for this breakpoint is ignored.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**0b0001, Linked Address Match breakpoint**

Generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>_EL1.{SSC, HMC, PMC} for this breakpoint. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-1660.
- A successful address match defined by this breakpoint, as described in *Breakpoint instruction address comparisons* on page D2-1661.
- A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

DBGBCR<n>_EL1.LBN for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

**0b0010, Unlinked Context ID Match breakpoint**

BT == 0b0010 is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-1660.
- A successful Context ID match, as described in *Breakpoint context comparisons* on page D2-1663.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored

**0b0011, Linked Context ID Match breakpoint**

BT == 0b0011 is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page D2-1661.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-1663.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page D2-1675.
  - A successful Context ID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-1663.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

**0b0100, Unlinked Address Mismatch breakpoint**

BT == 0b0100 is a reserved value in stage 1 of an AArch64 translation regime. See *Reserved BT values* on page D2-1663.

*0b0100, Unlinked Address Mismatch breakpoint* on page G2-3903 describes the behavior of Address Mismatch breakpoints in stage 1 of an AArch32 translation regime.

**0b0101, Linked Address Mismatch breakpoint**

BT == 0b0101 is a reserved value in stage 1 of an AArch64 translation regime. See *Reserved BT values* on page D2-1663.

*0b0101, Linked Address Mismatch breakpoint* on page G2-3904 describes the behavior of Address Mismatch breakpoints in stage 1 of an AArch32 translation regime.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**0b1000, Unlinked VMID Match breakpoint**

BT == 0b1000 is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-1660.
- A successful VMID match, as described in *Breakpoint context comparisons* on page D2-1663.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

**0b1001, Linked VMID Match breakpoint**

BT == 0b1000 is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address Match breakpoint that links to this breakpoint. See *Breakpoint instruction address comparisons* on page D2-1661.
  - A successful VMID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-1663.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page D2-1675.
  - A successful VMID match defined by this breakpoint, as described in *Breakpoint context comparisons* on page D2-1663.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

**0b1010, Unlinked Context ID and VMID Match breakpoint**

BT == 0b1010 is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates a Breakpoint exception for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-1660.
- A successful Context ID match.
- A successful VMID match.

*Breakpoint context comparisons* on page D2-1663 describes the requirements for a successful Context ID match and a successful VMID match.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

**0b1011, Linked Context ID and VMID Match breakpoint**

BT == 0b1011 is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- EL2 is not implemented.

For context-aware breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on all of the following:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page D2-1661.
  - A successful Context ID match defined by this breakpoint.
  - A successful VMID match defined by this breakpoint.
- Generation of a Watchpoint exception depends on all of the following:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page D2-1675.
  - A successful Context ID match defined by this breakpoint.
  - A successful VMID match defined by this breakpoint.

*Breakpoint context comparisons* on page D2-1663 describes the requirements for a successful Context ID match and a successful VMID match by this breakpoint.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

**Note**

See *Reserved DBGBCR<n>_EL1.BT values* on page D2-1663 for the behavior of breakpoints programmed with reserved BT values.

### D2.9.3 Execution conditions for which a breakpoint generates Breakpoint exceptions

Each breakpoint can be programmed so that it only generates Breakpoint exceptions for certain execution conditions. For example, a breakpoint might be programmed to generate Breakpoint exceptions only when the PE is executing at EL0 in Secure state.

DBGBCR<n>_EL1.{SSC, HMC, PMC} defines the execution conditions the breakpoint generates Breakpoint exceptions for, as follows:

**Security State Control, SSC**

Controls whether the breakpoint generates Breakpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

**Note**

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the breakpoint is set.

**Higher Mode Control, HMC, and Privileged Mode Control, PMC**

HMC and PMC together control which Exception levels the breakpoint generates Breakpoint exceptions in.

Table D2-11 on page D2-1661 shows the valid combinations of the values of HMC, SSC, and PMC, and for each combination shows which Exception levels breakpoints generate Breakpoint exceptions in.

In the table:

**Y or -** Means that a breakpoint programmed with the values of HMC, SSC and PMC shown in that row:

**Y** Can generate Breakpoint exceptions in that Exception level.

**-** Cannot generate Breakpoint exceptions in that Exception level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Res** Means that the combination of HMC, SSC, and PMC is reserved. See *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-1664.

**Table D2-11 Summary of breakpoint HMC, SSC, and PMC encodings**

| HMC | SSC | PMC | Security state the breakpoint is programmed to match in | EL3[a] | EL2 | EL1 | EL0 | Implementation | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | No EL3 | No EL3 and no EL2 |
| 0 | 00 | 01 | Both | - | - | Y | - | - | - |
| 0 | 00 | 10 | | - | - | - | Y | - | - |
| 0 | 00 | 11 | | - | - | Y | Y | - | - |
| 0 | 01 | 01 | Non-secure | - | - | Y | - | Res | Res |
| 0 | 01 | 10 | | - | - | - | Y | Res | Res |
| 0 | 01 | 11 | | - | - | Y | Y | Res | Res |
| 0 | 10 | 01 | Secure | - | - | Y | - | Res | Res |
| 0 | 10 | 10 | | - | - | - | Y | Res | Res |
| 0 | 10 | 11 | | - | - | Y | Y | Res | Res |
| 1 | 00 | 01 | Both | Y | Y | Y | - | - | Res |
| 1 | 00 | 11 | | Y | Y | Y | Y | - | Res |
| 1 | 01 | 01 | Non-secure | - | Y | Y | - | Res | Res |
| 1 | 01 | 11 | | - | Y | Y | Y | Res | Res |
| 1 | 10 | 00 | Secure | Y | - | - | - | Res | Res |
| 1 | 10 | 01 | | Y | - | Y | - | Res | Res |
| 1 | 10 | 11 | | Y | - | Y | Y | Res | Res |
| 1 | 11 | 00 | Non-secure | - | Y | - | - | - | Res |

a. Debug exceptions are not generated at EL3 using AArch64. This means that these combinations of HMC, SSC, and PMC are only relevant if breakpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PMC that generate Breakpoint exceptions at EL3 using AArch64.

All combinations of HMC, SSC, and PMC that this table does not show are reserved. See *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-1664.

### D2.9.4 Breakpoint instruction address comparisons

An address comparison is successful if bits [48:2] of the current instruction virtual address are equal to DBGBVR<n>_EL1[48:2].

**Note**

DBGBVR<n>_EL1 is a 64-bit register. The most significant bits of this register are sign-extension bits. DBGBVR<n>_EL1[1:0] are RES0 and are ignored.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Specifying the halfword-aligned address that an Address breakpoint matches on

For Address Match breakpoints, if the implementation supports AArch32 state, a debugger can program the *Byte Address Selection* field, DBGBCR<n>_EL1.BAS, so that the address comparison is successful on one of:

- The whole word starting at address DBGBVR<n>_EL1[48:2]:00.
- The halfword starting at address DBGBVR<n>_EL1[48:2]:00.
- The halfword starting at address ((DBGBVR<n>_EL1[48:2]:00) + 2).

This makes it possible to breakpoint on T32 instructions.

If EL1 is using AArch64 and EL0 is using AArch32, A32 and T32 instructions can be executed in stage 1 of an AArch64 translation regime. In this case, the instruction addresses are zero-extended before comparison with the breakpoint.

To breakpoint on an A64 instruction, ARM recommends that the debugger programs DBGBCR<n>_EL1.BAS so that the breakpoint generates Breakpoint exceptions on the whole word starting at address DBGBVR<n>_EL1[48:2]:00. The BAS field value for this is `0b1111`.

If the implementation is an AArch64-only implementation, all instructions are word-aligned and DBGBCR<n>_EL1.BAS is RES1.

Figure D2-3 shows a summary of when Address Match breakpoints programmed with particular BAS values generate Breakpoint exceptions. The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example, row 7 shows that a breakpoint with DBGBCR<n>_EL1.BAS programmed as either `0b0011` or `0b1111` generates Breakpoint exceptions for A64 instructions. A64 instructions are always at word-aligned addresses.

In the figure:

**Yes** Means that the breakpoint generates a Breakpoint exception.

**No** Means that the breakpoint does not generate a Breakpoint exception.

**UNP** Means that is it CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See *Other usage constraints for Address breakpoints* on page D2-1665.

| | Instruction set | Size | Location of instruction[a] -2 | -1 | 0 | +1 | +2 | +3 | +4 | +5 | BAS[3:0] 0b0011 | 0b1100 | 0b1111 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Row 1 | T32 | 16-bit | | | ■ | ■ | | | | | Yes | No | Yes |
| Row 2 | | 16-bit | | | | | ■ | ■ | | | No | Yes | UNP |
| Row 3 | T32 | 32-bit | ■ | ■ | ■ | ■ | | | | | UNP | No | UNP |
| Row 4 | | 32-bit | | | ■ | ■ | ■ | ■ | | | Yes | UNP | Yes |
| Row 5 | | 32-bit | | | | | ■ | ■ | ■ | ■ | No | Yes | UNP |
| Row 6 | A32 | 32-bit | | | ■ | ■ | ■ | ■ | | | Yes | UNP | Yes |
| Row 7 | A64 | 32-bit | | | ■ | ■ | ■ | ■ | | | Yes | UNP | Yes |

a. 0 means the word-aligned address held in the DBGBVR<n>_EL1[48:2]:00. The other locations are as follows:
- -2 means ((DBGBVR<n>_EL1[48:2]:00) – 2).
- -1 means ((DBGBVR<n>_EL1[48:2]:00) – 1).
- ...
- ...
- +5 means ((DBGBVR<n>_EL1[48:2]:00) + 5).

The solid areas show the location of the instruction.

**Figure D2-3 Summary of BAS field meanings for Address Match breakpoints**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D2.9.5 Breakpoint context comparisons

A context comparison is successful if, depending on the breakpoint type set by DBGBCR<n>_EL1.BT, one of the following is true:

- The current Context ID value is equal to DBGBVR<n>_EL1[31:0].
- The current VMID value is equal to DBGBVR<n>_EL1[39:32].
- The current Context ID value is equal to DBGBVR<n>_EL1[31:0], and the current VMID value is equal to DBGBVR<n>_EL1[39:32].

Context breakpoints do not generate Breakpoint exceptions when execution is in EL2 using either Execution state, or when execution is in EL3 using AArch64.

The following Context breakpoint types do not generate Breakpoint exceptions in Secure state:

- VMID Match breakpoints.
- VMID and Context ID Match breakpoints.

**Note**

- For all Context breakpoints, DBGBCR<n>_EL1.BAS is RES1 and is ignored.
- For Linked Context breakpoints, DBGBCR<n>_EL1.{LBN, SSC, HMC, PMC} are RES0 and are ignored.

### D2.9.6 Breakpoint usage constraints

See the following sections:

- *Reserved DBGBCR<n>_EL1.BT values*.
- *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-1664.
- *Reserved DBGBCR<n>_EL1.BAS values* on page D2-1664.
- *Reserved DBGBCR<n>_EL1.LBN values* on page D2-1665.
- *Other usage constraints for Address breakpoints* on page D2-1665.
- *Other usage constraints for Context breakpoints* on page D2-1665.

#### Reserved DBGBCR<n>_EL1.BT values

Table D2-12 shows when particular DBGBCR<n>_EL1.BT values are reserved.

**Table D2-12 Reserved BT values**

| BT value | Breakpoint type | Reserved |
|---|---|---|
| 0b001x | Context ID Match | For non context-aware breakpoints. |
| 0b010x | Address Mismatch | In stage 1 of an AArch64 translation regime, or if EDSCR.HDE is 1 and halting is allowed. |
| 0b011x | - | Always. |
| 0b100x | VMID Match | For non context-aware breakpoints, or if EL2 is not implemented. |
| 0b101x | Context ID and VMID Match | |
| 0b11xx | - | Always. |

If an enabled breakpoint is programmed with one of these reserved BT values:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BT value that is not reserved, other than for a direct read of DBGBCR<n>_EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For a direct read of DBGBCR<n>_EL1, if the reserved BT value:
  - Has no function for any execution conditions, the value read back is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the BT value so that the breakpoint functions for the other execution conditions.

The behavior of breakpoints with reserved BT values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

### Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values

Table D2-13 shows when particular combinations of DBGBCR<n>_EL1.{HMC, SSC, PMC} are reserved in stage 1 of an AArch64 translation regime.

**Table D2-13 Reserved HMC, SSC, and PMC combinations**

| HMC, SSC, and PMC combination | Reserved |
|---|---|
| All combinations with SSC set to `0b01` or `0b10`. | When EL3 is not implemented and EL2 is implemented. |
| Any combination where HMC or SSC is nonzero. | When both of EL2 and EL3 are not implemented. |
| Combinations not included in Table D2-11 on page D2-1661. | Always |

For all breakpoints except Linked Context breakpoints, if an enabled breakpoint is programmed with one of these reserved combinations:

- If the reserved combination has a function for other execution conditions:
  - The breakpoint must behave as if it is disabled.
  - A direct read of DBGBCR<n>_EL1.{HMC, SSC, PMC} returns the values written. This means that software can save and restore the combination so that the breakpoint can function for the other execution conditions.
- If the reserved combination does not have a function for other execution conditions:
  - The breakpoint must behave either as if it is programmed with a combination that is not reserved or as if it is disabled.
  - A direct read of DBGBCR<n>_EL1.{HMC, SSC, PMC} returns UNKNOWN values.

Linked Context breakpoints ignore the values of HMC, SSC, and PMC. See *Other usage constraints for Context breakpoints* on page D2-1665.

The behavior of breakpoints with reserved combinations of HMC, SSC, and PMC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

### Reserved DBGBCR<n>_EL1.BAS values

In an AArch64-only implementation, DBGBCR<n>_EL1.BAS for all breakpoints is RES1.

Otherwise:

- For all Context breakpoints, DBGBCR<n>_EL1.BAS is RES1 and is ignored.
- For all Address breakpoints:
  - BAS[2, 0] are RW.
  - BAS[3] is a RO copy of BAS[2], and BAS[1] is a RO copy of BAS[0].
  - However, ARM recommends that software programs the BAS field with a valid value, meaning a value where BAS[3] is the same as BAS[2], and BAS[1] is the same as BAS[0].


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Reserved DBGBCR<n>_EL1.LBN values

A Linked Address breakpoint must link to a context-aware breakpoint. For a Linked Address breakpoint, any DBGBCR<n>_EL1.LBN value that is not for a context-aware breakpoint is reserved.

### Other usage constraints for Address breakpoints

For all Address breakpoints:

- DBGBVR<n>_EL1[1:0] are RES0 and are ignored.
- If the implementation supports AArch32 state:
  - For 32-bit instructions, if a breakpoint matches on the address of the second halfword but not the address of the first halfword, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception.
  - If DBGBCR<n>.BAS is `0b1111`, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception for a T32 instruction starting at address ((DBGBVR<n>[48:2]:00) + 2). For T32 instructions, ARM recommends that the debugger programs the BAS field with either `0b0011` or `0b1100`.

For Unlinked Address breakpoints, DBGBCR<n>_EL1.LBN reads UNKNOWN and its value is ignored.

For Linked Address breakpoints:

- If a Linked Address breakpoint links to a breakpoint that is not implemented, or that is not context-aware, behavior is CONSTRAINED UNPREDICTABLE. The Linked Address breakpoint behaves as if it is either:
  - Disabled, and DBGBCR<n>_EL1.LBN for it reads UNKNOWN.
  - Links to an UNKNOWN context-aware breakpoint. In this case, it generates Breakpoint exceptions and DBGBCR<n>_EL1.LBN indicates which context-aware breakpoint it has linked to.
- If a Linked Address breakpoint links to a breakpoint that is implemented and that is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

### Other usage constraints for Context breakpoints

For all Context breakpoints:

- Any bits of DBGBVR<n>_EL1 that are not used to specify Context ID or VMID are RES0 and are ignored.
- DBGBCR<n>_EL1.LBN reads UNKNOWN and its value is ignored.

For Linked Context breakpoints:

- DBGBCR<n>_EL1.{LBN, SSC, HMC, PMC} are RES0 and are ignored.
- If no Linked Address breakpoints or Linked watchpoints link to a Linked Context breakpoint, the Linked Context breakpoint does not generate any Breakpoint exceptions.

## D2.9.7 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information*.
- *Preferred return address* on page D2-1666.

### Exception syndrome information

On taking a Breakpoint exception, the PE records information about the exception in the *Exception Syndrome Register* (ESR) at the Exception level the exception is taken to. The ESR used is one of:

- ESR_EL1.
- ESR_EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Breakpoint exceptions cannot be taken to EL3 using AArch64.

Table D2-14 shows the information that the PE records.

**Table D2-14 Information recorded in the ESR_ELx**

| ESR_ELx field | Information recorded in ESR_EL1 or ESR_EL2. | |
|---|---|---|
| *Exception Class*, EC | The PE sets this to:<br>• `0x30`, if the exception was taken from a lower Exception level.<br>• `0x31`, if the exceptions was taken without a change of Exception level. | |
| *Instruction Length*, IL | The PE sets this to 1. | |
| *Instruction Specific Syndrome*, ISS | **ISS[24:6]** | RES0. |
| | **ISS[5:0]** | *Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`. |

### Preferred return address

The preferred return address of a Breakpoint exception is the address of the instruction that was not executed because the PE took the Breakpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

## D2.9.8 Pseudocode description of Breakpoint exceptions taken from AArch64 state

`AArch64.BreakpointValueMatch()` tests the value in DBGBVR<n>_EL1.

```
// AArch64.BreakpointValueMatch()
// ==============================

boolean AArch64.BreakpointValueMatch(integer n, bits(64) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //    matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.

    // If a non-existant breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(ID_AA64DFR0_EL1.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(ID_AA64DFR0_EL1.BRPs));
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking.)
    if DBGBCR_EL1[n].E == '0' then return FALSE;

    context_aware = (n >= UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    type = DBGBCR_EL1[n].BT;
    if (type == 'x1xx' ||                                           // Reserved
          (type != '0x0x' && !context_aware) ||                     // Context matching
          (type == '1xxx' && !HaveEL(EL2))) then                    // VMID match
        (c, type) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Determine what to compare against.
    match_addr = type == '0x0x';
    match_vmid = type == '10xx';
    match_cid  = type == 'x01x';
    linked     = type == 'xxx1';

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return FALSE;

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return FALSE;

    // Do the comparison.
    if match_addr then
        byte = UInt(vaddress<1:0>);
        if HaveAnyAArch32() then
            // T32 instructions can be executed at EL0 in an AArch64 translation regime.
            assert byte IN {0,2};                 // "vaddress" is halfword aligned.
            byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0;                     // "vaddress" is word aligned
            byte_select_match = TRUE;             // DBGBCR_EL1[n].BAS<byte> is RES1
        top = AddrTop(vaddress);
        BVR_match = vaddress<top:2> == DBGBVR_EL1[n]<top:2> && byte_select_match;
    elsif match_cid then
        BVR_match = (PSTATE.EL IN {EL0,EL1} && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
    if match_vmid then
        BXVR_match = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                      VTTBR_EL2.VMID == DBGBVR_EL1[n]<39:32>);

    match = (!match_vmid || BXVR_match) && (!(match_addr || match_cid) || BVR_match);
    return match;
```

`AArch64.StateMatch()` tests the values in DBGBCR<n>_EL1.{SSC, HMC, PMC} and, if the breakpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

For a watchpoint, `AArch64.StateMatch()` tests the values in DBGWCR<n>_EL1.{SSC, HMC, PAC} and, if the watchpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

```
// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean ispriv)
    // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
    // "linked" is TRUE if this is a linked breakpoint/watchpoint type.

    // If parameters are set to a reserved type, behaves either as disabled or not a reserved-type
    if ((HMC:SSC:PxC) IN {'0xx00','011xx','100x0','101x0','11010','11101','1111x'} || // Reserved
          (SSC IN {'01','10'} && !HaveEL(EL3)) ||                                     // No EL3
          (HMC:SSC != '000' && !HaveEL(EL3) && !HaveEL(EL2))) then                    // No EL3/EL2
        (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
    EL2_match = HaveEL(EL2) && HMC == '1';
    EL1_match = PxC<0> == '1';
    EL0_match = PxC<1> == '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    case PSTATE.EL of
        when EL3  priv_match = EL3_match;
        when EL2  priv_match = EL2_match;
        when EL1  priv_match = if ispriv then EL1_match else EL0_match;
        when EL0  priv_match = EL0_match;

    case SSC of
        when '00'  security_state_match = TRUE;         // Both
        when '01'  security_state_match = !IsSecure();  // Non-secure only
        when '10'  security_state_match = IsSecure();   // Secure only
        when '11'  security_state_match = TRUE;         // Both

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));
        last_ctx_cmp = UInt(ID_AA64DFR0_EL1.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(64) UNKNOWN;
        linked_to = TRUE;
        linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

`AArch64.BreakpointMatch()` tests a committed instruction against all breakpoints.

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.BRPs);

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT == '0x01';
    linked_to = FALSE;

    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                     linked, DBGBCR_EL1[n].LBN, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAnyAArch32() && size == 4 then                 // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool();

    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    match = value_match && state_match && enabled;

    return match;
```

AArch64.CheckBreakpoint() generates a Breakpoint exception if all of the following are true:

- MDSCR_EL1.MDE is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level and Security state* on page D2-1643.
- All of the conditions required for Breakpoint exception generation are met. See *About Breakpoint exceptions* on page D2-1653.

**Note**

AArch64.CheckBreakpoint() might halt the PE and cause it to enter Debug state. External debug uses Debug state.

```
// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    match = FALSE;

    for i = 0 to UInt(ID_AA64DFR0_EL1.BRPs)
        match_i = AArch64.BreakpointMatch(i, vaddress, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

AArch64.BreakpointException() is called to generate a Breakpoint exception.

```
// AArch64.BreakpointException()
// =============================

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D2.10 Watchpoint exceptions

This section describes Watchpoint exceptions in stage 1 of an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing at either:

- An Exception level that is using AArch64.
- EL0 using AArch32 when EL1 is using AArch64.

It contains the following subsections:

- *About Watchpoint exceptions*.
- *Watchpoint types and linking of watchpoints* on page D2-1672.
- *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page D2-1673.
- *Watchpoint data address comparisons* on page D2-1675.
- *Determining the memory location that caused a Watchpoint exception* on page D2-1678.
- *Watchpoint behavior on other instructions* on page D2-1679.
- *Usage constraints* on page D2-1680.
- *Exception syndrome information and preferred return address* on page D2-1681.
- *Pseudocode description of Watchpoint exceptions taken from AArch64 state* on page D2-1682.

### D2.10.1 About Watchpoint exceptions

A *watchpoint* is a debug event that results from the execution of an instruction, based on a data address. Watchpoints are also known as *data breakpoints*.

A watchpoint operates as follows:

1. A debugger programs the watchpoint with a data address, or a data address range.
2. The watchpoint generates a *Watchpoint debug event* on an access to the address, or any address in the address range.

A watchpoint never generates a Watchpoint debug event on an instruction fetch.

An implementation can include between 2-16 watchpoints. In an implementation, ID_AA64DFR0_EL1.WRPs shows how many are implemented.

To use an implemented watchpoint, a debugger programs the following registers for the watchpoint:

- The *Watchpoint Control Register*, DBGWCR<n>_EL1. This contains controls for the watchpoint, for example an enable control.
- The *Watchpoint Value Register*, DBGWVR<n>_EL1. This holds the data virtual address used for watchpoint matching.

These registers are numbered, so that:

- DBGWCR1_EL1 and DBGWVR1_EL1 are for watchpoint number one.
- DBGWCR2_EL2 and DBGWVR2_EL1 are for watchpoint number two.
- …
- …
- DBGWCR<n>_EL1 and DBGWVR<n>_EL1 are for watchpoint number n.

A watchpoint can:

- Be programmed to generate Watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
- Link to a *Linked Context breakpoint*, so that a Watchpoint debug event is only generated if the PE is in a particular context when the address match occurs.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A single watchpoint can be programmed to match on one or more address bytes. A watchpoint generates a Watchpoint debug event on an access to any byte that it is watching. The number of bytes a watchpoint is watching is either:

- One to eight bytes, provided that these bytes are contiguous and that they are all in the same naturally-aligned doubleword. A debugger uses the *Byte Address Select* field, DBGWCR<n>_EL1.BAS, to select the bytes. See *Programming a watchpoint with eight bytes or fewer* on page D2-1676.
- Eight bytes to 2GB, provided that both of the following are true:
  - The number of bytes is a power-of-two.
  - The range starts at an address that is aligned to the range size.

  A debugger uses the *MASK* field, DBGWCR<n>_EL1.MASK, to program a watchpoint with eight bytes to 2GB. See *Programming a watchpoint with eight or more bytes* on page D2-1677.

A debugger must use either the BAS field or the MASK field. If it uses both, whether the watchpoint generates Watchpoint debug events is CONSTRAINED UNPREDICTABLE. See *Programming dependencies of the BAS and MASK fields* on page D2-1680.

For each memory access, all of the watchpoints are tested. When a watchpoint is tested, it generates a Watchpoint debug event if all of the following are true:

- The watchpoint is enabled. That is, the watchpoint enable control for it, DBGWCR<n>_EL1.E, is 1.
- The conditions specified in the DBGWCR<n>_EL1 are met.
- The comparison with the address held in the DBGWVR<n>_EL1 is successful.
- If the watchpoint links to a Linked Context breakpoint, the comparison or comparisons made by the Linked Context breakpoint also are successful. See Figure D2-2 on page D2-1657. See also *Breakpoint context comparisons* on page D2-1663.
- The instruction that initiates the memory access is committed for execution.
- The instruction that initiates the memory access passes its condition code check.

If halting is allowed and EDSCR.HDE is 1, Watchpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:
- Enabled, Watchpoint debug events generate Watchpoint exceptions.
- Disabled, Watchpoint debug events are ignored.

**Note**

The remainder of this Watchpoint Exceptions section, including all subsections, describes watchpoints as generating Watchpoint exceptions.

However, the behavior described also applies if watchpoints are causing entry to Debug state.

*The debug exception enable controls* on page D2-1640 describes the enable controls for Watchpoint debug events.

### D2.10.2 Watchpoint types and linking of watchpoints

When a debugger programs a watchpoint, it must program that watchpoint so that it is either:
- Used in isolation. In this case the watchpoint is called an *Unlinked watchpoint*.
- Enabled for linking to a Linked Context breakpoint. In this case the watchpoint is called a *Linked watchpoint*.

When a Linked watchpoint links to a Linked Context breakpoint, the Linked watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs. For example, a debugger might:

1. Program watchpoint number one with a data address.
2. Program breakpoint number five to be a *Linked VMID Match breakpoint*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

3. Link the watchpoint and the breakpoint together. A Watchpoint exception is only generated if both the data address matches and the VMID matches.

The *Watchpoint Type* field for a watchpoint, DBGWCR<n>_EL1.WT, controls whether the watchpoint is enabled for linking. If DBGWCR<n>_EL1.WT is 1, the watchpoint is enabled for linking.

### Rules for linking watchpoints

The rules for watchpoint linking are as follows:

- Only Linked watchpoints can be linked.
- A Linked watchpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, DBGWCR<n>_EL1.LBN, for the Linked watchpoint specifies the particular Linked Context breakpoint that the Linked watchpoint links to, and:
  - DBGWCR<n>_EL1.WT.{SSC, HMC, PAC} for the Linked watchpoint defines the execution conditions that the watchpoint generates Watchpoint exceptions for. See *Execution conditions for which a watchpoint generates Watchpoint exceptions*.
  - DBGBCR<n>_EL1.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- A Linked watchpoint cannot link to another watchpoint. The LBN field can therefore only specify a breakpoint.
- If a Linked watchpoint links to a breakpoint that is not context-aware, the behavior of the Linked watchpoint is CONSTRAINED UNPREDICTABLE. See *Usage constraints* on page D2-1680.
- If a Linked watchpoint links to an Unlinked Context breakpoint, the Linked watchpoint never generates any Watchpoint exceptions.
- Multiple Linked watchpoints can link to a single Linked Context breakpoint.

**Note**

Multiple Address breakpoints can also link to a single Linked Context breakpoint. *Breakpoint exceptions* on page D2-1653 describes breakpoints.

Figure D2-2 on page D2-1657 shows an example of permitted watchpoint linking.

## D2.10.3 Execution conditions for which a watchpoint generates Watchpoint exceptions

Each watchpoint can be programmed so that it only generates Watchpoint exceptions for certain execution conditions. For example, a watchpoint might be programmed to generate Watchpoint exceptions only when the PE is executing at EL0 in Secure state.

DBGWCR<n>_EL1.{SSC, HMC, PAC} define the execution conditions a watchpoint generates Watchpoint exceptions for, as follows:

**Security State Control, SSC**

Controls whether the watchpoint generates Watchpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

**Note**

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the watchpoint is set.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Higher Mode Control, HMC, and Privileged Access Control, PAC**

HMC and PAC together control which Exception levels the watchpoint generates Watchpoint exceptions in.

**Note**

PAC controls which access privilege the watchpoint matches. This means that if the PE is executing an unprivileged load/store instruction at EL1, the data access might trigger a watchpoint that is programmed to match on EL0 accesses.

Table D2-15 shows the valid combinations of HMC, SSC, and PAC, and for each combination shows which Exception levels watchpoints generate Watchpoint exceptions in.

In the table:

**Y or -** Means that a watchpoint programmed with the values of HMC, SSC, and PAC shown in that row:

**Y** Can generate Watchpoint exceptions in that Exception level.

**-** Cannot generate Watchpoint exceptions in that Exception level.

**Res** Means that the combination of HMC, SSC, and PAC is reserved. See *Reserved DBGWCR<n>_EL1.{HMC, SSC, PAC} values* on page D2-1680.

**Table D2-15 Summary of watchpoint HMC, SSC, and PAC encodings**

| HMC | SSC | PAC | Security state the watchpoint is programmed to match in | EL3[a] | EL2 | EL1 | EL0 | Implementation | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | No EL3 | No EL3 and no EL2 |
| 0 | 00 | 01 | Both | - | - | Y | - | - | - |
| 0 | 00 | 10 | | - | - | - | Y | - | - |
| 0 | 00 | 11 | | - | - | Y | Y | - | - |
| 0 | 01 | 01 | Non-secure | - | - | Y | - | Res | Res |
| 0 | 01 | 10 | | - | - | - | Y | Res | Res |
| 0 | 01 | 11 | | - | - | Y | Y | Res | Res |
| 0 | 10 | 01 | Secure | - | - | Y | - | Res | Res |
| 0 | 10 | 10 | | - | - | - | Y | Res | Res |
| 0 | 10 | 11 | | - | - | Y | Y | Res | Res |
| 1 | 00 | 01 | Both | Y | Y | Y | - | - | Res |
| 1 | 00 | 11 | | Y | Y | Y | Y | - | Res |
| 1 | 01 | 01 | Non-secure | - | Y | Y | - | Res | Res |
| 1 | 01 | 11 | | - | Y | Y | Y | Res | Res |
| 1 | 10 | 00 | Secure | Y | - | - | - | Res | Res |
| 1 | 10 | 01 | | Y | - | Y | - | Res | Res |
| 1 | 10 | 11 | | Y | - | Y | Y | Res | Res |
| 1 | 11 | 00 | Non-secure | - | Y | - | - | - | Res |

a. Debug exceptions are not generated at EL3 using AArch64. This means that these combinations of HMC, SSC, and PAC are only relevant if watchpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PMC that generate Watchpoint exceptions at EL3 using AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All combinations of HMC, SSC, and PAC that this table does not show are reserved. See *Reserved DBGWCR<n>_EL1.{HMC, SSC, PAC} values* on page D2-1680.

### D2.10.4 Watchpoint data address comparisons

An address comparison is successful if bits [48:2] of the current data virtual address are equal to DBGWVR<n>_EL1[48:2], taking into account all of the following:

- The size of the access. See *Size of the data access*.

  If EL1 is using AArch64 and EL0 is using AArch32, AArch32 instructions can be executed in stage 1 of an AArch64 translation regime. In this case, data addresses are zero-extended before comparison with the watchpoint.

- The bytes selected by DBGWVR<n>_EL1.BAS. See *Programming a watchpoint with eight bytes or fewer* on page D2-1676.

- Any address ranges indicated by DBGWVR<n>_EL1.MASK. See *Programming a watchpoint with eight or more bytes* on page D2-1677.

**Note**

- DBGWVR<n>_EL1 is a 64-bit register. The most significant bits of this register are sign-extension bits.
- DBGWVR<n>_EL1[1:0] are RES0 and are ignored

#### Size of the data access

Because watchpoints can be programmed to generate Watchpoint exceptions on individual bytes, the size of each data access must be taken into account. See Example D2-1.

**Example D2-1**

1. A debugger programs a watchpoint to generate Watchpoint exceptions only when the byte at address `0x1009` is accessed.
2. The PE accesses the unaligned doubleword starting at address `0x1003`.

In this scenario, the watchpoint must generate a Watchpoint exception.

The size of data accesses initiated by `DC ZVA` instructions is the DC ZVA block size that DCZID_EL0.BS defines.

The size of data accesses initiated by `DC IVAC` instructions is an IMPLEMENTATION DEFINED size that is both:

- From the inclusive range between:
  - The size that CTR_EL0.DminLine defines.
  - 2KB.
- A power-of-two.

For both of these instructions:

- The lowest address accessed by the instruction is the address supplied to the instruction, rounded down to the nearest multiple of the access size initiated by that instruction.
- The highest address accessed is (size - 1) bytes above the lowest address accessed.

See also, *Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA instruction* on page D2-1679.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Programming a watchpoint with eight bytes or fewer

The Byte Address Select field, DBGWCR<n>_EL1.BAS, selects which bytes in the doubleword starting at the address contained in the DBGWVR<n>_EL1 the watchpoint generates Watchpoint exceptions for.

If the address programmed into the DBGWVR<n>_EL1 is:

- Doubleword-aligned:
  - All eight bits of DBGWCR<n>_EL1.BAS are used, and the descriptions given in Table D2-16 apply.
- Word-aligned but not doubleword-aligned:
  - Only DBGWCR<n>_EL1.BAS[3:0] are used, and the descriptions given in Table D2-17 apply. In this case, DBGWCR<n>_EL1.BAS[7:4] are RES0.

**Table D2-16 Supported BAS values when the DBGWVRn_EL1 address alignment is doubleword**

| BAS value | Description |
|---|---|
| 0b00000000 | Watchpoint never generates a Watchpoint exception. |
| BAS[0] == 1 | Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[48:3]:000 is accessed. |
| BAS[1] == 1 | Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[48:3]:001 is accessed. |
| BAS[2] == 1 | Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[48:3]:010 is accessed. |
| BAS[3] == 1 | Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[48:3]:011 is accessed. |
| BAS[4] == 1 | Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[48:3]:100 is accessed. |
| BAS[5] == 1 | Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[48:3]:101 is accessed. |
| BAS[6] == 1 | Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[48:3]:110 is accessed. |
| BAS[7] == 1 | Generates a Watchpoint exception if the byte at address DBGWVR<n>_EL1[48:3]:111 is accessed. |

**Table D2-17 Supported BAS values when the DBGWVRn_EL1 address alignment is word**

| BAS value[a] | Description |
|---|---|
| 0b00000000 | Watchpoint never generates a Watchpoint exception |
| BAS[0] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>_EL1[48:2]:00 is accessed. |
| BAS[1] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>_EL1[48:2]:01 is accessed. |
| BAS[2] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>_EL1[48:2]:10 is accessed. |
| BAS[3] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>_EL1[48:2]:11 is accessed. |

a. DBGWCR<n>_EL1.BAS[7:4] are RES0.

If the BAS field is programmed with more than one byte, the bytes that it is programmed with must be contiguous. For watchpoint behavior when its BAS field is programmed with non-contiguous bytes, see *Other usage constraints on page D2-1681*.

When programming the BAS field with anything other than 0b11111111, a debugger must program DBGWCR<n>_EL1.MASK to be 0b00000. See *Programming dependencies of the BAS and MASK fields on page D2-1680*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A watchpoint generates a Watchpoint exception whenever a watched byte is accessed, even if:

- The access size is smaller or larger than the address region being watched.
- The access is misaligned, and the base address of the access is not in the doubleword or word of memory addressed by the DBGWVR<n>_EL1[48:3]. See Example D2-1 on page D2-1675.

The following are some example configurations of the BAS field:

- To program a watchpoint to generate a Watchpoint exception on the byte at address `0x1003`, program:
  - DBGWVR<n>_EL1 with `0x1000`.
  - DBGWCR<n>_EL1.BAS to be `0b00001000`.
- To program a watchpoint to generate a Watchpoint exception on the bytes at addresses `0x2003`, `0x2004` and `0x2005`, program:
  - DBGWVR<n>_EL1 with `0x2000`.
  - DBGWCR<n>_EL1.BAS to be `0b00111000`.
- If the address programmed into the DBGWVR<n>_EL1 is doubleword-aligned:
  - To generate a Watchpoint exception when any byte in the word starting at the doubleword-aligned address is accessed, program DBGWCR<n>_EL1.BAS to be `0b00001111`.
  - To generate a Watchpoint exception when any byte in the word starting at address DBGWVR<n>_EL1[31:3]:100 is accessed, program DBGWCR<n>_EL1.BAS to be `0b11110000`.

**Note**

ARM deprecates programming a DBGWVR<n>_EL1 with an address that is not doubleword-aligned.

### Programming a watchpoint with eight or more bytes

A debugger can use the *MASK* field, DBGWCR<n>_EL1.MASK, to program a single watchpoint with a data address range. The range must meet all of the following criteria:

- It is a size that is:
  - A power-of-two.
  - A minimum of eight bytes.
  - A maximum of 2GB.
- It starts at an address that is aligned to the size.

The MASK field specifies the number of least significant data address bits that must be masked. Up to 31 least significant bits can be masked:

| | | |
|---|---|---|
| **MASK** | `0b00000` | No bits are masked. |
| | `0b00001` | Reserved. |
| | `0b00010` | Reserved. |
| | `0b00011` | Three least significant bits are masked. |
| | `0b00100` | Four least significant bits are masked. |
| | `0b00101` | Five least significant bits are masked. |
| | **...** | … |
| | `0b11111` | 31 least significant bits are masked. |

If *n* least significant address bits are masked, the watchpoint generates a Watchpoint exception on all of the following:

- Address DBGWVR<n>_EL1[48:*n*]:000…
- Address DBGWVR<n>_EL1[48:*n*]:111…
- Any address between these two addresses.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For example, if the four least significant address bits are masked, Watchpoint exceptions are generated for all addresses between DBGWVR<n>_EL1[48:4]:0000 and DBGWVR<n>_EL1[48:4]:1111, including these addresses.

**Note**

- The 17 most significant bits cannot be masked. This means that the full address cannot be masked.
- For watchpoint behavior when its MASK field is programmed with a reserved value, see *Reserved DBGWCR<n>_EL1.MASK values* on page D2-1681.

When masking address bits, a debugger must both:

- Program DBGWCR<n>_EL1.BAS to be `0b11111111`. See *Programming dependencies of the BAS and MASK fields* on page D2-1680.
- In the DBGWVR<n>_EL1, set the masked address bits to 0. For watchpoint behavior when any of the masked address bits are not 0, see *Other usage constraints* on page D2-1681.

### D2.10.5 Determining the memory location that caused a Watchpoint exception

On taking a Watchpoint exception, the PE records an address in a *Fault Address Register* that the debugger can use to determine the memory location that triggered the watchpoint.

The Fault Address Register (FAR) used is either:

- FAR_EL1, if the exception is taken to EL1.
- FAR_EL2, if the exception is taken to EL2.

In cases where one instruction triggers multiple watchpoints, only one address is recorded.

On entering Debug state on a Watchpoint debug event, the PE records the address in the EDWAR.

For more information, see the subsections that follow. These are:

- *Address recorded for Watchpoint exceptions generated by instructions other than Data Cache instructions*
- *Address recorded for Watchpoint exceptions generated by Data Cache instructions* on page D2-1679

#### Address recorded for Watchpoint exceptions generated by instructions other than Data Cache instructions

The address recorded must be both:

- From the inclusive range between:
  - The lowest address accessed by the memory access that triggered the watchpoint.
  - The highest *watchpointed address* accessed by the memory access. A watchpointed address is an address that the watchpoint is watching.
- Within a naturally-aligned block of memory that is all of the following:
  - A power-of-two size.
  - No larger than the `DC ZVA` block size.
  - Contains a watchpointed address accessed by the memory access.

  The size of the block is IMPLEMENTATION DEFINED. There is no architectural means of discovering the size.

**Example D2-2 Address recorded for a watchpoint programmed on `0x8019`**

A debugger programs a watchpoint to generate a Watchpoint exception on any access to the byte `0x8019`.

An A32 load multiple instruction then loads nine registers starting from address `0x8004` upwards. This triggers the watchpoint.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the DC ZVA block size is:

- 32 bytes, the address that the PE records must be between 0x8004 and 0x8019 inclusive.
- 16 bytes, the address that the PE records must be between 0x8010 and 0x8019 inclusive.

---

#### Address recorded for Watchpoint exceptions generated by Data Cache instructions

The address recorded is the address passed to the instruction. This means that the address recorded might be higher than the address of the location that triggered the watchpoint.

### D2.10.6 Watchpoint behavior on other instructions

Under normal operating conditions, the following do not generate Watchpoint exceptions:

- Instruction cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.
- Prefetch memory instructions.
- All data cache maintenance instructions except DC IVAC.

However, the debug architecture allows for IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to enable checking for Watchpoint debug events on an IMPLEMENTATION DEFINED subset of these instructions. Whether a watchpoint treats the instruction as a load or a store, and the access size of instruction cache, address translation, and TLB operations are IMPLEMENTATION DEFINED.

**Note**

The DC ZVA instruction is not a data cache maintenance instruction.

---

See also the following subsections:

- *Watchpoint behavior on accesses by Store-Exclusive instructions*.
- *Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA instruction*.

#### Watchpoint behavior on accesses by Store-Exclusive instructions

If a watchpoint matches on a data access caused by a Store-Exclusive instruction, then:

- If the store fails because an exclusive monitor does not permit it, it is IMPLEMENTATION DEFINED whether the watchpoint generates a Watchpoint exception.
- Otherwise, the watchpoint generates a Watchpoint exception.

#### Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA instruction

DC IVAC and DC ZVA operations are treated as data stores. This means that for a watchpoint to match on an access caused by one of these instructions, the debugger must program DBGWCR<n>_EL1.LSC to be one of the following:

**10** Match on data stores.

**11** Match on data stores and data loads.

**Note**

For the size of data accesses performed by the DC IVAC instruction and the DC ZVA instruction, see *Watchpoint data address comparisons* on page D2-1675. The size of all data accesses must be considered because watchpoints can be programmed to match on individual bytes.

---


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D2.10.7 Usage constraints

See the following:

- *Reserved DBGWCR<n>_EL1.{HMC, SSC, PAC} values*.
- *Reserved DBGWCR<n>_EL1.LBN values*.
- *Programming dependencies of the BAS and MASK fields*.
- *Reserved DBGWCR<n>_EL1.BAS values* on page D2-1681.
- *Reserved DBGWCR<n>_EL1.MASK values* on page D2-1681.
- *Other usage constraints* on page D2-1681.

#### Reserved DBGWCR<n>_EL1.{HMC, SSC, PAC} values

Table D2-18 shows when particular combinations of DBGWCR<n>_EL1.{HMC, SSC, PAC} are reserved.

**Table D2-18 Reserved HMC, SSC, and PAC combinations**

| HMC, SSC, and PMC combination | Reserved |
|---|---|
| All combinations with SSC set to `0b01` or `0b10`. | When EL3 is not implemented and EL2 is implemented. |
| All combinations where HMC or SSC is nonzero, and all combinations with PAC set to `0b00`. | When both of EL2 and EL3 are not implemented. |
| Combinations not included in Table D2-15 on page D2-1674. | Always |

If an enabled watchpoint is programmed with one of these reserved combinations:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with a combination that is not reserved, other than for a direct read of DBGWCR<n>_EL1.
- For a direct read of DBGWCR<n>_EL1, if the reserved combination:
  - Has no function for any execution conditions, the value read back for each of HMC, SSC, and PMC is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the combination so that the watchpoint functions for the other execution conditions.

The behavior of watchpoints with reserved combinations of HMC, SSC, and PAC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

#### Reserved DBGWCR<n>_EL1.LBN values

A Linked watchpoint must link to a context-aware breakpoint. For a Linked watchpoint, any DBGWCR<n>_EL1.LBN value that is not for a context-aware breakpoint is reserved.

#### Programming dependencies of the BAS and MASK fields

When programming a watchpoint, a debugger must use either:

- The MASK field, to program the watchpoint with an address range that can be eight bytes to 2GB.
- The BAS field, to select which bytes in the doubleword or word starting at the address contained in the DBGWVR<n>_EL1 the watchpoint must generate Watchpoint exceptions for.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the debugger uses the:

- MASK field, it must program BAS to be `0b11111111`, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be `0b00000`, so that the MASK field does not indicate any address ranges.

If the debugger uses both of these fields, behavior of the watchpoint is CONSTRAINED UNPREDICTABLE. Either:

- The watchpoint treats the MASK field as if it is programmed with `0b00000`. In this case, the watchpoint is programmed with a single address and it generates Watchpoint exceptions for the bytes that the BAS field indicates.
- For each byte in the masked region, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception.

#### Reserved DBGWCR<n>_EL1.BAS values

If DBGWVR<n>_EL1[2] is 1, DBGWCR<n>_EL1.BAS[7:4] are RES0 and are ignored.

#### Reserved DBGWCR<n>_EL1.MASK values

If DBGWCR<n>_EL1.MASK is programmed with a reserved value, the watchpoint must behave as if it is either:

- Disabled.
- Programmed with an UNKNOWN value that is not reserved, that might be `0b00000`.

#### Other usage constraints

For all watchpoints:

- DBGWVR<n>_EL1[1:0] are RES0 and are ignored.
- If DBGWCR<n>_EL1.BAS is programmed with non-contiguous bytes of memory, it is CONSTRAINED UNPREDICTABLE whether the Watchpoint generates a Watchpoint exception for each byte in the doubleword or word of memory addressed by the DBGWVR<n>_EL1.
- If DBGWCR<n>_EL1.MASK is nonzero, and any masked bits of DBGWVR<n>_EL1 are not 0, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception when the unmasked bits match.
- A watchpoint never generates any Watchpoint exceptions if DBGWCR<n>_EL1.LSC is `0b00`.

For Unlinked watchpoints, DBGWCR<n>_EL1.LBN reads UNKNOWN and its value is ignored.

For Linked watchpoints:

- If a Linked watchpoint links to a breakpoint that is not implemented, or that is not context-aware, behavior is CONSTRAINED UNPREDICTABLE. The Linked watchpoint behaves as if it is either:
  - Disabled, and DBGWCR<n>_EL1.LBN for it reads UNKNOWN.
  - Links to an UNKNOWN context-aware breakpoint. In this case, it generates Watchpoint exceptions and DBGWCR<n>_EL1.LBN indicates which context-aware breakpoint it has linked to.
- If a Linked watchpoint links to a breakpoint that is implemented and is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

### D2.10.8 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information* on page D2-1682.
- *Preferred return address* on page D2-1682.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Exception syndrome information

On taking a Watchpoint exception, the PE records all of the following:

- Information about the exception in the *Exception Syndrome Register* (ESR) at the Exception level the exception is taken to.
- An address that the debugger can use to determine the memory location that caused the exception. The PE records this in a *Fault Address Register* (FAR).

The ESR and FAR used is either:

- ESR_EL1 and FAR_EL1, if the exception is taken to EL1.
- ESR_EL2 and FAR_EL2, if the exception is taken to EL2.

**Note**

Watchpoint exceptions cannot be taken to EL3 using AArch64.

Table D2-19 shows the recorded information.

**Table D2-19 Information recorded in the ESR_ELx**

| ESR_ELx field | Information recorded in ESR_EL1 or ESR_EL2 | |
|---|---|---|
| *Exception Class*, EC | This is set to:<br>• `0x34`, if the exception was taken from a lower Exception level.<br>• `0x35`, if the exception was taken without a change of Exception level. | |
| *Instruction Length*, IL | This is set to 1. | |
| *Instruction Specific Syndrome*, ISS | **ISS[24]** | *Instruction Syndrome Valid* (ISV). This is 0, because Watchpoint exceptions are not stage 2 aborts. |
| | **ISS[23:9]** | RES0. |
| | **ISS[8]** | *Cache Maintenance* (CM). This indicates whether a cache maintenance instruction generated the exception:<br>**0** Not generated by a cache maintenance instruction.<br>**1** Generated by a cache maintenance instruction.<br>If a `DC ZVA` instruction generated the exception, CM is 0. |
| | **ISS[7]** | RES0. |
| | **ISS[6]** | *Write-not-Read* (WnR). This indicates whether the access was by a read instruction or a write instruction:<br>**0** Read instruction.<br>**1** Write instruction. |
| | **ISS[5:0]** | *Data Fault Status Code* (DFSC). The PE sets this to the code for a debug exception, `0b100010`. |

### Preferred return address

The preferred return address of a Watchpoint exception is the address of the instruction that was not executed because the PE took the Watchpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

## D2.10.9 Pseudocode description of Watchpoint exceptions taken from AArch64 state

`AArch64.WatchpointByteMatch()` tests an individual byte accessed by an operation.

```
// AArch64.WatchpointByteMatch()
// =============================
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean AArch64.WatchpointByteMatch(integer n, bits(64) vaddress)

    top = AddrTop(vaddress);
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool();
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                    // Not contiguous
            byte_select_match = ConstrainUnpredictableBool();
            bottom = 3;                                       // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;           // Disabled
            when Constraint_NONE      mask = 0;               // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool();
    else
        WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

    return WVR_match && byte_select_match;
```

`AArch64.StateMatch()` tests the values in DBGWCR<n>_EL1.{HMC, SSC, PAC}, and if the watchpoint is Linked, also tests the Linked Context breakpoint that the watchpoint links to. `AArch64.StateMatch()` is given in the Breakpoint exceptions section. See page D2-1667.

`AArch64.WatchpointMatch()` tests the value in DBGWVR<n>_EL1.

```
// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
                                boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.WRPs);

    // "ispriv" is FALSE for LDTR/STTR instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';

    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, DBGWCR_EL1[n].LBN, ispriv);

    ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, vaddress + byte);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return value_match && state_match && ls_match && enabled;
```

AArch64.CheckWatchpoint() generates a FaultRecord that AArch64.Abort raises a Watchpoint exception for if all of the following are true:

- MDSCR_EL1.MDE is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level and Security state* on page D2-1643.
- All of the conditions required for Watchpoint exception generation are met. See *About Watchpoint exceptions* on page D2-1671.

**Note**

AArch64.CheckWatchpoint() might halt the PE and cause it to enter Debug state. External debug uses Debug state.

```
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = PSTATE.EL != EL0 && !(PSTATE.EL == EL1 && acctype == AccType_UNPRIV);

    for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
        match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

AArch64.WatchpointExeption() is called to generate a Watchpoint exception.

```
// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D2.11 Vector Catch exceptions

Vector Catch exceptions are not generated in AArch64 translation regimes.

**Note**

This means that they are never taken to EL1 using AArch64 and are only supported if at least EL1 using AArch32 is supported.

A debugger that is executing in EL2 using AArch64 can route Vector Catch exceptions to EL2 using AArch64. See *Routing debug exceptions* on page D2-1641.

`AArch64.VectorCatchException()` is called to generate a Vector Catch exception:

```
// AArch64.VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert HaveEL(EL2) && !IsSecure() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

*Vector Catch exceptions* on page G2-3937 describes Vector Catch exceptions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D2.12 Software Step exceptions

The following subsections describe Software Step exceptions:

- *About Software Step exceptions*.
- *Rules for setting MDSCR_EL1.SS to 1*.
- *The software step state machine* on page D2-1687.
- *Entering the active-not-pending state* on page D2-1688.
- *Behavior in the active-not-pending state* on page D2-1692.
- *Entering the active-pending state* on page D2-1693.
- *Behavior in the active-pending state* on page D2-1694.
- *Stepping T32 IT instructions* on page D2-1695.
- *Exception syndrome information and preferred return address* on page D2-1695.
- *Additional considerations* on page D2-1697.
- *Pseudocode description of Software Step exceptions* on page D2-1699.

### D2.12.1 About Software Step exceptions

Software step is an ARMv8-A resource that a debugger can use to make the PE single-step instructions.

For example, by using software step, debugger software executing at a higher Exception level can single-step instructions at a lower Exception level.

Operation is as follows:

1. A debugger:
   a. Enables software step by setting MDSCR_EL1.SS to 1. See *The debug exception enable controls* on page D2-1640.
   b. Executes an exception return instruction, `ERET`, to branch to the instruction to be single-stepped in the software being debugged.
2. The PE then:
   a. Executes the instruction to be single-stepped.
   b. Takes a Software Step exception on the next instruction, returning control to the debugger.

The PE can only take a Software Step exception if debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level and Security state* on page D2-1643.

A state machine describes the behavior of software step, shown in *The software step state machine* on page D2-1687.

Throughout this *Software Step exceptions* section, including in all subsections, $EL_D$ means the Exception level that Software Step exceptions are targeting. *Routing debug exceptions* on page D2-1641 defines $EL_D$ as the *debug target Exception level*.

### D2.12.2 Rules for setting MDSCR_EL1.SS to 1

Debugger software must be executing in an Exception level and Security state that debug exceptions are disabled from when it sets MDSCR_EL1.SS to 1.

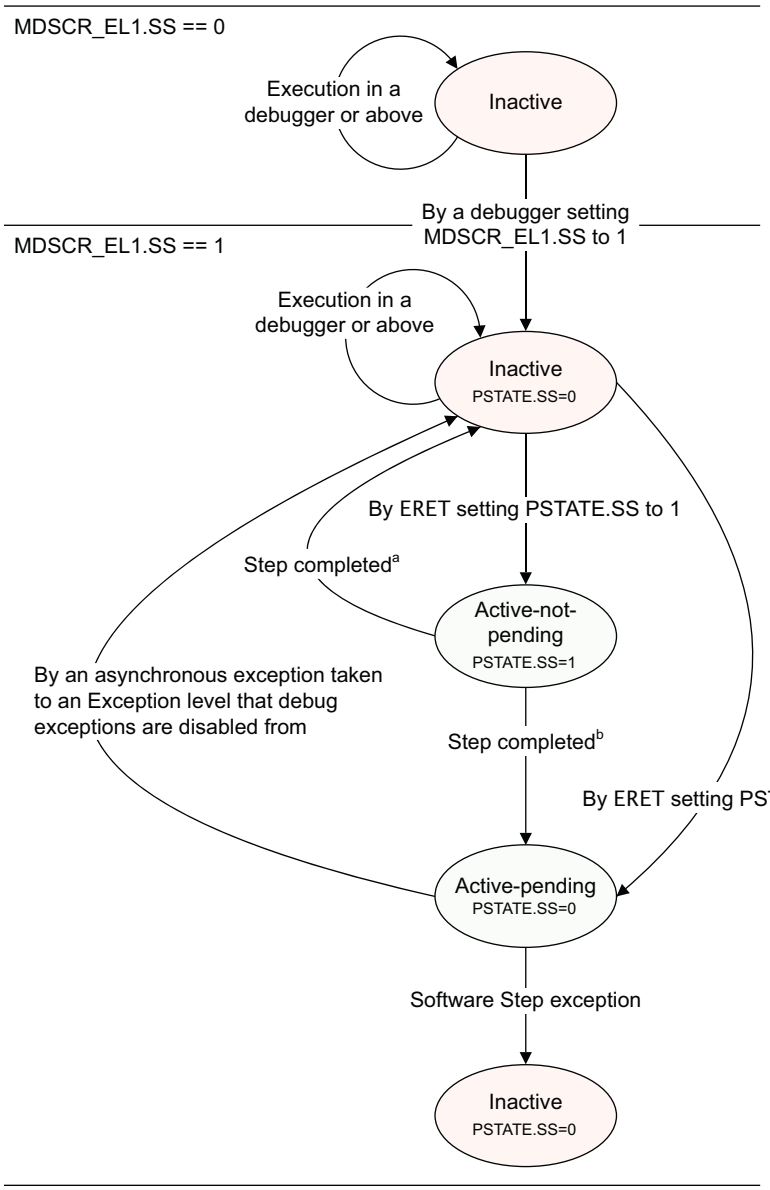
The Exception level that hosts the debugger software must be using AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D2.12.3 The software step state machine

In Figure D2-4:

- The OS Lock is unlocked and EDPRSR.DLK is 0.
- The PE is not in Secure state with MDCR_EL3.SDD set to 1. That is, the PE is in Non-secure state, or is in Secure state with MDCR_EL3.SDD set to 0, or the implementation does not include EL3.




Execution is at either:

- An Exception level that is higher than $EL_D$.
- $EL_D$ with (PSTATE.D == 1 || MDSCR_EL1.KDE == 0).

This is termed *execution in a debugger or above*.

To make the PE single-step an instruction, the debugger:

1. Sets SPSR_ELx.SS to 1.
2. Programs the ELR_ELx to point to the instruction to be stepped.
3. Executes an ERET instruction.

Execution is in the software being debugged, at either:

- An Exception level that is lower than $EL_D$.
- $EL_D$ with (PSTATE.D == 0 && MDSCR_EL1.KDE == 1).

Execution is in the software being debugged, at either:

- An Exception level that is lower than $EL_D$.
- $EL_D$ with (PSTATE.D == 0 && MDSCR_EL1.KDE == 1).

A Software Step exception is pending.

Execution has returned to the debugger.

a. The step is the PE either:
   - Taking an exception to an Exception level that debug exceptions are disabled from.
   - If execution is at $EL_D$ with MDSCR_EL1.KDE == 1, executing an instruction that sets PSTATE.D to 1.

   Software step is inactive when debug exceptions are disabled from the current Exception level, and debug exceptions are disabled from $EL_D$ when PSTATE.D is 1.

b. The step is the PE either:
   - Executing the instruction to be stepped without taking an exception.
   - Taking an exception to an Exception level that debug exceptions are enabled from. The Exception level might be using AArch64 or AArch32.

c. Or, if execution is at $EL_D$ with MDSCR_EL1.KDE == 1, by software setting PSTATE.D to 0.

**Figure D2-4 Software step state machine**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a description of when debug exceptions are enabled or disabled from an Exception level, see *Enabling debug exceptions from the current Exception level and Security state* on page D2-1643.

For more information about how a step is completed, see *Behavior in the active-not-pending state* on page D2-1692.

The software step states are:

**Inactive** Software step is inactive. It cannot generate any Software Step exceptions or affect PE execution. Software step is inactive whenever any of the following are true:

- MDSCR_EL1.SS is 0.
- $EL_D$ is using AArch32.
- Debug exceptions are disabled from the current Exception level or Security state.

**Active-not-pending**

None of the conditions mentioned in *Inactive* are true, therefore software step is active.

The current instruction is the instruction to be stepped.

**Active-pending**

None of the conditions mentioned in *Inactive* are true, therefore software step is active.

A Software Step exception is pending on the current instruction.

Whenever software step is active, whether the state machine is in the active-not pending state or the active-pending state depends on PSTATE.SS. Table D2-20 shows this.

**Table D2-20 State machine states**

| MDSCR_EL1.SS | $EL_D$ is using: | Debug exceptions enabled or disabled from the current Exception level and Security state | PSTATE.SS | State |
|---|---|---|---|---|
| 0 | X | X | X | Inactive |
| 1 | AArch32 | X | X | Inactive |
| 1 | AArch64 | Disabled | X | Inactive |
| 1 | AArch64 | Enabled | 1 | Active-not-pending |
| 1 | AArch64 | Enabled | 0 | Active-pending |

### D2.12.4 Entering the active-not-pending state

Software step can only enter the active-not-pending state from the inactive state.

Software step:

- Enters the active-not-pending state when an `ERET` instruction writes 1 to PSTATE.SS, by copying from SPSR_ELx.SS when it restores PSTATE.
- Might enter the active-not-pending state on exiting Debug state when DSPSR_EL0.SS or DSPSR.SS is 1. See *Exiting Debug state* on page H2-4858.

An `ERET` instruction only copies 1 from SPSR_ELx.SS to PSTATE.SS if all of the following are true:

- MDSCR_EL1.SS is 1.
- $EL_D$ is using AArch64.
- Debug exceptions are disabled from the current Exception level.
- Debug exceptions are enabled from the Exception level that the `ERET` instruction targets.

Otherwise, `ERET` instructions set PSTATE.SS to 0, regardless of the value of SPSR_ELx.SS.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D2-21 shows this. In the table:

**Lock** Means the value of (OSLSR_EL1.OSLK OR EDPRSR.DLK).

**NS** Is SCR_EL3.NS.

**SDD** Is MDCR_EL3.SDD. See *Disabling debug exceptions from Secure state* on page D2-1644.

**TDE** Is MDCR_EL2.TDE. See *Routing debug exceptions* on page D2-1641.

**Table D2-21 Value an ERET writes to PSTATE.SS**

| MDSCR_EL1.SS | Lock | NS | SDD | TDE | EL1 is using | EL2 is using | Value an ERET writes to PSTATE.SS |
|---|---|---|---|---|---|---|---|
| 0 | X | X | X | X | X | X | 0 |
| 1 | 1 | X | X | X | X | X | 0 |
| | 0 | 0 | 1 | X | X | n/a | 0 |
| | | | 0 | X | AArch32 | n/a | 0 |
| | | | | | AArch64 | n/a | See Table D2-22 on page D2-1690 |
| | | 1 | X | 0 | AArch32 | X | 0 |
| | | | | | AArch64 | AArch64 | See Table D2-22 on page D2-1690 |
| | | | | 1 | AArch32 | AArch32 | 0 |
| | | | | | X | AArch64 | See Table D2-23 on page D2-1691 |

For:

- SCR_EL3.NS == 0 or MDCR_EL2.TDE == 0, and EL1 using AArch64, so that $EL_D$ is EL1 using AArch64, Table D2-22 on page D2-1690 shows the value an ERET writes to PSTATE.SS.
- SCR_EL3.NS == 1 and MDCR_EL2.TDE == 1 and EL2 using AArch64, so that $EL_D$ is EL2 using AArch64, Table D2-23 on page D2-1691 shows the value an ERET writes to PSTATE.SS.

In both tables:

**From EL** Means the Exception level that the PE executes the ERET at.

**Target EL** Is the target Exception level of the ERET.

**Note**

If the ERET is an illegal exception return, the target Exception level of the ERET is the current Exception level. See *Illegal return events from AArch64 state* on page D1-1553.

**KDE** Is MDSCR_EL1.KDE. See *Enabling debug exceptions from the current Exception level and Security state* on page D2-1643.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D2-22 Value an ERET writes to PSTATE.SS if $EL_D$ is EL1 using AArch64**

| From EL | Target EL | KDE | PSTATE.D | SPSR_ELx.D | Software Step exceptions are enabled or disabled | | Value an ERET writes to PSTATE.SS |
|---|---|---|---|---|---|---|---|
| | | | | | From EL | Target EL | |
| EL3 | EL3 | X | X | X | Disabled | Disabled | 0 |
| | EL2 | X | X | X | Disabled | Disabled | 0 |
| | EL1 | 0 | X | X | Disabled | Disabled | 0 |
| | | 1 | X | 1 | Disabled | Disabled | 0 |
| | | | | 0 | Disabled | Enabled | SPSR_EL3.SS |
| | EL0 | X | X | X | Disabled | Enabled | SPSR_EL3.SS |
| EL2 | EL2 | X | X | X | Disabled | Disabled | 0 |
| | EL1 | 0 | X | X | Disabled | Disabled | 0 |
| | | 1 | X | 1 | Disabled | Disabled | 0 |
| | | | | 0 | Disabled | Enabled | SPSR_EL2.SS |
| | EL0 | X | X | X | Disabled | Enabled | SPSR_EL2.SS |
| EL1 | EL1 | 0 | X | X | Disabled | Disabled | 0 |
| | | 1 | 0 | X | Enabled[a] | -[b] | 0 |
| | | | 1 | 1 | Disabled | Disabled | 0 |
| | | | | 0 | Disabled | Enabled | SPSR_EL1.SS |
| | EL0 | 0 | X | X | Disabled | Enabled | SPSR_EL1.SS |
| | | 1 | 0 | X | Enabled[a] | Enabled | 0 |
| | | | 1 | X | Disabled | Enabled | SPSR_EL1.SS |

a. Because MDSCR_EL1.SS == 1, it means that the ERET is itself being stepped.
b. Depends on SPSR_EL1.D.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D2-23 Value an ERET writes to PSTATE.SS if $EL_D$ is EL2 using AArch64**

| From EL | Target EL | KDE | PSTATE.D | SPSR_ELx.D | Software Step exceptions are enabled or disabled | | Value an ERET writes to PSTATE.SS |
|---|---|---|---|---|---|---|---|
| | | | | | From EL | Target EL | |
| EL3 | EL3 | X | X | X | Disabled | Disabled | 0 |
| | EL2 | 0 | X | X | Disabled | Disabled | 0 |
| | | 1 | X | 1 | Disabled | Disabled | 0 |
| | | | | 0 | Disabled | Enabled | SPSR_EL3.SS |
| | EL1 | X | X | X | Disabled | Enabled | SPSR_EL3.SS |
| | EL0 | X | X | X | Disabled | Enabled | SPSR_EL3.SS |
| EL2 | EL2 | 0 | X | X | Disabled | Disabled | 0 |
| | | 1 | 0 | X | Enabled[a] | -[b] | 0 |
| | | | 1 | 1 | Disabled | Disabled | 0 |
| | | | | 0 | Disabled | Enabled | SPSR_EL2.SS |
| | EL1 | 0 | X | X | Disabled | Enabled | SPSR_EL2.SS |
| | | 1 | 0 | X | Enabled[a] | Enabled | 0 |
| | | | 1 | X | Disabled | Enabled | SPSR_EL2.SS |
| | EL0 | 0 | X | X | Disabled | Enabled | SPSR_EL2.SS |
| | | 1 | 0 | X | Enabled[a] | Enabled | 0 |
| | | | 1 | X | Disabled | Enabled | SPSR_EL2.SS |
| EL1 | EL1 | X | X | X | Enabled[a] | Enabled | 0 |
| | EL0 | X | X | X | Enabled[a] | Enabled | 0 |

a. Because MDSCR_EL1.SS == 1, it means that the ERET is itself being stepped.
b. Depends on SPSR_EL2.D.

**Note**

No AArch32 instruction can set PSTATE.SS to 1.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### D2.12.5 Behavior in the active-not-pending state

In this state, the PE does one of the following:

- Executes the instruction to be stepped and either:
  - Completes it without taking a synchronous exception.
  - Takes a synchronous exception if the instruction generates one.
- Takes an asynchronous exception without executing any instructions.
- Enters Debug state because of a *Halting debug event*.

If the PE executes the instruction to be stepped without taking any exceptions:

- After it has executed the instruction, it sets PSTATE.SS to 0 and software step advances to the active-pending state. See *Behavior in the active-pending state* on page D2-1694.

If the PE takes either a synchronous or an asynchronous exception, behavior is as described in one of the following:

- *If the PE takes an exception to an Exception level that is using AArch64*.
- *If the PE takes an exception to an Exception level that is using AArch32* on page D2-1693.

If the PE enters Debug state because of a Halting debug event, behavior is as described in *Entering Debug state and Software Step* on page H2-4830.

#### If the PE takes an exception to an Exception level that is using AArch64

As part of exception entry, the PE does all of the following:

- Sets SPSR_ELx.SS to 0 or 1, depending on the exception. See Table D2-24.
- Sets PSTATE.SS to 0. This causes software step to enter either the active-pending state or the inactive state, depending on whether debug exceptions are enabled or disabled from the Exception level that the exception is taken to:

  **Enabled** Software step enters the active-pending state.

  **Disabled** Software step enters the inactive state.

  In either case, on taking the exception, a step is complete.
- Sets PSTATE.D to 1.

**Table D2-24 Categorization of exceptions, for setting SPSR_ELx.SS to 0 or 1**

| Exception description | Exceptions | SPSR_ELx.SS |
|---|---|---|
| Exceptions whose preferred return address is for the instruction that follows the instruction to be stepped. | Supervisor Call (SVC) exceptions.<br>Hypervisor Call (HVC) exceptions.<br>Secure Monitor Call (SMC) exceptions. | 0 |
| Exceptions whose preferred return address is the address of the instruction to be stepped. | All other synchronous exceptions, and asynchronous exceptions that are taken before the instruction to be stepped. | 1 |

**Note**

If an SMC instruction executed at Non-secure EL1 is trapped to EL2 because HCR_EL2.TSC is 1, the exception is a Trap exception, not a Secure Monitor Call exception, and so SPSR_ELx.SS is set to 1, not 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### If the PE takes an exception to an Exception level that is using AArch32

This can only happen when all of the following is true:

- EL2 is implemented and is using AArch64, the PE is executing in Non-secure state, and MDCR_EL2.TDE is 1. Because MDCR_EL2.TDE is 1, $EL_D$ is EL2.
- The exception is taken to Non-secure EL1 using AArch32.

As part of exception entry, the PE sets PSTATE.SS to 0. This causes software step to enter the active-pending state.

**Note**

- Software step always enters the active-pending state because the exception is taken to an Exception level that debug exceptions are enabled from, EL1. Debug exceptions are enabled from EL1 because $EL_D$ is EL2, and debug exceptions are always enabled from Exception levels that are lower than $EL_D$.
- AArch32 SPSRs have no SS bit. Where an SPSR_ELx register architecturally maps to an AArch32 SPSR_<mode> register, SPSR_ELx.SS maps to SPSR_<mode>[21].

  SPSR_<mode>[21] is always RES0. The PE always sets SPSR_<mode>[21] to 0 on taking an exception to an Exception level that is using AArch32.

### Summary of behavior in the active-not-pending state

Table D2-25 summarizes behavior in the active-not-pending state.

**Table D2-25 Summary of behavior in the active-not-pending state**

| Event | Value written to PSTATE.SS | Execution state of the target Exception level | Exception type | Value written to SPSR_ELx.SS | Next state |
|---|---|---|---|---|---|
| No exception | 0 | n/a | n/a | n/a | Active-pending |
| Exception | 0 | AArch64 | Supervisor Call (SVC)<br>Hypervisor Call (HVC)<br>Secure Monitor Call (SMC) | 0 | Active-pending or inactive[a] |
| | | | Other | 1 | |
| | | AArch32 | All | 0[b] | Active-pending |

a. Which state software step enters depends on whether debug exceptions are enabled or disabled from the target Exception level. See Figure D2-4 on page D2-1687.

b. SPSR_<mode>[21] is RES0.

## D2.12.6 Entering the active-pending state

Software step enters the active-pending state after any of the following operations, provided that both:

- MDSCR_EL1.SS is 1.
- Debug exceptions are enabled from the Exception level and Security state that execution is in after the operation.

The operations are:

**While software step is in the active-not-pending state**

The PE either:

- Executing the instruction to be stepped without taking any exceptions.
- Taking an exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

If entry to the active-pending state is because of the PE taking an exception, it means that the exception is one that is taken to Non-secure EL1 when MDCR_EL2.TDE is 1. Otherwise, debug exceptions are masked by PSTATE.D, therefore they would be disabled from the target Exception level of the exception.

**While software step is in the inactive state**

The PE either:

- Executing an `ERET` instruction when SPSR_ELx.SS is 0.
- If MDSCR_EL1.KDE is 1, executing an `MSR DAIF` or `MSR DAIFClr` instruction that clears PSTATE.D to 0.

In addition, software step might enter the active-pending state either:

- After a direct write to a System register, for example a write to MDSCR_EL1.KDE or MDSCR_EL1.SS. These writes require explicit synchronization to guarantee their effect. See *Synchronization and the software step state machine* on page D2-1698.
- On exiting Debug state when DSPSR_EL0.SS or DSPSR.SS is 0. See *Exiting Debug state* on page H2-4858.

### D2.12.7 Behavior in the active-pending state

In this state, a Software Step exception is pending, and the PE takes it on the current instruction.

Software Step exceptions have priority over all other exceptions except asynchronous exceptions taken to an Exception level or Security state that debug exceptions are disabled from.

This means that there are some asynchronous exceptions that Software Step exceptions have priority over.

**Note**

- This is the only case where a synchronous exception explicitly has a higher priority than asynchronous exceptions.
- For a description of when debug exceptions are enabled or disabled from an Exception level or Security state, see *Enabling debug exceptions from the current Exception level and Security state* on page D2-1643.

In cases where both a Software Step exception is pending and an asynchronous exception taken to an Exception level or Security state that debug exceptions are disabled from is pending, the architecture does not define which exception is taken first.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D2.12.8 Stepping T32 IT instructions

The ARMv8-A architecture permits a combination of an IT instruction and another 16-bit T32 instruction to comprise one 32-bit instruction.

For the purpose of stepping an item, it is IMPLEMENTATION DEFINED whether:

- The PE considers this combination to be one instruction.
- The PE considers this combination to be two instructions.

In an implementation that supports the ITD control, that can disable some uses of the IT instruction, it is then IMPLEMENTATION DEFINED whether this behavior depends on the value of the applicable ITD field. For example:

- The PE might consider this combination to be one instruction, regardless of the state of the applicable ITD field.
- The PE might consider this combination to be two instructions, regardless of the state of the applicable ITD field.
- The PE might consider this combination to be one instruction when the applicable ITD field is 1, and two instructions when it is 0.

The applicable ITD field is one of :

- SCTLR_EL1.ITD if execution is at EL0 using AArch32 when EL1 is using AArch64.
- SCTLR.ITD if execution is at Non-secure EL0 or EL1 using AArch32.
- HSCTLR.ITD if execution is at Non-secure EL2 using AArch32.

### D2.12.9 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information*.
- *Preferred return address* on page D2-1697.

#### Exception syndrome information

On taking a Software Step exception, the PE records information about the exception in the Exception Syndrome Register (ESR) at the Exception level the exception is taken to. The ESR used is one of:

- ESR_EL1.
- ESR_EL2.

**Note**

Software Step Exceptions cannot be taken to EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D2-26 shows the information that the PE records.

**Table D2-26 Information recorded in the ESR_ELx**

| ESR_ELx field | Information recorded in ESR_EL1 or ESR_EL2 | |
|---|---|---|
| *Exception Class*, EC | The PE sets this to:<br>• `0x32`, if the exception was taken from a lower Exception level.<br>• `0x33`, if the exception was taken from the current Exception level. | |
| *Instruction Length*, IL | The PE sets this to 1. | |
| *Instruction Specific Syndrome*, ISS | **ISS[24]** | *Instruction Syndrome Valid* (ISV). This indicates whether the EX bit, ISS[6], is valid. The PE sets this to:<br>**0** Not valid.<br>**1** Valid. |
| | **ISS[23:7]** | RES0. |
| | **ISS[6]** | *Exclusive operation* (EX). The PE sets this to indicate whether the instruction stepped was a Load-Exclusive class of instruction:<br>**0** The stepped instruction was not a Load-Exclusive instruction.<br>**1** The stepped instruction was a Load-Exclusive instruction.<br>A debugger can use this information when stepping code that uses exclusive monitors. See *Stepping code that uses exclusive monitors* on page D2-1698 |
| | **ISS[5:0]** | *Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`. |

The PE only sets ISV to 1 if an instruction was stepped. If the PE sets ISV to 1, it must also set EX to indicate whether the instruction stepped was a Load-Exclusive class of instruction.

If no instruction was stepped because software step entered the active-pending state from the inactive state without passing through the active-not-pending state, the PE sets both ESR_ELx.{ISV, EX} to 0.

**Note**

An implementation that always sets ISV to 0 and never sets EX is not compliant.

ESR_ELx.ISV is UNKNOWN if, in the active-not-pending state, either:

- The instruction stepped was an `ERET` or an `ISB`. In these cases, ESR_ELx.EX is set to 0.
- MDCR_EL2.TDE was set to 1 and either:
  - The instruction to be stepped generated a synchronous exception that was taken to Non-secure EL1. In this case, the instruction to be stepped never completed.
  - The PE took an asynchronous exception before it could execute the instruction to be stepped. In this case, the instruction to be stepped was never executed.

  In both of these cases, ESR_ELx.EX is set to the correct value for the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D2-27 shows the permitted scenarios.

**Table D2-27 Values that the PE can record in ESR_ELx.{ISV, EX}**

| Description | ESR_ELx.ISV | ESR_ELx.EX |
|---|---|---|
| Syndrome data is not available because no instruction was stepped. | 0 | 0 |
| Syndrome data is available because an instruction was stepped. The instruction stepped was an instruction other than a Load-Exclusive class of instruction. | 1 | 0 |
| Syndrome data is available because an instruction was stepped. The instruction stepped was a Load-Exclusive class of instruction. | 1 | 1 |
| The instruction stepped was an ERET or an ISB. | UNKNOWN | 0 |
| The instruction to be stepped generated a synchronous exception that was taken to Non-secure EL1. | UNKNOWN | Set to the correct value for the instruction. |
| The PE took an asynchronous exception before it could execute the instruction to be stepped. | UNKNOWN | Set to the correct value for the instruction. |

**Note**

- A Load-Exclusive class of instruction is any one of the following:
  - In the A64 instruction set, any instruction that has a mnemonic starting with either LDX or LDAX.
  - In the A32 and T32 instruction sets, any instruction that has a mnemonic starting with either LDREX or LDAEX.
- ESR_ELx.EX is UNKNOWN if the stepped instruction was a conditional Load-Exclusive instruction that failed its condition code test.

### Preferred return address

The preferred return of a Software Step exception is the address of the instruction that was not executed because the PE took the Software Step exception instead.

## D2.12.10 Additional considerations

This section contains the following:

- *Behavior when an ERET instruction is an illegal exception return*.
- *Behavior when the instruction stepped writes a misaligned PC value* on page D2-1698.
- *Stepping code that uses exclusive monitors* on page D2-1698.
- *Synchronization and the software step state machine* on page D2-1698.

### Behavior when an ERET instruction is an illegal exception return

If the conditions for entering the active-not-pending state in *Entering the active-not-pending state* on page D2-1688 are met, but the PE executes an ERET instruction that is an illegal exception return, the exception return must be taken to the same Exception level that it was taken from. In this scenario, even though the Exception level remains the same before and after the ERET, software step can advance from the inactive state to one of the active states. Consider the following case:

1. MDSCR_EL1.SS is 1 and software step is inactive. The current Exception level is EL1 using AArch64, the OS Lock and OS Double Lock are unlocked, and MDCR_EL2.TDE is 0, MDSCR_EL1.KDE is 1, and PSTATE.D is 1.

   PSTATE.D == 1 is the reason why software step is inactive, because PSTATE.D == 1 means that debug exceptions are disabled from the current Exception level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2. The PE executes an ERET instruction.

3. The intended target of the ERET is EL2. This means that the ERET is an illegal exception return because the intended target is higher than the Exception level the ERET it is executed at. In this case, the ERET must target EL1 instead of EL2.

   If SPSR_EL1.D is 0, then on the ERET PSTATE.D becomes 0 and debug exceptions become enabled from the current Exception level. Software step therefore advances from the inactive state to one of the active states.

Which active state software step advances to depends on whether SPSR_ELx.SS is 1 or 0:

- If SPSR_ELx.SS is 1, software step advances to the active-not-pending state.

  In this case, an Illegal Execution state exception is pending on the instruction to be stepped, and the PE takes the Illegal Execution state exception instead of executing the instruction to be stepped.

- If SPSR_ELx.SS is 0, software step advances to the active-pending state.

  In this case, a Software Step exception and an Illegal Execution state exception are both pending. The Software Step exception has higher priority. On taking the Software Step exception, the PE sets SPSR_ELx.IL to 1.

**Note**

*Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 shows the relative priorities of synchronous exceptions.

### Behavior when the instruction stepped writes a misaligned PC value

An indirect branch that writes a misaligned PC value might generate a Misaligned PC exception at the target of the branch. However, if the indirect branch is stepped using software step, the PE takes a Software Step exception instead, because the Software Step exception has higher priority. Behavior on returning from the Software Step exception depends on which Execution state the Exception level being returned to is using:

**AArch64** A Misaligned PC exception is generated.

**AArch32** The return from the Software Step exception forces the PC to the correct alignment, and no Misaligned PC exception is generated.

Debugger software must therefore take care when using software step to single-step an indirect branch instruction executed in AArch32 state, that it does not hide a Misaligned PC exception.

### Stepping code that uses exclusive monitors

The ARMv8-A architecture provides no mechanism for preserving the state of the exclusive monitors when a Load-Exclusive or a Store-Exclusive instruction is stepped.

However, for certain progressions through the software step state machine, on taking a Software Step exception, the PE provides an indication of whether the instruction stepped was a Load-Exclusive class of instruction.

Debugger software can use this to detect the state of the exclusive monitors. For example, if the PE reports that the instruction stepped was a Load-Exclusive class of instruction, the debugger is aware that the next Store-Exclusive operation will fail, because all exclusive monitors are cleared on returning from the Software Step exception. The debugger must then take action to ensure that the code being stepped makes forwards progress.

For more information on how the PE reports whether the instruction stepped was a Load-Exclusive instruction, see *Exception syndrome information and preferred return address* on page D2-1695.

### Synchronization and the software step state machine

Any of the following can cause transitions between software step states:

- A direct write to a System register.
- A write to an external debug register that affects the routing of debug exceptions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Because the software step state machine indirectly reads these registers, it is not guaranteed to observe any new values until after a *Context Synchronization Operation* (CSO) has occurred.

In the time between a write to one of these registers and the next CSO, it is CONSTRAINED UNPREDICTABLE whether software step uses the state of the PE before the write, or the state of the PE after the write.

After a CSO, the state machine must use the state of the PE after the write.

**Example D2-3**

1. Software changes MDSCR_EL1.SS from 0 to 1 when debug exceptions are enabled.
2. The PE executes some instructions.
3. A CSO occurs.

During step 2, it is CONSTRAINED UNPREDICTABLE whether software step remains in the inactive state, as if MDSCR_EL1.SS is 0, or enters the active-pending state because MDSCR_EL1.SS is 1. If it is in the:

- Inactive state, then after the CSO, it must enter the active-pending state.
- Active-pending state, the PE might take a Software Step exception before the CSO.

### D2.12.11 Pseudocode description of Software Step exceptions

`SSAdvance()` advances software step from the active-not-pending state to the active-pending state, by setting PSTATE.SS to 0. It is called on completing execution of each instruction.

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

`CheckSoftwareStep()` checks whether software step is in the active-pending state, and if it is, generates a Software Step exception. It is called before each instruction executed, regardless of Execution state, before checking for any other synchronous exceptions.

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    if !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() then
        if MDSCR_EL1.SS == '1' && PSTATE.SS == '0' then AArch64.SoftwareStepException();
```

`DebugExceptionReturnSS()` returns the value to write to PSTATE.SS on an exception return or an exit from Debug state. See *Entering the active-not-pending state* on page D2-1688.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(32) spsr)
    assert Halted() || Restarting() ||  PSTATE.EL != EL0;

    SS_bit = '0';

    if MDSCR_EL1.SS == '1' then
        if Restarting() then
            enabled_at_source = FALSE;
        elsif UsingAArch32() then
            enabled_at_source = AArch32.GenerateDebugExceptions();
        else
            enabled_at_source = AArch64.GenerateDebugExceptions();

        if IllegalExceptionReturn(spsr) then
            dest = PSTATE.EL;
        else
            (valid, dest) = ELFromSPSR(spsr);  assert valid;

        secure = IsSecureBelowEL3() || dest == EL3;

        if ELUsingAArch32(dest) then
            enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
        else
            mask = spsr<9>;
            enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);

        ELd = DebugTargetFrom(secure);
        if !ELUsingAArch32(ELd) && !enabled_at_source && enabled_at_dest then
            SS_bit = spsr<21>;

    return SS_bit;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D2.13 Synchronization and debug exceptions

The behavior of debug depends on all of the following:

- The state of the external debug authentication interface.
- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.

If a change is made to any of these, the effect of that change on debug exception generation cannot be relied on until after a *Context Synchronization Operation* (CSO) has occurred. Similarly, the effect of the change on the software step state machine cannot be relied on until after a CSO has occurred.

For any instructions executed between the time when the change is made and the time when the next CSO occurs, it is CONSTRAINED UNPREDICTABLE whether debug uses the state of the PE before the change, or the state of the PE after the change.

**Example D2-4**

1. Software changes MDSCR_EL1.MDE from 0 to 1.
2. An instruction is executed, that would cause a Breakpoint exception if self-hosted debug uses the state of the PE after the change.
3. A CSO occurs.

In this case, it is CONSTRAINED UNPREDICTABLE whether the instruction generates a Breakpoint exception.

**Example D2-5**

1. Software unlocks the OS lock.
2. The PE executes some instructions.
3. A CSO occurs.

During the time when the PE is executing some instructions, step 2, it is CONSTRAINED UNPREDICTABLE whether debug exceptions other than Software Breakpoint Instruction exceptions can be generated.

**Note**

- Some register updates are self synchronizing. Others require an explicit CSO. For more information, see both:
  - *Accessing PSTATE fields* on page D1-1530.
  - *Synchronization requirements for System registers* on page D7-1908.
  - *Synchronization of changes to the external debug registers* on page H8-4946.
- See *Context synchronization operation* for the definition of this term.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Chapter D3
# The AArch64 System Level Memory Model

This chapter provides a system level view of the general features of the memory system. It contains the following sections:


- *About the memory system architecture* on page D3-1704.
- *Address space* on page D3-1705.
- *Mixed-endian support* on page D3-1706.
- *Cache support* on page D3-1707.
- *External aborts* on page D3-1727.
- *Memory barrier instructions* on page D3-1729.
- *Pseudocode description of general memory system instructions* on page D3-1730.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D3.1 About the memory system architecture

The ARM architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that permits implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system see *Memory hierarchy* on page B2-70.

### D3.1.1 Form of the memory system architecture

The ARMv8 A-profile architecture includes a *Virtual Memory System Architecture* (VMSA), described in Chapter D4 *The AArch64 Virtual Memory System Architecture*.

### D3.1.2 Memory attributes

*Memory types and attributes* on page B2-93 describes the memory attributes, including how different memory types have different attributes. Each location in memory has a set of memory attributes, and the translation tables define the virtual memory locations, and the attributes for each location.

Table D3-1 shows the memory attributes that are visible at the system level.

**Table D3-1 Memory attribute summary**

| Memory type | Shareability | Cacheability |
|---|---|---|
| Device[a] | Outer Shareable | Non-cacheable. |
| Normal | One of:<br>• Non-shareable.<br>• Inner Shareable.<br>• Outer Shareable. | One of:<br>• Non-cacheable[b].<br>• Write-Through Cacheable.<br>• Write-Back Cacheable. |

a. Takes additional attributes, see *Device memory* on page B2-98.

b. See also *Cacheability, cache allocation hints, and cache transient hints* on page D3-1709.

For more information on cacheability and shareability see *Shareable Normal memory* on page B2-94, *Non-shareable Normal memory* on page B2-95, and *Caches and memory hierarchy* on page B2-70.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D3.2 Address space

The ARMv8 architecture is designed to support a wide range of applications with different memory requirements. It supports a range of *physical address* (PA) sizes, and provides associated control and identification mechanisms. For more information, see *Address size configuration* on page D4-1750.

### D3.2.1 Instruction address space overflow

When a PE performs a *Simple sequential execution* of instructions, it calculates:

```
(address_of_current_instruction) + (size_of_executed_instruction)
```

This calculation is performed after each instruction to determine which instruction to execute next.

If the address calculation performed after executing an instruction overflows `0xFFFF FFFF FFFF FFFF`, the program counter becomes UNKNOWN.

**Note**

Address tags are not propagated to the program counter, so the tag does not affect the address calculation.

Where an instruction accesses a sequential set of bytes that crosses the `0xFFFF_FFFF_FFFF_FFFF` boundary when tagged addresses are not used, or the `0xxxFF_FFFF_FFFF_FFFF` boundary when tagged addresses are used, then the virtual address accessed for the bytes above this boundary is UNKNOWN. When tagged addresses are used, the value of the tag associated with the address also becomes UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D3.3 Mixed-endian support

A control bit, SCTLR_EL1.E0E is provided to allow the endianness of explicit data accesses made while executing at EL0 to be controlled independently of those made while executing at EL1. Table D3-2 shows the endianness of explicit data accesses and translation table walks.

**Table D3-2 Endianness support**

| Exception level | Explicit data accesses | Stage 1 translation table walks | Stage 2 translation table walks |
|---|---|---|---|
| EL0 | SCTLR_EL1.E0E | SCTLR_EL1.EE | SCTLR_EL2.EE |
| EL1 | SCTLR_EL1.EE | SCTLR_EL1.EE | SCTLR_EL2.EE |
| EL2 | SCTLR_EL2.EE | SCTLR_EL2.EE | N/A |
| EL3 | SCTLR_EL3.EE | SCTLR_EL3.EE | N/A |

**Note**

SCTLR_EL1.E0E has no effect on the endianness of the LDTR, LDTRH, LDTRSH, and LDTRSW instructions, or on the endianness of the STTR and STTRH instructions, when these are executed at EL1.

ARMv8 provides the following options for endianness support:

- All Exception levels support mixed-endianness:
  - SCTLR_ELx.EE is RW and SCTLR_EL1.E0E is RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only little-endianness:
  - SCTLR_ELx is RES0 and SCTLR_EL1.E0E is RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only big-endianness:
  - SCTLR_ELx is RES1 and SCTLR_EL1.E0E is RW.
- All Exception levels support only little-endianness:
  - SCTLR_ELx is RES0 and SCTLR_EL1.E0E is RES0.
- All Exception levels support only big-endianness:
  - SCTLR_ELx is RES1 and SCTLR_EL1.E0E is RES1.

If mixed endian support is implemented for an Exception level using AArch32, endianness is controlled by PSTATE.E. For exception returns to AArch32 state, PSTATE.E is copied from SPSR_ELx.E. If the target Exception level supports only little-endian accesses, SPSR_ELx.E is RES0. If the target Exception level supports only big-endian accesses, SPSR_ELx.E is RES1. PSTATE.E is ignored in AArch64 state.

The `BigEndian()` function determines whether the current Exception level and Execution state is using big-endian data:

```
// BigEndian()
// ===========

boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR_EL1.E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D3.4 Cache support

This section describes the ARMv8 cache identification and control mechanisms, and the cache maintenance instructions, in the following sections:

- *General behavior of the caches*.
- *Cache identification* on page D3-1708.
- *Cacheability, cache allocation hints, and cache transient hints* on page D3-1709.
- *Behavior of caches at reset* on page D3-1709
- *Cache enabling and disabling* on page D3-1710.
- *Non-cacheable accesses and instruction caches* on page D3-1712.
- *Overview of the cache maintenance instructions* on page D3-1712.
- *Cache maintenance instructions* on page D3-1716
- *Data cache zero instruction* on page D3-1724.
- *Cache lockdown* on page D3-1725.
- *System level caches* on page D3-1726.
- *Branch prediction* on page D3-1726.

See also *Caches in a VMSA implementation* on page D4-1850.

### D3.4.1 General behavior of the caches

When a memory location is marked with a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. The following non-exhaustive list of factors might be involved:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- A memory location present in the cache remains in the cache.
- A memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
  - A particular implementation.
  - Some memory attributes.
- An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.
- A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

  **Note**

  For more information, see *The interaction of cache lockdown with cache maintenance instructions* on page D3-1725.

- If a memory location is marked as cacheable for the translation regime at either the current Exception level or at a higher Exception level, then there is no mechanism that can guarantee that the memory location cannot be allocated to an enabled cache at any time.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Any application must assume that any memory location with such access permissions and cacheability attributes can be allocated to any enabled cache at any time.

- It is guaranteed that no memory location that does not have a Cacheable attribute is allocated into the cache.
- It is guaranteed that no memory location is allocated to the cache if it is not marked as cacheable in both:
  - The translation regime at the current Exception level.
  - The translation regime at a higher Exception level.
- For data accesses, any memory location that is marked as Normal Inner Shareable or Normal Outer Shareable is guaranteed to be coherent with all masters in its shareability domain.
- Any memory location is not guaranteed to remain incoherent with the rest of memory.
- The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by an observer in the shareability domain of that memory location. The maximum size of the memory that can be overwritten is called the *Cache Write-back Granule*. In some implementations the CTR_EL0 identifies the Cache Write-back Granule.
- The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer, if it was previously visible to that observer.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to its size.

### D3.4.2 Cache identification

The ARMv8 cache identification registers describe the implemented caches that are under the control of the PE:

- The Cache Type Register, CTR_EL0, defines:
  - The minimum line length of any of the instruction caches affected by the instruction cache maintenance instructions.
  - The minimum line length of any of the data or unified caches, affected by the data cache maintenance instructions.
  - The cache indexing and tagging policy of the Level 1 instruction cache.
- A single Cache Level ID Register defines:
  - The type of cache implemented at each cache level, up to the maximum of seven levels.
  - The *Level of Coherence* (LoC) for the caches. See *Terms used in describing the maintenance instructions* on page D3-1712 for a definition of these terms.
  - The *Level of Unification Uniprocessor* (LoUU) for the caches. See *Terms used in describing the maintenance instructions* on page D3-1712 for a definition of these terms.
  - An optional ISB field to indicate the cacheable boundary between inner and outer domains.

  For more information, see *CLIDR_EL1, Cache Level ID Register* on page D7-1930.
- A single Cache Size Selection Register selects the cache level and cache type of the current Cache Size Identification Register, see *CSSELR_EL1, Cache Size Selection Register* on page D7-1940.
- For each implemented cache, across all the levels of caching, a Cache Size Identification Register defines:
  - Whether the cache supports Write-Through, Write-Back, Read-Allocate and Write-Allocate.
  - The number of sets, associativity and line length of the cache. See *Terms used in describing the maintenance instructions* on page D3-1712 for a definition of these terms.

  For more information, see *CCSIDR_EL1, Current Cache Size ID Register* on page D7-1928.

To determine the cache topology associated with a PE:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache. This register also provides the size of the smallest cache lines used for the instruction caches, and for the data and unified caches. These values are used in cache maintenance instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache type fields, for cache levels 1 to 7. Scanning these fields, starting from Level 1, identifies the instruction, data or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are defined. The Cache Level ID Register also specifies the Level of Unification (LoU) and the Level of Coherence (LoC) for the cache implementation.

3. For each cache identified at stage 2:
   - Write to the Cache Size Selection Register to select the required cache. A cache is identified by its level, and whether it is:
     — An instruction cache.
     — A data or unified cache.
   - Read the Cache Size ID Register to find details of the cache.

### D3.4.3 Cacheability, cache allocation hints, and cache transient hints

Cacheability only applies to Normal memory, and can be defined independently for Inner and Outer cache locations.

As described in *Memory types and attributes* on page B2-93, the memory attributes include a cacheability attribute that is one of:

- Non-cacheable.
- Write-Through cacheable.
- Write-Back cacheable.

Cacheability attributes other than Non-cacheable can be complemented by a *cache allocation hint*. This is an indication to the memory system of whether allocating a value to a cache is likely to improve performance. A *cache transient hint* provides a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future.

The following cache allocation hints can be used in ARMv8:

- Read-Allocate, Transient Read-Allocate, or No Read-Allocate.
- Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

**Note**

A Cacheable location with both no Read-Allocate and no Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply for a location that is Cacheable, no Read-Allocate, no Write-Allocate.

The architecture does not require an implementation to make any use of cache allocation hints. This means an implementation might not make any distinction between memory locations with attributes that differ only in their cache allocation hint.

### D3.4.4 Behavior of caches at reset

In ARMv8:

- All caches reset to IMPLEMENTATION DEFINED states that might be UNKNOWN.
- All caches are disabled at reset.
- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before it is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, and the routine must be documented clearly as part of the documentation of the device.
- If an implementation permits cache hits when the cache is disabled the cache initialization routine must:
  — Provide a mechanism to ensure the correct initialization of the caches.
  — Be documented clearly as part of the documentation of the device.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In particular, if an implementation permits cache hits when the cache is disabled and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.

- ARM recommends that whenever an invalidation routine is required, it is based on the ARMv8 cache maintenance instructions.

See also *TLB behavior at reset* on page D4-1835.

### D3.4.5 Cache enabling and disabling

When a data cache or unified cache is disabled for a translation regime, data accesses and translation table walks from that translation regime to all Normal memory types behave as Non-cacheable for all levels of data caches and unified caches.

For the EL1&0 translation regime:

- When the value of SCTLR_EL1.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL1&0 stage 1 translation tables are Non-cacheable.
- When the value of HCR_EL2.CD is 1:
  - All stage 2 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL1&0 stage 2 translation tables are Non-cacheable.

  **Note**
  - In Non-secure state, the stage 1 and stage 2 cacheability attributes are combined as described in *Combining the stage 1 and stage 2 cacheability attributes for Normal memory* on page D4-1822.
  - The SCTLR_EL1.C bit has no effect on the EL2 and EL3 translation regimes.
  - The HCR_EL2.CD bit affects only stage 2 of the Non-secure EL1&0 translation regime.
- When the value of HCR_EL2.DC is 1, all stage 1 translations for data accesses and all accesses to the EL1&0 stage 1 translation tables are Normal Non-shareable Inner Write-Back Cacheable Read Allocate Write Allocate, Outer Write-Back Cacheable Read Allocate Write Allocate.

For the EL2 translation regime:

- When the value of SCTLR_EL2.C is 0, all data accesses to Normal memory using the EL2 translation regime are Non-cacheable. This means all accesses made by the EL2 translation table walks are Non-cacheable.

  **Note**

  The SCTLR_EL2.C bit has no effect on the EL1&0 and EL3 translation regimes.

For the EL3 translation regime:

- When the value of SCTLR_EL3.C is 0, all data accesses to Normal memory using the EL3 translation regime are Non-cacheable. It also makes all accesses made by the EL3 translation table walks are Non-cacheable.

  **Note**

  The SCTLR_EL3.C bit has no effect on the EL1&0 and EL2 translation regimes.

The effect of SCTLR_ELx.C, HCR_EL2.DC and HCR_EL2.CD is reflected in the result of the address translation instructions in the PAR when these bits have an effect on the stages of translation being reported in the PAR.

When an instruction cache is disabled for a translation regime, instruction accesses from that translation regime to all Normal memory types behave as Non-cacheable for all levels of instruction caches and unified caches


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the EL1&0 translation regime:

- When SCTLR_EL1.I == 0, this makes all stage 1 translations for instruction accesses to Normal memory Non-cacheable.
- When HCR_EL2.ID == 1, this makes all stage 2 translations for instruction accesses to Normal memory Non-cacheable.

  **Note**

  — In Non-secure state, the stage 1 and stage 2 cacheability attributes are combined as described in *Combining the stage 1 and stage 2 cacheability attributes for Normal memory* on page D4-1822.

  — The SCTLR_EL1.I bit has no effect on the EL2 and EL3 translation regimes.

  — The HCR_EL2.ID bit affects only stage 2 of the Non-secure EL1&0 translation regime.

- If HCR_EL2.DC == 1, then the Non-secure stage 1 EL1&0 translation regime is cacheable regardless of the value of SCTLR_EL1.I.

For the EL2 translation regime:

- When SCTLR_EL2.I == 0, all instruction accesses to Normal memory using the EL2 translation regime are Non-cacheable.

  **Note**

  The SCTLR_EL2.I bit has no effect on the EL1&0 and EL3 translation regimes.

For the EL3 translation regime:

- When SCTLR_EL3.I == 0, all instruction accesses to Normal memory using the EL3 translation regime are Non-cacheable.

  **Note**

  The SCTLR_EL3.I bit has no effect on the EL1&0 and EL2 translation regimes

In addition, when SCTLR_ELx.M == 0, indicating that the stage 1 translations are disabled for that translation regime, the SCTLR_ELx.I bit has the following effect:

- If SCTLR_ELx.I == 0, instruction accesses to Normal memory from stage 1 of that translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If SCTLR_ELx.I == 1, instruction accesses to Normal memory from stage 1 of that translation regime are Outer Shareable, Inner Write-Through, Outer Write-Through.

When address translation is disabled, all instruction accesses are to Normal memory and:

- If SCTLR_ELx.I == 0, the behavior is Normal Non-cacheable.
- If SCTLR_ELx.I == 1, the behavior is Normal Outer Shareable, Inner Write-Through Cacheable, Outer Write-Through Cacheable.

For the Non-secure EL1&0 translation regime, when both stages of address translation are disabled and SCTLR_EL1.M == 0 and HCR_EL2.DC == 0, all instruction accesses are to Normal memory and:

- If SCTLR_ELx.I == 0, the behavior is Normal Non-cacheable.
- If SCTLR_ELx.I == 1, the behavior is Normal Outer Shareable, Inner Write-Through Cacheable, Outer Write-Through Cacheable.

**Note**

- In conjunction with the requirements in *Non-cacheable accesses and instruction caches* on page D3-1712, this means that the architecturally required effect of SCTLR_ELx.I is limited to its effect on caching instruction accesses in unified caches.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- This specification can give rise to different cacheability attributes between instruction and data accesses to the same location. Where this occurs, the measures for mismatch memory attributes described in *Mismatched memory attributes* on page B2-104 must be followed to manage the corresponding loss of coherency.

### D3.4.6 Non-cacheable accesses and instruction caches

Instruction accesses to Non-cacheable Normal memory can be held in instruction caches.

Correspondingly, the sequence for ensuring that modifications to instructions are available for execution must include invalidation of the modified locations from the instruction cache, even if the instructions are held in Normal Non-cacheable memory. This includes cases where the instruction cache is disabled.

Therefore when using self-modified code in non-cacheable space in a uniprocessor system, the following sequence is required:

```
; Enter this code with <Wt> containing the new 32-bit instruction
; to be held at a location pointed to by <Xn> in Normal Non-cacheable memory.
STR <Wt>, [Xn]
DSB ISH; Ensure visibility of the data stored
IC IVAU, [Xn] ; Invalidate instruction cache by VA to PoU
DSB ISH; Ensure completion of the invalidations
ISB ;
```

In a multiprocessor system, the IC IVAU is broadcast to all PEs within the Inner Shareable domain of the PE running this sequence, but additional software steps might be required to synchronize the threads with other PEs. This might be necessary so that the PEs executing the modified instructions can execute an `ISB` after completing the invalidation, and to avoid issues associated with concurrent modification and execution of instruction sequences.

Larger blocks of instructions can be modified using the IC IALLU instruction for a uniprocessor system, or a IC IALLUIS for a multiprocessor system.

**Note**

This section applies even when the instruction cache is disabled in AArch64, as described in *Cache enabling and disabling* on page D3-1710.

### D3.4.7 Overview of the cache maintenance instructions

The following sections give general information about cache maintenance:

- *Terms used in describing the maintenance instructions*.
- *The ARMv8 abstraction of the cache hierarchy* on page D3-1715.

The following sections describe cache maintenance instruction:

- *Instruction cache maintenance instructions (IC*)* on page D3-1717.
- *Data cache maintenance instructions (DC*)* on page D3-1717.

#### Terms used in describing the maintenance instructions

Cache maintenance instructions are defined to act on particular memory locations. Instructions can be defined:

- By the address of the memory location to be maintained, referred to as operating *by VA*.
- By a mechanism that describes the location in the hardware of the cache, referred to as operating *by set/way*.

In addition, for instruction caches, there are instructions that invalidate all entries.

The following subsections define the terms used in the descriptions of the cache maintenance instructions:

- *Terminology for cache maintenance instruction operating by virtual address, VA* on page D3-1713.
- *Terminology for cache maintenance instructions operating by set/way* on page D3-1713.
- *Terminology for Clean, Invalidate, and Clean and Invalidate instructions* on page D3-1714.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Terminology for cache maintenance instruction operating by virtual address, VA

The addresses used by the PE are VAs. When all applicable stages of translation are disabled, the virtual address is identical to the physical address.

**Note**

For more information about memory system behavior when MMUs are disabled, see *The effects of disabling a stage of address translation* on page D4-1781.

For the cache maintenance instruction, any instruction described as operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, the current system *Address Space IDentifier*, ASID.
- The current Security state.
- Whether the instruction was performed from Hyp mode, or from Non-secure EL1 state.
- For an instruction executed from a Non-secure EL1 state, the *Virtual Machine IDentifier*, VMID.

For a data or unified cache maintenance instruction by VA, the operation cannot generate a Data Abort exception for a Permission fault, except for the Permission fault cases described in:

- *Data cache maintenance instructions (DC*)* on page D3-1717.
- *Stage 2 fault on a stage 1 translation table walk* on page D4-1830.

For an instruction cache maintenance instruction by VA:

- It is IMPLEMENTATION DEFINED whether the operation can generate a Data Abort exception for a Translation fault or an Access flag fault.
- The operation cannot generate a Data Abort exception for a Permission fault, except for the Permission fault case described in *Stage 2 fault on a stage 1 translation table walk* on page D4-1830.

For more information about these faults, see *MMU faults* on page D4-1824.

### Terminology for cache maintenance instructions operating by set/way

Cache maintenance instruction that operate by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an instruction works on. These parameters are:

**Level** The cache level of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED and can be determined from the Cache Level ID register. See *CLIDR_EL1, Cache Level ID Register* on page D7-1930.

In the ARM architecture, the lower numbered levels are those closest to the PE. See *Memory hierarchy* on page B2-70.

**Set** Each level of a cache is split up into a number of *sets*. Each set is a set of locations in a cache level to which an address can be assigned. Usually, the set number is an IMPLEMENTATION DEFINED function of an address.

In the ARM architecture, sets are numbered from 0.

**Way** The associativity of a cache is the number of locations in a set to which a specific address can be assigned. The *way* number specifies one of these locations.

In the ARM architecture, ways are numbered from 0.

**Note**

Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, ARM expects that most portable software will use only the cache maintenance instructions by set/way as single steps in a routine to perform maintenance on the entire cache.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Terminology for Clean, Invalidate, and Clean and Invalidate instructions*

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a PE that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.

2. Updates to memory locations by other observers that can access memory might not be visible to a PE that accesses a cache. This can occur when the cache contains an old, or *stale*, copy of the memory location that has been updated.

The *Clean* and *Invalidate* instructions address these two issues. The definitions of these instructions are:

**Clean** A cache clean instruction ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the instruction is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the instruction is performed, for example to the Point of Unification.

The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the shareability domain of that memory location.

**Invalidate** A cache invalidate instruction ensures that updates made visible by observers that access memory at the point to which the invalidate is defined, are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate instruction that have been written by observers that access the cache, if those updates have not been cleaned from the cache since they were made.

If the address of an entry on which the invalidate instruction operates does not have a Normal Cacheable attribute, or if the cache is disabled, then an invalidate instruction also ensures that this address is not present in the cache.

**Note**

Entries for addresses with a Normal Cacheable attribute can be allocated to an enabled cache at any time, and so the cache invalidate instruction cannot ensure that the address is not present in an enabled cache.

**Clean and Invalidate**

A cache *clean and invalidate* instruction behaves as the execution of a clean instruction followed immediately by an invalidate instruction. Both instructions are performed to the same location.

The points to which a cache maintenance instruction can be defined differ depending on whether the instruction operates by VA or by set/way:

- For instructions operating by set/way, the point is defined to be to the next level of caching. For the All operations, the point is defined as the Point of Unification for each location held in the cache.

- For instruction operating by VA, two conceptual points are defined:

  **Point of Coherency (PoC)**

  For a particular VA, the PoC is the point at which all agents that can access memory are guaranteed to see the same copy of a memory location. In many cases, this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherence between memory system agents.

  **Note**

  The presence of system caches can affect the definition of point of coherency as described in *System level caches* on page D3-1726.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Point of Unification (PoU)**

The PoU for a PE is the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. In many cases, the Point of Unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

The PoU for an Inner Shareable shareability domain is the point by which the instruction and data caches and the translation table walks of all the PEs in that Inner Shareable shareability domain are guaranteed to see the same copy of a memory location. Defining this point permits self-modifying software to ensure future instruction fetches are associated with the modified version of the software by using the standard correctness policy of:

1. Clean data cache entry by address.
2. Invalidate instruction cache entry by address.

The following fields in the CLIDR_EL1 relate to these conceptual points:

**LoC, Level of Coherence**

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Coherency. The LoC value is a cache level, so, for example, if LoC contains the value 3:

- A clean to the Point of Coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.
- Level 4 cache is the first level that does not have to be maintained.

If the LoC field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Coherency.

If the LoC field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Coherency.

**LoUU, Level of Unification, uniprocessor**

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the PE. As with LoC, the LoUU value is a cache level.

If the LoUU field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification.

If the LoUU field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

**LoUIS, Level of Unification, Inner Shareable**

In any implementation:

- This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable shareability domain. As with LoC, the LoUIS value is a cache level.
- If the LoUIS field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable shareability domain.
- If the LoUIS field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

For more information, see *CLIDR_EL1, Cache Level ID Register* on page D7-1930.

## The ARMv8 abstraction of the cache hierarchy

The following subsections describe the ARMv8 abstraction of the cache hierarchy:

- *Cache maintenance instructions that operate by address* on page D3-1716.
- *Cache maintenance instructions that operate by set/way* on page D3-1716.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### Cache maintenance instructions that operate by address

The address-based cache maintenance instructions are described as operating by VA. Each of these instructions is always qualified as being either:

- Performed to the Point of Coherency.
- Performed to the Point of Unification.

See *Terms used in describing the maintenance instructions* on page D3-1712 for definitions of Point of Coherency and Point of Unification, and more information about possible meanings of VA.

*Cache maintenance instructions* lists the address-based maintenance instructions.

The CTR_EL0 holds minimum line length values for:

- The instruction caches.
- The data and unified caches.

These values support efficient invalidation of a range of addresses, because this value is the most efficient address stride to use to apply a sequence of address-based maintenance instructions to a range of addresses.

For the Invalidate data or unified cache line by VA instruction, the Cache Write-back Granule field of the CTR_EL0 defines the maximum granule that a single invalidate instruction can invalidate. This meaning of the Cache Write-back Granule is in addition to its defining the maximum size that can be written back.

#### Cache maintenance instructions that operate by set/way

*Cache maintenance instructions* lists the set/way-based maintenance instructions. Some encodings of these instructions include a required field that specifies the cache level for the instruction:

- A clean instruction cleans from the level of cache specified through to at least the next level of cache, moving further from the PE.
- An invalidate instruction invalidates only at the level specified.

### D3.4.8 Cache maintenance instructions

The A64 cache maintenance instructions are part of the A64 system instruction class in the register encoding space. For encoding details and other general information on these system instruction, see *System instructions* on page C3-140, *SYS* on page C6-807 and *Cache maintenance instructions, and data cache zero* on page C5-272.

The instruction and data cache maintenance instructions have the same functionality in AArch32 state and in AArch64 state. Table D3-3 shows these system instructions. Instructions that take an argument include Xt in the instruction description.

**Note**

In Table D3-3 the Point of Unification is the Point of Unification of the PE executing the cache maintenance instruction.

**Table D3-3 System instructions for cache maintenance**

| Register | Instruction | Notes |
|---|---|---|
| Instruction cache maintenance instructions, see *System instructions* on page C3-140 | | |
| IC IALLUIS | Invalidate all to Point of Unification, Inner Shareable | EL1 or higher access. |
| IC IALLU | Invalidate all to Point of Unification | EL1 or higher access. |
| IC IVAU, Xt | Invalidate by virtual address to Point of Unification | When SCTLR_EL1.UCI == 1, EL0 access. Otherwise, EL1 or higher access. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D3-3 System instructions for cache maintenance (continued)**

| Register | Instruction | Notes |
|---|---|---|
| Data cache maintenance instructions, see *System instructions* on page C3-140 | | |
| DC IVAC, Xt | Invalidate by virtual address to Point of Coherency | EL1 or higher access. |
| DC ISW, Xt | Invalidate by set/way | EL1 or higher access. |
| DC CVAC, Xt | Clean by virtual address to Point of Coherency | When SCTLR_EL1.UCI == 1, EL0 access. Otherwise EL1 or higher access. |
| DC CSW, Xt | Clean by set/way | EL1 or higher access. |
| DC CVAU, Xt | Clean by virtual address to Point of Unification | When SCTLR_EL1.UCI == 1, EL0 access. Otherwise EL1 or higher access. |
| DC CIVAC, Xt | Clean and invalidate by virtual address to Point of Coherency | When SCTLR_EL1.UCI == 1, EL0 access. Otherwise EL1 or higher access. |
| DC CISW, Xt | Clean and invalidate by set/way | EL1 or higher access. |

A `DSB` or `DMB` instruction intended to ensure the completion of cache or branch predictor maintenance instructions must have an access type of both loads and stores.

The following subsections give more information about these instructions:

- *Instruction cache maintenance instructions (IC*)*.
- *Data cache maintenance instructions (DC*)*.
- *EL0 accessibility to cache maintenance instructions* on page D3-1718.
- *General requirements for the scope of maintenance instructions* on page D3-1718.
- *Effects of instructions that operate by VA to the Point of Coherency* on page D3-1719.
- *Effects of instructions operate by VA but not to the Point of Coherency* on page D3-1719.
- *Effects of All and set/way maintenance instructions* on page D3-1720.
- *Effects of virtualization and security on the cache maintenance instructions* on page D3-1720.
- *Boundary conditions for cache maintenance instructions* on page D3-1721.
- *Ordering and completion of data and instruction cache instructions* on page D3-1722.
- *Performing cache maintenance instructions* on page D3-1723.

### Instruction cache maintenance instructions (IC*)

The A64 assembly syntax for these instructions is described in *System instructions* on page C3-140.

Where an address argument for these instructions is required, it takes the form of a 64-bit register that holds the virtual address argument. No restrictions apply for this address.

See also *EL0 accessibility to cache maintenance instructions* on page D3-1718 and *Ordering and completion of data and instruction cache instructions* on page D3-1722.

### Data cache maintenance instructions (DC*)

The A64 assembly syntax for these instructions is described in *System instructions* on page C3-140.

Where an address argument for these instructions is required, it takes the form of a 64-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

Data cache maintenance instructions that take a set/way/level argument take a 64-bit register, the upper 32 bits of which are RES0.

DC IVAC requires write permission or else a Permission fault is generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A DC IVAC or DC ISW executed at Non-secure EL1 is performed by the PE as clean and invalidate, that is as a DC CIVAC or DC CISW, if both of the following apply:

- EL2 is implemented.
- HCR_EL2.VM is set to 1, meaning the second stage of address translation is enabled.

**Note**

This also applies to the AArch32 cache maintenance instructions DCIMVAC and DCISW. see *AArch32 data cache maintenance instructions (DC*)* on page G3-3963.

If a data cache maintenance by set/way instruction specifies a set, way, or level argument that is larger than the value supported by the implementation then the instruction is CONSTRAINED UNPREDICTABLE, see *Out of range values of the Set/Way/Index fields in cache maintenance instructions* on page K1-5513 or the instruction description.

If a memory fault that sets the FAR for the translation regime applicable for the cache maintenance instruction is generated from a data cache maintenance instruction, the FAR holds the address specified in the register argument of the instruction.

**Note**

Despite its mnemonic, DC ZVA is not a cache maintenance instruction. For more information, see *DC ZVA, Data Cache Zero by VA* on page C5-353.

See also *EL0 accessibility to cache maintenance instructions* and *Ordering and completion of data and instruction cache instructions* on page D3-1722.

### EL0 accessibility to cache maintenance instructions

The SCTLR_EL1.UCI bit enables EL0 access for the DC CVAU, DC CVAC, DC CIVAC, and IC IVAU instructions. When EL0 use of these instructions is disabled because SCTLR_EL1.UCI == 0, executing one of these instructions at EL0 generates a trap to EL1, that is reported using EC = `0x18`.

For these instructions read access permission is required. When the value of SCTLR_EL1.UCI is 1:

- For the DC CVAU, DC CVAC, and DC CIVAC instructions, if the instruction is executed at EL0 and the address specified in the argument cannot be read at EL0, a Permission fault is generated.
- For the IC IVAU instruction, if the instruction is executed at EL0 and the address specified in the argument cannot be read at EL0, it is IMPLEMENTATION DEFINED whether a Permission fault is generated.

Software can read the CTR_EL0 to discover the stride needed for cache maintenance instructions. The SCTLR_EL1.UCT bit enables EL0 access to the CTR_EL0. When EL0 access to the Cache Type register is disabled, a register access instruction executed at EL0 is trapped to EL1 using EC = `0x18`.

### General requirements for the scope of maintenance instructions

The ARMv8 specification of the cache maintenance instructions describes what each instruction is guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements described in *General behavior of the caches* on page D3-1707, *Behavior of caches at reset* on page D3-1709, and *Preloading caches* on page B2-74.

This means that as a side-effect of a cache maintenance instruction:

- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

**Note**

ARM recommends that, for best performance, such side-effects are kept to a minimum. ARM strongly recommends that the side-effects of operations performed in Non-secure state do not have a significant performance impact on execution in Secure state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Effects of instructions that operate by VA to the Point of Coherency

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instruction is operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the Point of Coherency.

Table D3-4 shows the scope of the Data and unified cache maintenance instructions.

**Table D3-4 PEs affected by cache maintenance instructions to the Point of Coherency**

| Shareability | PEs affected | Effective to |
|---|---|---|
| Non-shareable | The PE performing the operation | The Point of Coherency of the entire system |
| Inner Shareable | All PEs in the same Inner Shareable shareability domain as the PE performing the operation | The Point of Coherency of the entire system |
| Outer Shareable | All PEs in the same Outer Shareable shareability domain as the PE performing the operation | The Point of Coherency of the entire system |

### Effects of instructions operate by VA but not to the Point of Coherency

For these instructions, Table D3-5 shows how, for a VA in a Normal or Device memory location, the shareability attribute of the VA determines the minimum set of PEs affected, and the point to which the instruction must be effective.

**Table D3-5 PEs affected by cache maintenance instructions to the Point of Unification**

| Shareability | PEs affected | Effective to |
|---|---|---|
| Non-shareable | The PE executing the instruction | The point of unification of instruction cache fills, data cache fills and write-backs, and translation table walks, on the PE executing the instruction |
| Inner Shareable or Outer Shareable | All PEs in the same Inner Shareable shareability domain as the PE executing the instruction | The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, of all PEs in the same Inner Shareable shareability domain as the PE executing the instruction |

**Note**

The set of PEs guaranteed to be affected is never greater than the PEs in the Inner Shareable shareability domain containing the PE executing the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Effects of All and set/way maintenance instructions

The IC IALLU and DC set/way instructions apply only to the caches of the PE that performs the instruction.

The IC IALLUIS instruction can affect the caches of all PEs in the same Inner Shareable shareability domain as the PE that performs the instruction. This instruction has an effect to the Point of Unification of instruction cache fills, data cache fills, and write-backs, and translation table walks, of all PEs in the same Inner Shareable shareability domain.

**Note**

The possible presence of system caches, as described in *System level caches* on page D3-1726, means architecture does not guarantee that all levels of the cache can be maintained using set/way instructions.

### Effects of virtualization and security on the cache maintenance instructions

Each Security state has its own physical address (PA) space, therefore cache entries are associated with PA space.

Table D3-6 shows the effects of virtualization and security on the cache maintenance instructions. In the table, the *Specified entries* are entries that the architecture requires the instruction to affect. The rules described in *General behavior of the caches* on page D3-1707 mean that an instruction might also affect other entries.

**Table D3-6 Effects of virtualization and security on the maintenance instructions**

| Cache maintenance instructions | Security state | Specified entries |
|---|---|---|
| Data or unified cache maintenance instructions | | |
| Invalidate, Clean, or Clean and Invalidate by VA: IVAC, CVAC, CVAU, CIVAC | Both | All lines that hold the PA that, in the current Security state, is mapped to by the combination of all of:<br>• The specified VA.<br>• For an instruction executed at EL1 or EL0, the current ASID if the location is mapped to by a non-global page.<br>• For an instruction executed at Non-secure EL1 or Non-secure EL0, the current VMID[a]. |
| Invalidate, Clean, or Clean and Invalidate by set/way: ISW, CSW, CISW | Non- secure | Line specified by set/way provided that the entry comes from the Non-secure PA space. |
| | Secure | Line specified by set/way regardless of the PA space that the entry has come from. |
| Instruction cache maintenance instructions | | |
| Invalidate by VA: IVAU | Both | All lines corresponding to the specified VA[b] in the current translation regime and:<br>• For an instruction executed at EL1 or EL0, the current ASID.<br>• For an instruction executed at Non-secure EL1 or Non-secure EL0, the current VMID[a]. |
| Invalidate All: IALLU, IALLUIS | Both | For an instruction executed at:<br>• Non-secure EL1, all instruction cache lines containing entries associated with the current VMID[a].<br>• EL2, all instruction cache lines containing Non-secure entries.<br>• EL3 or Secure EL1, all instruction cache lines. |

a. Dependencies on the VMID apply even when HCR_EL2.VM is set to 0. The architecture does not define a reset value for VTTBR_EL2.VMID, and therefore, in any implementation that includes EL2, the boot software executed when reset is deasserted must initialize VTTBR_EL2.VMID.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

b. The type of instruction cache used affects the interpretation of the specified entries in this table such that:
   • For a PIPT instruction cache, the cache maintenance applies to all entries whose physical address corresponds to the specified address.
   • For a VIVT instruction cache, the cache maintenance applies to entries whose virtual tag and physical address corresponds to the specified address.
   • For an ASID and VMID tagged VIVT instruction cache, the cache maintenance applies to entries whose virtual address corresponds to the specified address.

   For information on types of instruction cache see *Instruction caches* on page D4-1850.

For locked entries and entries that might be locked, the behavior of cache maintenance instructions described in *The interaction of cache lockdown with cache maintenance instructions* on page D3-1725 applies.

With an implementation that generates aborts if entries are locked or might be locked in the cache, when the use of lockdown aborts is enabled, these aborts can occur on any cache maintenance instructions.

In an implementation that includes EL2:

- The architecture does not require cache cleaning when switching between virtual machines. Cache invalidation by set/way must not present an opportunity for one virtual machine to corrupt state associated with a second virtual machine. To ensure this requirement is met, Non-secure clean by set/way operations can be upgraded to clean and invalidate by set/way.
- The AArch64 Data cache invalidate instructions, DC IVAC and DC ISW, at EL1 and EL0, and the AArch32 Data cache invalidate instructions DCIMVAC and DCISW, at EL1, perform a cache clean as well as a cache invalidation if both of the following apply:
  — The value of HCR.VM is 1.
  — The instruction is executed in Non-secure state, or EL3 is not implemented.

  The means that, in Non-secure state:
  — At EL1 using AArch64 or EL0 using AArch64, a DC IVAC instruction operates as DC CIVAC, and a DC ISW instruction operates as DC CISW.
  — At EL1 using AArch32, a DCIMVAC instruction operates as DCCIMVAC, and a DCISW instruction operates as DCCISW.
- The AArch64 Data cache invalidate by set/way instruction DC ISW, at EL1 and EL0, and the AArch32 Data cache invalidate by set/way instruction DCISW, at EL1, perform a cache clean as well as a cache invalidation if both of the following apply:
  — The value of HCR_EL2.SWIO is 1.
  — The instruction is executed in Non-secure state, or EL3 is not implemented.

  This means that, in Non-secure state:
  — At EL1 using AArch64 or EL0 using AArch64, a DC ISW instruction operates as DC CISW.
  — At EL1 using AArch32, a DCISW instruction operates as DCCISW.
- When the value of HCR_EL2.FB is 1, TLB and instruction cache invalidate instructions executed in the Non-secure EL1 Exception level are broadcast across the Inner Shareable domain. When Non-secure EL1 is using AArch64, this applies to the TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, TLBI VAALE1, and IC IALLU instructions. This means the instruction is upgraded to the corresponding Inner Shareable instruction, for example IC IALLU is upgraded to IC IALLUIS.

For more information about the cache maintenance instructions, see *Overview of the cache maintenance instructions* on page D3-1712, *Cache maintenance instructions* on page D3-1716, and Chapter D4 *The AArch64 Virtual Memory System Architecture*.

### Boundary conditions for cache maintenance instructions

Cache maintenance instructions operate on the caches when the caches are enabled or when they are disabled.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For address-based cache maintenance instructions, the instructions operate on the caches regardless of the memory type and cacheability attributes marked for the memory address in the VMSA translation table entries. This means that the effects of the cache maintenance instructions can apply regardless of:

- Whether the address accessed:
  - Is Normal memory or Device memory.
  - Has the Cacheable attribute or the Non-cacheable attribute.
- Any applicable domain control of the address accessed.
- The access permissions for the address accessed, other than the effect of the stage two write permission on data or unified cache invalidation instructions.

### Ordering and completion of data and instruction cache instructions

All data cache instructions, other than DC ZVA, that specify an address:

- Execute in program order relative to loads or stores that access an address in Normal memory with either Inner Write Through or Inner Write Back attributes within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
- Can execute in any order relative to loads or stores that access any address with the Device memory attribute, or with Normal memory with Inner Non-cacheable attribute unless a `DMB` or `DSB` is executed between the instructions.
- Execute in program order relative to other data cache maintenance instructions, other than `DC ZVA`, that specify an address within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
- Can execute in any order relative to loads or stores that access an address in a different cache line of minimum size, as indicated by CTR_EL0.DMinLine, unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to other data cache maintenance instructions, other than `DC ZVA`, that specify an address in a different cache line of minimum size, as indicated by CTR_EL0.DMinLine, unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to data cache maintenance instructions that do not specify an address unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to instruction cache maintenance instructions unless a `DSB` is executed between the instructions.

**Note**

- Data cache ordering rules by address are consistent with physically indexed physically tagged caches. See *Data and unified caches* on page D4-1850.
- *Data cache zero instruction* on page D3-1724 describes the ordering and completion rules for Data Cache Zero.

All data cache maintenance instructions that do not specify an address:

- Can execute in any order relative to data cache maintenance instructions that do not specify an address unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to data cache maintenance instructions that specify an address, other than Data Cache Zero, unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to loads or stores unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to instruction cache maintenance instructions unless a `DSB` is executed between the instructions.

All instruction cache instructions can execute in any order relative to other instruction cache instructions, data cache instructions, loads, and stores unless a `DSB` is executed between the instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A cache maintenance instruction can complete at any time after it is executed, but is only guaranteed to be complete, and its effects visible to other observers, following a DSB instruction executed by the PE that executed the cache maintenance instruction.

In all cases, where the text in this section refers to a DMB or a DSB, this means a DMB or DSB whose required access type is both loads and stores.

**Note**

These ordering requirements are extended from the requirements in AArch32 state given:

- *Ordering of cache and branch predictor maintenance instructions* on page G3-3969.
- *AArch32 instruction cache maintenance instructions (IC*)* on page G3-3962.

## Performing cache maintenance instructions

To ensure all cache lines in a block of address space are maintained through all levels of cache ARM strongly recommends that software:

- For data or unified cache maintenance, uses the CTR_EL0.DMinLine value to determine the loop increment size for a loop of data cache maintenance by VA instructions.
- For instruction cache maintenance, uses the CTR_EL0.IMinLine value to determine the loop increment size for a loop of instruction cache maintenance by VA instructions.

### *Example code for cache maintenance instructions*

The cache maintenance instructions by set/way can clean or invalidate, or both, the entirety of one or more levels of cache attached to a PE. However, unless all PEs attached to the caches regard all memory locations as Non-cacheable, it is not possible to prevent locations being allocated into the cache during such a sequence of the cache maintenance instructions.

**Note**

Since the set/way instructions are performed only locally, there is no guarantee of the atomicity of cache maintenance between different PEs, even if those different PEs are each executing the same cache maintenance instructions at the same time. Since any cacheable line can be allocated into the cache at any time, it is possible for a cache line to migrate from an entry in the cache of one PE to the cache of a different PE in a way that means the line is not affected by set/way based cache maintenance. Therefore, ARM strongly discourages the use of set/way instructions to manage coherency in coherent systems. The expected use of the cache maintenance instructions that operate by set/way is limited to the cache maintenance associated with the powerdown and powerup of caches, if this is required by the implementation.

The limitations of cache maintenance by set/way mean maintenance by set/way does not happen on multiple PEs, and cannot be made to happen atomically for each address on each PE. Therefore in multiprocessor or multithreaded systems, the use of cache maintenance by set/way to clean, or clean and invalidate, the entire cache for coherency management with very large buffers or with buffers with unknown address can fail to provide the expected coherency results because of speculation by other PEs, or possibly by other threads. The only way that these instructions can be used in this way is to first ensure that all PEs that might cause speculative accesses to caches that need to be maintained are not capable of generating speculative accesses. This can be achieved by ensuring that those PEs have no memory locations marked as cacheable. Such an approach can have very large system performance effects, and ARM advises implementers to use hardware coherency mechanisms in systems where this will be an issue.

*System level caches* on page D3-1726 refers to other limitations of cache maintenance by set/way.

The following example code for cleaning a data or unified cache to the Point of Coherency illustrates a generic mechanism for cleaning the entire data or unified cache to the Point of Coherency.

```
MRS     X0, CLIDR_EL1
AND     W3, W0, #0x07000000     // Get 2 x Level of Coherence
LSR     W3, W3, #23
CBZ     W3, Finished
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        MOV     W10, #0                 // W10 = 2 x cache level
        MOV     W8, #1                  // W8 = constant 0b1
Loop1:  ADD     W2, W10, W10, LSR #1    // Calculate 3 x cache level
        LSR     W1, W0, W2              // extract 3-bit cache type for this level
        AND     W1, W1, #0x7
        CMP     W1, #2
        B.LT    Skip                    // No data or unified cache at this level
        MSR     CSSELR_EL1, X10         // Select this cache level
        ISB                             // Synchronize change of CSSELR
        MRS     X1, CCSIDR_EL1          // Read CCSIDR
        AND     W2, W1, #7              // W2 = log2(linelen)-4
        ADD     W2, W2, #4              // W2 = log2(linelen)
        UBFX    W4, W1, #3, #10         // W4 = max way number, right aligned
        CLZ     W5, W4                  // W5 = 32-log2(ways), bit position of way in DC operand
        LSL     W9, W4, W5              // W9 = max way number, aligned to position in DC operand
        LSL     W16, W8, W5             // W16 = amount to decrement way number per iteration
Loop2:  UBFX    W7, W1, #13, #15        // W7 = max set number, right aligned
        LSL     W7, W7, W2              // W7 = max set number, aligned to position in DC operand
        LSL     W17, W8, W2             // W17 = amount to decrement set number per iteration
Loop3:  ORR     W11, W10, W9            // W11 = combine way number and cache number ...
        ORR     W11, W11, W7            //  ... and set number for DC operand
        DC      CSW, X11                // Do data cache clean by set and way
        SUBS    W7, W7, W17             // Decrement set number
        B.GE    Loop3
        SUBS    X9, X9, X16             // Decrement way number
        B.GE    Loop2
Skip:   ADD     W10, W10, #2            // Increment 2 x cache level
        CMP     W3, W10
        DSB                             // Ensure completion of previous cache maintenance instruction
        B.GT    Loop1
Finished:
```

Similar approaches can be used for all cache maintenance instructions.

### D3.4.9 Data cache zero instruction

The Data Cache Zero by Address instruction, DC ZVA, writes `0b00` to each of a block of N bytes, aligned in memory to N bytes in size, where the block in memory is identified by the address passed. There are no alignment restrictions on the address supplied. The DCZID_EL0 register indicates the block size that is written with byte values of zero.

Software can restrict access to this operation. See *Configurable instruction enables and disables, and trap controls* on page D1-1575.

If disabled, the operation at EL0 is trapped to EL1.

The `DC ZVA` instruction behaves as a set of stores to the location being accessed, and:

- Generates a Permission fault if the translation regime being used when the instruction is executed does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instruction.

In addition:

- When the instruction is executed, it can generate memory faults or watchpoints that are prioritized in the same way as other memory related faults or watchpoints. Where a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the syndrome field is not set to 1, which would be the case for all other cache maintenance instructions. See *Exception from a Data abort* on page D1-1549 for more information about the encoding of ESR_ELx and the associated ISS field.
- If the memory region being zeroed is any type of Device memory, then `DC ZVA` generates an Alignment fault which is prioritized in the same way as other alignment faults that are determined by the memory type.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The architecture makes no statements about whether or not a DC ZVA instruction causes allocation to any particular level of the cache, for addresses that have a cacheable attribute for those levels of cache.

### D3.4.10 Cache lockdown

The concept of an entry locked in a cache is allowed, but not architecturally defined. How lockdown is achieved is IMPLEMENTATION DEFINED and might not be supported by:

- An implementation.
- Some memory attributes.

An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

#### The interaction of cache lockdown with cache maintenance instructions

The interaction of cache lockdown and cache maintenance instructions is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance instruction on a locked cache line must comply with the following general rules:

- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by set/way, DC CSW.
  - Cache invalidate by set/way, DC ISW.
  - Cache clean and invalidate by set/way, DC CISW.
  - Instruction cache invalidate all, IC IALLU and IC IALLUIS.

  However, one of the following approaches must be adopted in all these cases:

  1. If the instruction specified an invalidation, a locked entry is not invalidated from the cache.
  2. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
  3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the fault status code defined for this purpose. See *Exception from a Data abort* on page D1-1549.

  This permits a usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked.

The effect of the following instructions is IMPLEMENTATION DEFINED:

- Cache clean by virtual address, DC CVAC and DC CVAU.
- Cache invalidate by virtual address, DC IVAC.
- Cache clean and invalidate by virtual address, DC CIVAC.

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is invalidated from the cache. For the clean and invalidate instructions, the entry must be cleaned before it is invalidated.
2. If the instruction specified an invalidation, a locked entry is not invalidated from the cache. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the fault status code defined for this purpose. See *ESR_ELx* on page K13-5711.

In an implementation that includes EL2, if HCR_EL2.TIDCP is set to 1, any exception relating to lockdown of an entry associated with Non-secure memory is routed to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

An implementation that uses an abort mechanisms for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other permitted alternatives for the locked entries.

ARM recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use architecturally-defined instructions. This minimizes the number of customized instructions required.

In addition, an implementation that uses an abort to handle cache maintenance instructions for entries that might be locked must provide a mechanism that ensures that no entries are locked in the cache.

The reset setting of the cache must be that no cache entries are locked.

#### *Additional cache functions for the implementation of lockdown*

An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED spaces reserved for Cache Lockdown, see *Reserved encodings for IMPLEMENTATION DEFINED registers* on page C5-288.

### D3.4.11 System level caches

The system level architecture might define further aspects of the software view of caches and the memory model that are not defined by the ARMv8 architecture. These aspects of the system level architecture can affect the requirements for software management of caches and coherency. For example, a system design might introduce additional levels of caching that cannot be managed using the architecturally-defined maintenance instructions. Such caches are referred to as *system caches*.

Conceptually, three classes of system cache can be envisaged:

1. System caches which lie before the point of coherency and cannot be managed by any cache maintenance instructions. Such systems fundamentally undermine the concept of cache maintenance instructions operating to the point of coherency, as they imply the use of non-architecture mechanisms to manage coherency. The use of such systems in the ARM architecture is explicitly prohibited.
2. System caches which lie before the point of coherency and can be managed by cache maintenance by address instructions that apply to the point of coherency, but cannot be managed by cache maintenance by set/way instructions. Where maintenance of the entirety of such a cache must be performed, as in the case for power management, it must be performed using non-architectural mechanisms.
3. System caches which lie beyond the point of coherency and so are invisible to the software. The management of such caches is outside the scope of the architecture.

### D3.4.12 Branch prediction

ARMv8 does not define any branch predictor maintenance instructions for AArch64 state.

If branch prediction is architecturally visible, cache maintenance must also apply to branch prediction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D3.5 External aborts

The ARM architecture defines external aborts as errors that occur in the memory system, other than those that are detected by the MMU or debug logic. External aborts include parity or ECC errors detected by the caches or other parts of the memory system. For example, an uncorrectable parity or ECC failure on a Level 2 Memory structure might generate an external abort.

An external abort is one of the following:

- Synchronous.
- Precise asynchronous.
- Imprecise asynchronous.

For more information, see *Exception terminology* on page D1-1515.

The ARM architecture does not provide any method to distinguish between precise asynchronous and imprecise asynchronous aborts.

In AArch64 state, asynchronous aborts are reported using the SError interrupt exception. See *Asynchronous exception types, routing, masking and priorities* on page D1-1569.

Synchronous external aborts are reported using the Instruction Abort and Data Abort exceptions. See *Synchronous exception types, routing and priorities* on page D1-1563.

VMSAv8-64 permits external aborts on data accesses, translation table walks, and instruction fetches to be either synchronous or asynchronous.

It is IMPLEMENTATION DEFINED which external aborts, if any, are supported.

Normally, external aborts are rare. An imprecise asynchronous external abort is likely to be fatal to the process that is running, ARM recommends that implementations make external aborts precise wherever possible.

The following subsections give more information about possible external aborts:

- *External abort on an instruction fetch*.
- *External abort on data read or write*.
- *Provision for the classification of external aborts*.
- *Parity or ECC error reporting* on page D3-1728.

### D3.5.1 External abort on an instruction fetch

An external abort on an instruction fetch can be either synchronous or asynchronous.

A synchronous external abort on an instruction fetch is taken precisely using the Instruction Abort exception.

An implementation can report the external abort asynchronously from the instruction that it applies to. In such an implementation the abort is taken using the SError interrupt exception.

### D3.5.2 External abort on data read or write

Externally-generated errors that occur during a data read or write can be either synchronous or asynchronous.

A synchronous external abort on a data read or write is taken precisely using the Data Abort exception.

An implementation can report the external abort asynchronously from the instruction that generated the access. In such an implementation the abort is taken using the SError interrupt exception.

### D3.5.3 Provision for the classification of external aborts

In AArch64 state, an implementation can use ESR_ELx.EA, ISS[9], to provide more information about synchronous external aborts. For more information, see *Exception from an Instruction abort* on page D1-1549 and *Exception from a Data abort* on page D1-1549.

For all aborts other than synchronous external aborts reported using the EC values `0x20`, `0x21`, `0x24`, and `0x25`, ESR_ELx.EA, ISS[9], returns a value of 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D3.5.4 Parity or ECC error reporting

The ARM architecture supports the reporting of both synchronous and asynchronous parity or ECC errors from the cache system. It is IMPLEMENTATION DEFINED what parity or ECC errors in the cache systems, if any, result in synchronous or asynchronous parity or ECC errors.

A fault code is defined for reporting parity or ECC errors, see *Use of the ESR_EL1, ESR_EL2, and ESR_EL3* on page D1-1539. However, when parity or ECC error reporting is implemented, it is IMPLEMENTATION DEFINED whether a parity or ECC error is reported using the assigned fault code or using another appropriate encoding.

For all purposes other than the fault status encoding, parity or ECC errors are treated as external aborts.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D3.6 Memory barrier instructions

*Memory barriers* on page B2-86 describes the memory barrier instructions. This section describes the system level controls of those instructions.

### D3.6.1 EL2 control of the shareability of data barrier instructions executed at Non-secure EL0 or EL1

In an implementation that includes EL2 and supports shareability limitations on the data barrier instructions, the HCR_EL2.BSU field can upgrade the required shareability of an instruction that is executed at EL0 or EL1 in Non-secure state. Table D3-7 shows the encoding of this field.

**Table D3-7 EL2 control of shareability of barrier instructions executed at Non-secure EL0 or EL1**

| HCR_EL2.BSU | Minimum shareability of barrier instructions |
|---|---|
| 00 | No effect, shareability is as specified by the instruction |
| 01 | Inner Shareable |
| 10 | Outer Shareable |
| 11 | Full system |

For an instruction executed at EL0 or EL1 in Non-secure state, Table D3-8 shows how the HCR_EL2.BSU is combined with the shareability specified by the argument of the `DMB` or `DSB` instruction to give the scope of the instruction.

**Table D3-8 Effect of HCR_EL2.BSU on barrier instructions executed at Non-secure EL1 or EL0**

| Shareability specified by the `DMB` or `DSB` argument | HCR_EL2.BSU | Resultant shareability |
|---|---|---|
| Full system | Any | Full system |
| Outer Shareable | `00`, `01`, or `10` | Outer Shareable |
| | `11`, Full system | Full system |
| Inner Shareable | `00` or `01` | Inner Shareable |
| | `10`, Outer Shareable | Outer Shareable |
| | `11`, Full system | Full system |
| Non-shareable | `00`, No effect | Non-shareable |
| | `01`, Inner Shareable | Inner Shareable |
| | `10`, Outer Shareable | Outer Shareable |
| | `11`, Full system | Full system |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D3.7 Pseudocode description of general memory system instructions

This section contains the following pseudocode describing general memory operations:

- *Memory data type definitions*.
- *Basic memory access* on page D3-1731.
- *Aligned memory access* on page D3-1731.
- *Unaligned memory access* on page D3-1732.
- *Exclusive monitors operations* on page D3-1733.
- *Access permission checking* on page D3-1735.
- *Abort exceptions* on page D3-1736.
- *Memory barriers* on page D3-1738.

### D3.7.1 Memory data type definitions

This section describes the memory data type definitions.

The address descriptor type is defined as follows:

```
type AddressDescriptor is (
    FaultRecord      fault,      // fault.type indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    bits(64)         vaddress
)
```

The full address type is defined as follows:

```
type FullAddress is (
    bits(48) physicaladdress,
    bit      NS                  // '0' = Secure, '1' = Non-secure
)
```

The memory attributes types are defined as follows:

```
type MemoryAttributes is (
    MemType      type,

    DeviceType   device,     // For Device memory types
    MemAttrHints inner,      // Inner hints and attributes
    MemAttrHints outer,      // Outer hints and attributes

    boolean      shareable,
    boolean      outershareable
)
```

The memory type is defined as follows.

```
enumeration MemType {MemType_Normal, MemType_Device};
```

The Device memory types are defined as follows:

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

For Normal memory, the inner and outer attributes are defined as follows:

```
type MemAttrHints is (
    bits(2) attrs,  // The possible encodings for each attributes field are as below
    bits(2) hints,  // The possible encodings for the hints are below
    boolean transient
)
```

The cacheability attributes are defined as follows:

```
constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The allocation hints are defined as follows:

```
constant bits(2) MemHint_No = '00';     // No allocate
constant bits(2) MemHint_WA = '01';     // Write-allocate, Read-no-allocate
constant bits(2) MemHint_RA = '10';     // Read-allocate, Write-no-allocate
constant bits(2) MemHint_RWA = '11';    // Read-allocate and Write-allocate
```

The access permissions type is defined as follows:

```
type Permissions is (
    bits(3) ap,   // Access permission bits
    bit     xn,   // Execute-never bit
    bit     pxn   // Privileged execute-never bit
)
```

### D3.7.2 Basic memory access

The two `_Mem[]` accessors, Non-assignment (memory read) and Assignment (memory write), are the operations that perform single-copy atomic, aligned, little-endian memory accesses of `size` bytes to or from the underlying physical memory array of bytes.

```
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccType acctype];

_Mem[AddressDescriptor desc, integer size, AccType acctype] = bits(8*size) value;
```

The functions address the array using `desc.paddress` which supplies:

- A 48-bit physical address.
- A single NS bit to select between Secure and Non-secure parts of the array.

The `AccType` parameter describes the access type, such as normal, exclusive, ordered, and streaming. For a definition of `AccType`, see *Address space* on page B2-68.

The actual implemented array of memory might be smaller than the $2^{48}$ bytes implied. In this case the scheme for aliasing is IMPLEMENTATION DEFINED, or some parts of the address space might give rise to external aborts or a System Error.

The attributes in `memaddrdesc.memattrs` are used by the memory system to determine caching and ordering behaviors as described in *Memory types and attributes* on page B2-93, *Memory ordering* on page B2-83, and *Atomicity in the ARM architecture* on page B2-80.

`PAMax()` returns the IMPLEMENTATION DEFINED size of the physical address.

```
integer PAMax();
```

### D3.7.3 Aligned memory access

The `MemSingle[]` function makes an atomic, little-endian accesses of `size` bytes.

```
// AArch64.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Memory array access
    value = _Mem[memaddrdesc, size, acctype];
    return value;

// AArch64.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8)
value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    _Mem[memaddrdesc, size, acctype] = value;
    return;
```

### D3.7.4 Unaligned memory access

The `Mem[]` function makes an access of the required type. If that access is not architecturally defined to be atomic, it synthesizes accesses from multiple calls to `MemSingle[]`. It also reverses the byte order if the access is big-endian.

```
// Mem[] - non-assignment (read) form
// ==================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    integer i;
    boolean iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    atomic = (aligned && !(acctype IN {AccType_VEC, AccType_VECSTREAM})) || size == 1;

    if !atomic then
        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch64.MemSingle[address, size, acctype, aligned];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem[] - assignment (write) form
// ===============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
    integer i;
    boolean iswrite = TRUE;

    if BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    atomic = (aligned && !(acctype IN {AccType_VEC, AccType_VECSTREAM})) || size == 1;

    if !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    else
        AArch64.MemSingle[address, size, acctype, aligned] = value;
    return;
```

The `CheckAlignment()` function is:

```
// AArch64.CheckAlignment()
// ========================

boolean AArch64.CheckAlignment(bits(64) address, integer size, AccType acctype, boolean iswrite)

    aligned = (address == Align(address, size));
    A = SCTLR[].A;

    if !aligned && (acctype == AccType_ATOMIC || acctype == AccType_ORDERED || A == '1') then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

### D3.7.5 Exclusive monitors operations

The `SetExclusiveMonitors()` function sets the exclusive monitors for a block of bytes, the size of which is determined by `size`, at the virtual address defined by `address`.

```
// AArch64.SetExclusiveMonitors()
// ==============================

// Sets the Exclusive Monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)

    acctype = AccType_ATOMIC;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    iswrite = FALSE;
    aligned = (address != Align(address, size));

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

The `ExclusiveMonitorsPass()` function checks whether the exclusive monitors are set to include the location of a number of bytes specified by `size`, at the virtual address defined by `address`. The atomic write that follows after the exclusive monitors have been set must be to the same physical address. It is permitted, but not required, for this function to return FALSE if the virtual address is not the same as that used in the previous call to `SetExclusiveMonitors()`.

```
// AArch64.ExclusiveMonitorsPass()
// ===============================

// Return TRUE if the Exclusive Monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusive Monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

The `ExclusiveMonitorsStatus()` function returns 0 if the previous atomic write was to the same physical memory locations selected by `ExclusiveMonitorsPass()` and therefore succeeded. Otherwise the function returns 1, indicating that the address translation delivered a different physical address.

```
bit ExclusiveMonitorsStatus();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The MarkExclusiveGlobal() procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure records that the PE processorid has requested exclusive access covering at least size bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB and no smaller than two words, and aligned in the address space to the size of the location. It is UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same PE to be cleared.

```
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

The MarkExclusiveLocal() procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure records in a local record that PE processorid has requested exclusive access to an address covering at least size bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory but is no smaller than two words, and is aligned in the address space to the size of the location. It is IMPLEMENTATION DEFINED whether this procedure also performs a MarkExclusiveGlobal() using the same parameters.

```
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

The IsExclusiveGlobal() function takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The function returns TRUE if the PE processorid has marked in a global record an address range as exclusive access requested that covers at least size bytes from address paddress. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, IsExclusiveGlobal() returns FALSE.

```
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

The IsExclusiveLocal() function takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The function returns TRUE if the PE processorid has marked an address range as exclusive access requested that covers at least the size bytes from address paddress. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of size bytes from address paddress. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of IsExclusiveGlobal() with the same parameters.

```
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

The ClearExclusiveByAddress() procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure clears the global records of all PEs, other than processorid, for which an address region including any of size bytes starting from paddress has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether the equivalent global record of the PE processorid is also cleared if any of size bytes starting from paddress has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

```
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

The ClearExclusiveLocal() procedure takes as arguments the PE identifier processorid. The procedure clears the local record of PE processorid for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE processorid that an address has had a request for an exclusive access.

```
ClearExclusiveLocal(integer processorid);
```

### D3.7.6 Access permission checking

The function CheckPermission() is used by the architecture to perform access permission checking based on attributes derived from the translation tables or location descriptors. It returns the result of the call to AArch64.NoFault().

The interpretation of access permission is shown in *Memory access control* on page D4-1808.

The pseudocode function for checking access permissions is as follows:

```
// AArch64.CheckPermission()
// =========================
// Function used for permission checking from AArch64 stage 1 translations
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
FaultRecord AArch64.CheckPermission(Permissions perms, bits(64) vaddress, integer level,
                                    bit NS, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    wxn = SCTLR[].WXN == '1';

    if PSTATE.EL IN {EL0,EL1} then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';
        user_xn = perms.xn == '1' || (user_w && wxn);
        priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;
        ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2 or EL3
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
        xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(48) UNKNOWN;
        return AArch64.PermissionFault(ipaddress, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

### D3.7.7 Abort exceptions

The `Abort()` function generates either a Data Abort or an Instruction Abort exception by calling `AArch64.DataAbort()` or `AArch64.InstructionAbort()`. It also can generate a debug exception for debug related faults, see Chapter D2 *AArch64 Self-hosted Debug*.

```
// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

    if IsDebugException(fault) then
        if fault.acctype == AccType_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
        else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.WatchpointException(vaddress, fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
```

The `DataAbort()` function generates a Data Abort exception, routes the exception to EL2 or EL3, and records the information required for the Exception Syndrome registers, ESR_ELx. See *Exception from a Data abort* on page D1-1549. A second stage abort might also record the intermediate physical address, IPA, but this depends on the type of the abort.

For a synchronous abort, `DataAbort()` also sets the FAR to the VA of the abort.

The pseudocode for the `DataAbort()` function is as follows:

```
// AArch64.DataAbort()
// ===================

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || IsSecondStage(fault)));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

The `InstructionAbort()` function generates an Instruction Abort exception, routes the exception to EL2 or EL3, and records the information required for the Exception Syndrome registers, ESR_ELx. See *Exception from an Instruction abort* on page D1-1549. A second stage abort might also record the intermediate physical address, IPA, but this depends on the type of the abort.

For a synchronous abort, `InstructionAbort()` also sets the FAR to the VA of the abort.

The pseudocode for the `InstructionAbort()` function is as follows:

```
// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || IsSecondStage(fault)));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The FaultRecord type describes a fault. Functions that check for faults return a record of this type appropriate to the type of fault. *Pseudocode description of the MMU faults* on page D4-1832 provides a number of wrappers to generate FaultRecords.

The NoFault() function returns a null record that indicates no fault. The IsFault() function tests whether a FaultRecord contains a fault.

```
enumeration Fault {Fault_None,
                   Fault_AccessFlag,
                   Fault_Alignment,
                   Fault_Background,
                   Fault_Domain,
                   Fault_Permission,
                   Fault_Translation,
                   Fault_AddressSize,
                   Fault_SyncExternal,
                   Fault_SyncExternalOnWalk,
                   Fault_SyncParity,
                   Fault_SyncParityOnWalk,
                   Fault_AsyncParity,
                   Fault_AsyncExternal,
                   Fault_Debug,
                   Fault_TLBConflict,
                   Fault_Lockdown,
                   Fault_Exclusive,
                   Fault_ICacheMaint};
type FaultRecord is (Fault    type,         // Fault Status
                     AccType  acctype,      // Type of access that faulted
                     bits(48) ipaddress,    // Intermediate physical address
                     boolean  s2fs1walk,    // Is on a Stage 1 page table walk
                     boolean  write,        // TRUE for a write, FALSE for a read
                     integer  level,        // For translation, access flag and permission faults
                     bit      extflag,      // IMPLEMENTATION DEFINED syndrome for external aborts
                     boolean  secondstage,  // Is a Stage 2 abort
                     bits(4)  domain,       // Domain number, AArch32 only
                     bits(4)  debugmoe)     // Debug method of entry, from AArch32 only
// AArch64.NoFault()
// =================

FaultRecord AArch64.NoFault()

    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_None, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
// IsFault()
// =========
// Return true if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.type != Fault_None;
```

### D3.7.8 Memory barriers

The definition for the memory barrier functions is:

```
enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};

enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These functions define the required shareability domains and required access types used as arguments for DMB and DSB instructions.

The following procedures perform the memory barriers:

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);

InstructionSynchronizationBarrier();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter D4
# The AArch64 Virtual Memory System Architecture

This chapter provides a system level view of the AArch64 Virtual Memory System Architecture (VMSA), the memory system architecture of an ARMv8 implementation that is executing in AArch64 state. It contains the following sections:


- *About the Virtual Memory System Architecture (VMSA)* on page D4-1742.
- *The VMSAv8-64 address translation system* on page D4-1745.
- *VMSAv8-64 translation table format descriptors* on page D4-1799.
- *Access controls and memory region attributes* on page D4-1808.
- *MMU faults* on page D4-1824.
- *Translation Lookaside Buffers (TLBs)* on page D4-1833.
- *TLB maintenance requirements and the TLB maintenance instructions* on page D4-1837.
- *Caches in a VMSA implementation* on page D4-1850.




*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D4.1 About the Virtual Memory System Architecture (VMSA)

This chapter describes the *Virtual Memory System Architecture* (VMSA) that applies to a PE executing in AArch64 state. This is VMSAv8-64, as defined in *ARMv8 VMSA naming* on page D4-1747.

A VMSA provides a *Memory Management Unit* (MMU), that controls address translation, access permissions, and memory attribute determination and checking, for memory accesses made by the PE.

The process of address translation maps the *virtual addresses* (VAs) used by the PE onto the *physical addresses* (PAs) of the physical memory system. These translations are defined independently for different Exception levels and Security states, and Figure D4-1 shows.

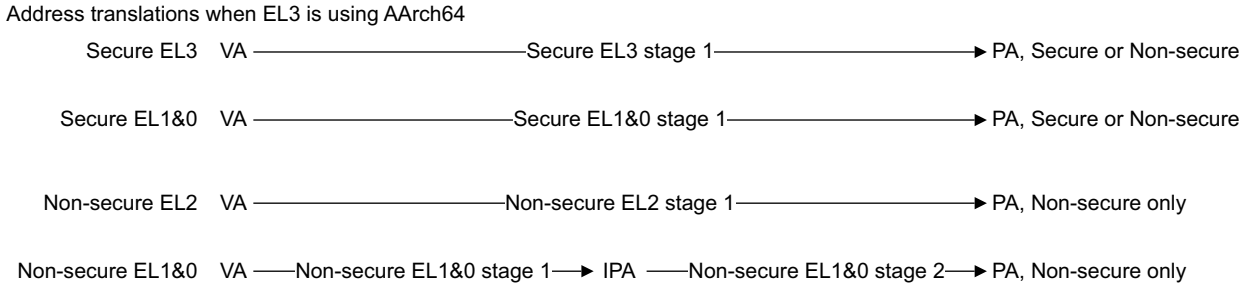



**Figure D4-1 Address translations for different Exception levels and Security states**

VMSAv8-64 supports tagging of VAs, as described in *Address tagging in AArch64 state*. As that section describes, this address tagging has no effect on the address translation process.

The remainder of this chapter gives a full description of VMSAv8-64 for an implementation that includes all of the Exception levels. *The implemented Exception levels and the resulting translation stages and regimes* on page D4-1783 describes the differences in the VMSA if some Exception levels are not implemented.

### D4.1.1 Address tagging in AArch64 state

In AArch64 state, the ARMv8 architecture supports tagged addresses for data values. In these cases the top eight bits of the virtual address are ignored when determining:

- Whether the address causes a Translation fault from being out of range if the translation system is enabled.
- Whether the address causes an Address size fault from being out of range if the translation system is not enabled.
- Whether the address requires invalidation when performing a TLB invalidation instruction by address.

The use of address tags is controlled as follows:

**For addresses using the VMSAv8-64 EL1&0 translation regime**

The value of bit[55] of the VA determines the register bit that controls the use of address tags, as follows:

**VA[55]==0** TCR_EL1.TBI0 determines whether address tags are used. If stage 1 translation is enabled, TTBR0_EL1 holds the base address of the translation tables used to translate the address.

**VA[55]==1** TCR_EL1.TBI1 determines whether address tags are used. If stage 1 translation is enabled, TTBR1_EL1 holds the base address of the translation tables used to translate the address.

**For addresses using the VMSAv8-64 EL2 translation regime**

TCR_EL2.TBI determines whether address tags are used. If stage 1 translation is enabled, TTBR0_EL2 holds the base address of the translation tables used to translate the address.

**For addresses using the VMSAv8-64 EL3 translation regime**

TCR_EL3.TBI determines whether address tags are used. If stage 1 translation is enabled, TTBR0_EL3 holds the base address of the translation tables used to translate the address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The TCR_ELx.TBI*n* bits determine whether address tags are used regardless of whether the corresponding translation regime is enabled.

An address tag enable bit also has an effect on the PC value in the following cases:

- Any branch or procedure return within the controlled Exception level.
- On taking an exception to the controlled Exception level, regardless of whether this is also the Exception level from which the exception was taken.
- On performing an exception return to the controlled Exception level, regardless of whether this is also the Exception level from which the exception return was performed.
- Exiting from debug state to the controlled Exception level.

**Note**

As an example of what is meant by the *controlled Exception level*, TCR_EL2.TBI controls this effect for:

- A branch or procedure return within EL2.
- Taking an exception to EL2.
- Performing an exception return or a debug state exit to EL2.

The effect of the controlling TBI{*n*} bit is:

**For EL0 or EL1** If the controlling TBI*n* bit for the address being loaded into the PC is set to 1, then bits[63:56] of the PC are forced to be a sign-extension of bit[55] of that address.

**For EL2 or EL3** If the controlling TBI bit for the address being loaded into the PC is set to 1, then bits[63:56] of the PC are forced to be `0x00`.

The `AddrTop()` pseudocode function shows the algorithm determining the most significant bit of the VA, and therefore whether the virtual address is using tagging. For the EL1&0 translation regime, this pseudocode includes the selection between TTBR0_EL1 and TTBR1_EL1 described in *Selection between TTBR0 and TTBR1* on page D4-1774.

```
// AddrTop()
// =========

integer AddrTop(bits(64) address)
    // Return the MSB number of a virtual address in the current stage 1 translation
    // regime. If EL1 is using AArch64 then addresses from EL0 using AArch32
    // are zero-extended to 64 bits.
    if UsingAArch32() && !(PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) then
        // AArch32 translation regime.
        return 31;
    else
        // AArch64 translation regime.
        case PSTATE.EL of
            when EL0, EL1
                tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            when EL2
                tbi = TCR_EL2.TBI;
            when EL3
                tbi = TCR_EL3.TBI;
        return (if tbi == '1' then 55 else 63);
```

**Note**

The required behavior prevents a tagged address being propagated to the program counter.

When address tagging is enabled for an address that causes a Data Abort or a Watchpoint, the address tag is included in the virtual address returned in the FAR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Relaxation of the tagged address handling requirements on an Illegal exception return

The `AddrTop()` pseudocode function, and the pseudocode description of exception return, does not cover a relaxation to the requirements for tagged address handling that applies to an Illegal exception return. The following pseudocode describes the algorithm for the address branched to, ensuring that any address tag is not propagated to the PC:

```
if (target_exception_level == EL0) || (target_exception_level == EL1) then
    if NewAddress<55> == '1' && TCR_EL1.TBI1== '1' then
        NewAddress<63:56> = '11111111';
    if NewAddress<55> == '0' && TCR_EL1.TBI0== '1' then
        NewAddress<63:56> = '00000000';
if (target_exception_level == EL2) then
    if TCR_EL2.TBI== '1' then
        NewAddress<63:56> = '00000000';
if (target_exception_level == EL3) then
    if TCR_EL3.TBI== '1' then
        NewAddress<63:56> = '00000000';
PC = NewAddress;
```

In this pseudocode:

`NewAddress` Is the address being branched to, or returned to.

`target_exception_level`

- Is the current Exception level for a branch or procedure return.
- Is the Exception level being returned to for an exception return.

  If the exception return triggers the Illegal exception return mechanism, it is IMPLEMENTATION DEFINED whether `target_exception_level` is the Exception level that was described in the SPSR at the time of the exception return or the Exception level that the exception return instruction was executed from.
- Is the Exception level the exception is taken to for an exception entry

**Note**

- The TCR_ELx.TBIx fields have the effect shown in the pseudocode regardless of whether the corresponding translation regime is enabled.
- In the case of an Illegal exception return, the tag bits of the address can be propagated to the PC if all of the following apply:
  - The implementation treats the `target_exception_level` as being the Exception level that was described in the SPSR at the time of the exception return.
  - For the Exception level that was described in the SPSR at the time of the exception return, the value of TCR_ELx.TBI is 0.
  - In the Exception level that the exception was taken from, the value of TCR_ELx.TBI is 1.

  In all other cases, the tag bits cannot be propagated to the PC.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D4.2 The VMSAv8-64 address translation system

This section describes the VMSAv8-64 address translation system, that maps VAs to PAs. Related to this:

- *VMSAv8-64 translation table format descriptors* on page D4-1799 describes the translation table entries.
- *Access controls and memory region attributes* on page D4-1808 describes the attributes that are held in the translation table entries, including how different attributes can interact.
- *Translation Lookaside Buffers (TLBs)* on page D4-1833 describes the caching of translation table lookups in TLBs, and the architected instructions for maintaining TLBs.
- *AArch64 Address translation examples* on page K7-5572 gives detailed descriptions of typical examples of translating a VA to a final PA, and obtaining the memory attributes of that PA.

In this section, the following subsections describe the VMSAv8-64 address translation system:

- *About the VMSAv8-64 address translation system*.
- *Controlling address translation stages* on page D4-1749.
- *Memory translation granule size* on page D4-1755.
- *Translation tables and the translation process* on page D4-1760.
- *Overview of the VMSAv8-64 address translation stages* on page D4-1762.
- *The VMSAv8-64 translation table format* on page D4-1771.
- *The algorithm for finding the translation table entries* on page D4-1778.
- *The effects of disabling a stage of address translation* on page D4-1781.
- *The implemented Exception levels and the resulting translation stages and regimes* on page D4-1783.
- *Pseudocode description of VMSAv8-64 address translation* on page D4-1784.
- *Address translation instructions* on page D4-1795.

### D4.2.1 About the VMSAv8-64 address translation system

The *Memory Management Unit* (MMU) controls address translation, memory access permissions, and memory attribute determination and checking, for memory accesses made by the PE.

The general model of MMU operation is that the MMU takes information about a required memory access, including an *input address* (IA), and either:

- Returns an associated *output address* (OA), and the *memory attributes* for that address.
- Is unable to perform the translation for one of a number of reasons, and therefore causes an exception to be generated. This exception is called an MMU fault. An MMU fault is generated by a particular stage of translation, and can be described as either a stage 1 MMU fault or a stage 2 MMU fault.

The process of mapping an IA to an OA is an *address translation*, or more precisely a single stage of address translation.

The architecture defines a number of *translation regimes*, where a translation regime comprises either:

- A single stage of address translation.

  This maps an input *virtual address* (VA) to an output *physical address* (PA).
- Two, sequential, stages of address translation, where:
  - Stage 1 maps an input VA to an output *intermediate physical address* (IPA).
  - Stage 2 maps an input IPA to an output PA.

The *translation granule* specifies the granularity of the mapping from IA to OA. That is, it defines both:

- The *page size* for a stage of address translation, where a page is the smallest block of memory for which an IA to OA mapping can be specified.
- The size of a complete translation table for that stage of address translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The MMU is controlled by System registers, that provide independent control of each address translation stage, including a control to disable the stage of address translation. *The effects of disabling a stage of address translation* on page D4-1781 defines how the MMU handles an access for which a required address translation stage is disabled.

**Note**

- In the ARM architecture, a software agent, such as an operating system, that uses or defines stage 1 memory translations, might be unaware of the second stage of translation, and of the distinction between IPA and PA.
- A more generalized description of the translation regimes is that a regime always comprises two sequential stages of translation, but in some regimes the stage 2 translation both:
  - Returns an OA that equals the IA. This is called a *flat mapping* of the IA to the OA.
  - Does not change the memory attributes returned by the stage 1 address translation.

For an access to a stage of address translation that does not generate an MMU fault, the MMU translates the IA to the corresponding OA. System registers are used to report any faults that occur on a memory access.

This section describes the address translation system for an implementation that includes all of the Exception levels, and gives a complete description of translations that are controlled by an Exception level that is using AArch64.

Figure D4-2 shows these translation stages and translation regimes when EL3 is using AArch64.

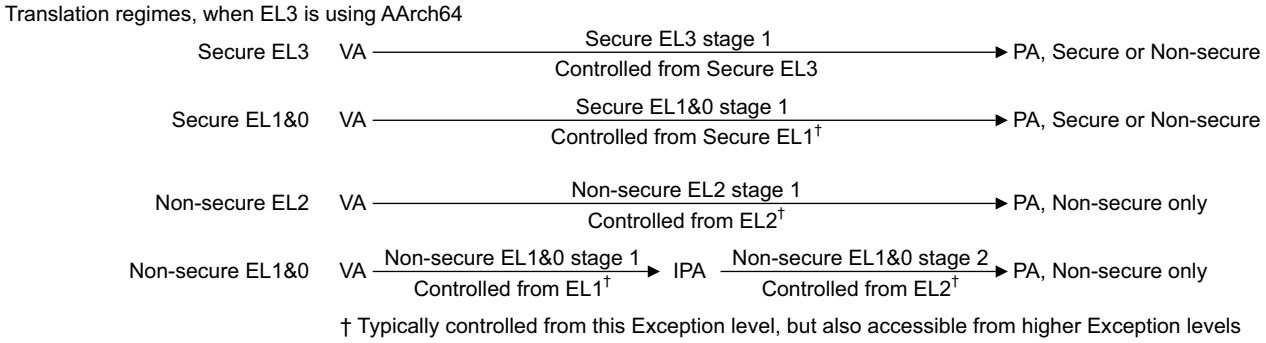



**Figure D4-2 VMSAv8 AArch64 translation regimes, translation stages, and associated controls**

*ARMv8 VMSA naming* on page D4-1747 gives more information about the options for the different stages of address translation shown in Figure D4-2, and:

- Chapter G4 *The AArch32 Virtual Memory System Architecture* describes:
  - The translation stages and translation regimes when EL3 is using AArch32.
  - Any stages of address translation that are using VMSAv8-32 when EL3 is using AArch64.
- *The implemented Exception levels and the resulting translation stages and regimes* on page D4-1783 describes the effect on the address translation model when some Exception levels are not implemented.

Each enabled stage of address translation uses a set of address translations and associated memory properties held in memory mapped tables called *translation tables*. A single translation table lookup can resolve only a limited number of bits of the IA, and therefore a single address translation can require multiple lookups. These are described as different *levels* of lookup.

Translation table entries can be cached in a *Translation Lookaside Buffer* (TLB).

As well as defining the OA that corresponds to the IA, the translation table entries define the following properties:

- Access to the Secure or Non-secure address map, for accesses made from Secure state.
- Memory access permission control.
- Memory region attributes.

For more information, see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D4-1803.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following subsections give more information:

- *ARMv8 VMSA naming*.
- *VMSA address types and address spaces*.
- *About address translation* on page D4-1748.
- *The VMSAv8-64 translation table format* on page D4-1748.

## ARMv8 VMSA naming

The ARMv8 VMSA naming model reflects the possible stages of address translation, as follows:

**VMSAv8** The overall translation scheme, within which an address translation has one or two stages.

**VMSAv8-32** The translation scheme for a single stage of address translation that is managed from an Exception level that is using AArch32.

**VMSAv8-64** The translation scheme for a single stage of address translation that is managed from an Exception level that is using AArch64.

## VMSA address types and address spaces

A description of the VMSA refers to the following address types.

**Note**

These descriptions relate to the VMSAv8 description and therefore give more detail than the generic definitions given in the glossary.

**Virtual address (VA)**

An address used in an instruction, as a data or instruction address, is a Virtual Address (VA).

**Note**

This means that an address held in the PC, LR, SP, or an ELR, is a VA.

In AArch64 state, the VA address space has a maximum address width of 48 bits. With a single VA range this gives a maximum VA space of 256TB, with VA range of `0x0000_0000_0000_0000` to `0x0000_FFFF_FFFF_FFFF`.

However, for the EL1&0 translation stage the VA range is split into two subranges, one at the bottom of the full 64-bit address range of the PC, and one at the top, as follows:

- The bottom VA range runs up from address `0x0000_0000_0000_0000`. With the maximum address width of 48 bits this gives a VA range of `0x0000_0000_0000_0000` to `0x0000_FFFF_FFFF_FFFF`.
- The top VA subrange runs up to address `0xFFFF_FFFF_FFFF_FFFF`. With the maximum address width of 48 bits this gives a VA range of `0xFFFF_0000_0000_0000` to `0xFFFF_FFFF_FFFF_FFFF`. Reducing the address width for this subrange increases the bottom address of the range.

This means that there are two VA subranges, each of up to 256TB.

Each translation regime, that takes a VA as an input address, can be configured to support fewer than 48 bits of virtual address space, see *Address size configuration* on page D4-1750.

**Intermediate physical address (IPA)**

In a translation regime that provides two stages of address translation, the IPA is:

- The OA from the stage 1 translation.
- The IA for the stage 2 translation.

In a translation regime that provides only one stage of address translation, the IPA is identical to the PA. Alternatively, the translation regime can be considered as having no concept of IPAs.

The IPA address space has a maximum address width of 48 bits, see *Address size configuration* on page D4-1750.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Physical address (PA)**

The address of a location in a physical memory map. That is, an output address from the PE to the memory system.

The EL3 and Secure EL1 Exception levels provide independent definitions of physical address spaces for Secure and Non-secure operation. This means they provide two independent address spaces, where:

- A VA accessed in Secure state can be translated to either the Secure or the Non-secure physical address space.
- When in Non-secure state, a VA is always mapped to the Non-secure physical address space.

Each PA address space has a maximum address width of 48 bits, but an implementation can implement fewer than 48 bits of physical address. See *Address size configuration* on page D4-1750.

## About address translation

For a single stage of address translation, a *Translation table base register* (TTBR) indicates the start of the first translation table required for the mapping from input address to output address. Each implemented translation stage shown in *VMSAv8 AArch64 translation regimes, translation stages, and associated controls* on page D4-1746 requires its own set of translation tables.

For the EL1&0 stage 1 translation, the split of the VA mapping into two subranges requires two tables, one for the lower part of the VA space, and the other for the upper part of the VA space. *Example use of the split VA range, and the TTBR0_EL1 and TTBR1_EL1 controls* on page D4-1775 shows how these ranges might be used.

*Controlling address translation stages* on page D4-1749 summarizes the System registers that control address translation by the MMU.

A full translation table lookup is called a *translation table walk*. It is performed automatically by hardware, and can have a significant cost in execution time. To support fine granularity of the VA to PA mapping, a single IA to OA translation can require multiple accesses to the translation tables, with each access giving finer granularity. Each access is described as a *level* of address lookup. The final level of the lookup defines:

- The high bits of the required output address.
- The *attributes* and *access permissions* of the addressed memory.

Translation table entries can be cached in a *Translation Lookaside Buffer*, see *Translation Lookaside Buffers (TLBs)* on page D4-1833.

## The VMSAv8-64 translation table format

Stages of address translation that are controlled by an Exception level that is using AArch64 use the VMSAv8-64 translation table format. This format uses 64-bit descriptor entries in the translation tables.

**Note**

This format is an extension of the VMSAv8-32 Long-descriptor translation table format originally defined by the ARMv7 Large Physical Address Extension, and extended slightly by ARMv8. VMSAv8-32 also supports a Short-descriptor translation table format. Chapter G4 *The AArch32 Virtual Memory System Architecture* describes both of these formats.

The VMSAv8-64 translation table format provides:

- Up to four levels of address lookup.
- Input addresses of up to 48 bits.
- Output addresses of up to 48 bits.
- A translation granule size of 4KB, 16KB, or 64KB.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D4.2.2 Controlling address translation stages

*The implemented Exception levels and the resulting translation stages and regimes* on page D4-1783 defines the translation regimes and stages. For each supported address translation stage:

- A System register bit enables the stage of address translation.
- A System register bit determines the endianness of the translation table lookups.
- A *Translation Control Register* (TCR) controls the stage of address translation.
- If a stage of address translation supports splitting the VA range into two subranges then that stage of translation provides a TTBR for each VA subrange, and the stage of address translation has:
  — A single TCR.
  — A TTBR for each VA subrange.

  Otherwise, a single TTBR holds the address of the translation table that must be used for the first lookup for the stage of address translation.

For address translation stages controlled from AArch64:

- Table D4-1 shows the endianness bit and the enable bit for each stage of address translation. Each register entry in the table gives the endianness bit followed by the enable bit. Except for the Non-secure EL1&0 stage 2 translation, these two bits are in the same register.

**Table D4-1 Enable and endianness bits for the AArch64 translation stages**

| Translation stage | Controlled from | Controlling register |
|---|---|---|
| Secure EL3 stage 1 | EL3 | SCTLR_EL3.{EE, M} |
| Secure EL1&0 stage 1 | Secure EL1 | SCTLR_EL1.{EE, M} |
| Non-secure EL2 stage 1 | EL2 | SCTLR_EL2.{EE, M} |
| Non-secure EL1&0 stage 2 | EL2 | SCTLR_EL2.EE, HCR_EL2.VM |
| Non-secure EL1&0 stage 1 | Non-secure EL1 | SCTLR_EL1.{EE, M} |

**Note**

If the PA of the software that enables or disables a particular stage of address translation differs from its VA, speculative instruction fetching can cause complications. ARM strongly recommends that the PA and VA of any software that enables or disables a stage of address translation are identical if that stage of translation controls translations that apply to the software currently being executed.

- Table D4-2 shows the TCR and TTBR, or TTBRs, for each stage of address translation. In the table, each *Controlling registers* entry gives the TCR followed by the TTBR or TTBRs.

**Table D4-2 TCRs and TTBRs for the AArch64 translation stages**

| Translation stage | Controlled from | Controlling registers |
|---|---|---|
| Secure EL3 stage 1 | EL3 | TCR_EL3, TTBR0_EL3 |
| Secure EL1&0 stage 1 | Secure EL1 | TCR_EL1, TTBR0_EL1, TTBR1_EL1 |
| Non-secure EL2 stage 1 | EL2 | TCR_EL2, TTBR0_EL2 |
| Non-secure EL1&0 stage 2 | EL2 | VTCR_EL2, VTTBR_EL2 |
| Non-secure EL1&0 stage 1 | Non-secure EL1 | TCR_EL1, TTBR0_EL1, TTBR1_EL1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following subsections give more information about controlling address translation:

- *System registers relevant to MMU operation*.
- *Address size configuration*.
- *Atomicity of register changes on changing virtual machine* on page D4-1754.
- *Use of out-of-context translation regimes* on page D4-1754.

## System registers relevant to MMU operation

In AArch64 state, System registers have a suffix, that indicates the lowest Exception level from which they can be accessed. In some general descriptions of MMU control and address translation, this chapter uses a *Common abbreviation* for each of the System registers that affects MMU operation, as Table D4-3 shows. The common abbreviation is used when describing features that apply to all the translation regimes.

**Note**

The only translation regime that supports a stage 2 translation is the Non-secure EL1&0 translation regime.

**Table D4-3 Abbreviations for System registers used in this chapter**

| Common abbreviation | Translation stage | Exception level | | |
|---|---|---|---|---|
| | | EL1 | EL2 | EL3 |
| HCR | - | - | HCR_EL2 | - |
| SCTLR | - | SCTLR_EL1 | SCTLR_EL2 | SCTLR_EL3 |
| TCR | Stage 1 | TCR_EL1 | TCR_EL2 | TCR_EL3 |
| | Stage 2 | - | VTCR_EL2 | - |
| TTBR | Stage 1 | TTBR0_EL1, TTBR1_EL1 | TTBR0_EL2 | TTBR0_EL3 |
| | Stage 2 | - | VTTBR_EL2 | - |

## Address size configuration

The following subsubsections specify the configuration of the physical address size and of the input and output address sizes for each of the stages of address translation:

- *Physical address size*.
- *Output address size* on page D4-1751.
- *Input address size* on page D4-1752.
- *Supported IPA size* on page D4-1753.

### *Physical address size*

The ID_AA64MMFR0_EL1.PARange field indicates the implemented physical address size, as Table D4-4 shows.

**Table D4-4 Physical address size implementation options**

| ID_AA64MMFR0_EL1.PARange | Total PA size | PA address size |
|---|---|---|
| 0000 | 4 GB | 32 bits, PA[31:0] |
| 0001 | 64 GB | 36 bits, PA[35:0] |
| 0010 | 1 TB | 40 bits, PA[39:0] |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D4-4 Physical address size implementation options (continued)**

| ID_AA64MMFR0_EL1.PARange | Total PA size | PA address size |
|---|---|---|
| 0011 | 4 TB | 42 bits, PA[41:0] |
| 0100 | 16 TB | 44 bits, PA[43:0] |
| 0101 | 256 TB | 48 bits, PA[47:0] |

All other PARange values are reserved.

### *Output address size*

For each enabled stage of address translation, TCR.{I}PS must be programmed to maximum output address size for that stage of translation, using the encodings as shown in Table D4-5.

**Table D4-5 Output address size implementation options**

| TCR.{I}PS | Total output size | Output address size |
|---|---|---|
| 000 | 4 GB | 32 bits, PA[31:0] |
| 001 | 64 GB | 36 bits, PA[35:0] |
| 010 | 1 TB | 40 bits, PA[39:0] |
| 011 | 4 TB | 42 bits, PA[41:0] |
| 100 | 16 TB | 44 bits, PA[43:0] |
| 101 | 256 TB | 48 bits, PA[47:0] |

**Note**

- This field is called IPS in the TCR_EL1, and PS in the other TCRs.
- The {I}PS fields are 3-bit fields, corresponding to the least-significant PARange bits shown in Table D4-4 on page D4-1750.

If {I}PS is programmed to a value larger than the implemented physical address size, then the PE behaves as if programmed with the implemented physical address size, but software must not rely on this behavior. That is, the output address size is never larger than the implemented physical address size. Table D4-4 on page D4-1750 shows the implemented physical address size.

The PE checks that the TTBR, translation table entries, and the output address for the stage of address translation have the address bits above the output address size set to zero. If this is not the case, an Address size fault is generated for the level and stage of translation that caused the fault. An Address size fault from the TTBR is reported as a level 0 fault.

If stage 1 translation is disabled and the input address is larger than the implemented physical address size, then a stage 1 level 0 Address size fault is generated.

When using two stages of translation:

- If stage 2 translation is disabled and the output address from the stage 1 translation is larger than the implemented physical address size, then a stage 1 Address size fault is generated for the level of the stage 1 translation that generated the output address.
- If stage 2 translation is enabled and the output address from the stage 1 translation does not generate a stage 1 Address Size fault, but is larger than the input address size specified for the stage 2 translation, then a stage 2 Translation fault is generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Input address size

For each enabled stage of address translation, the TCR.T*x*SZ fields specify the input address size:

- TCR_EL1 has two T*x*SZ fields, corresponding to the two VA subranges:
  - TCR_EL1.T0SZ specifies the size for the lower VA range, translated using TTBR0_EL1.
  - TCR_EL1.T1SZ specifies the size for the upper VA range, translated using TTBR1_EL1.
- Each of the other TCRs has a single T0SZ field, and input addresses are translated using a single TTBR.

Attempting to translate an address that is larger than the configured input address size generates a Translation fault. This means:

- For a TCR with a single T0SZ field, Figure D4-3 shows the input address map:

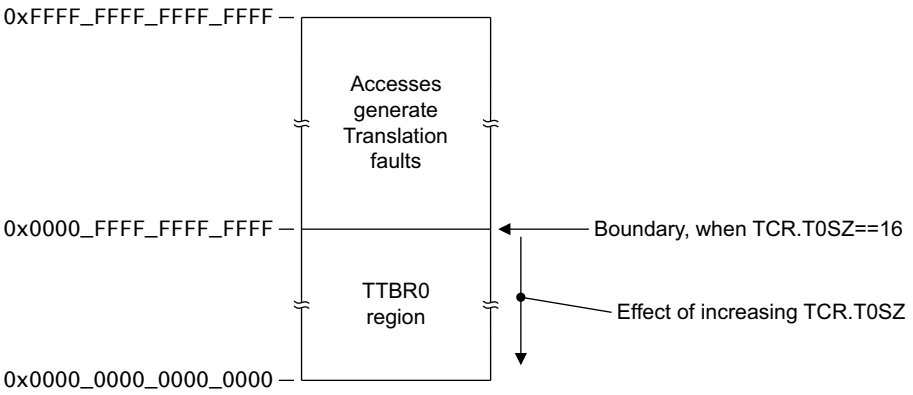



**Figure D4-3 AArch64 input address map when using a single TTBR**

- For a TCR with two T*x*SZ fields, the input address is always a VA, and *Selection between TTBR0 and TTBR1* on page D4-1774 describes the VA address map.

For the Non-secure EL1&0 translation regime, when both stages of translation are enabled, if the output address from the stage 1 translation does not generate a stage 1 address size fault, and is larger than the input address specified by VTCR_EL2.T0SZ, then the input address size check for the stage 2 translation generates a Translation fault.

Although software can configure the input address size to be smaller than 48 bits, all implemented AArch64 TTBRs must support address sizes of up to 48 bits.

*Overview of the VMSAv8-64 address translation stages* on page D4-1762 gives more information about the relationship between the required input address size, the value of T*x*SZ, and the required initial lookup level, and how these are affected by the translation granule size. However:

**For all translation stages**

The maximum T*x*SZ value is 39. If T*x*SZ is programmed to a value larger than 39 then it is IMPLEMENTATION DEFINED whether:

- The implementation behaves as if the field is programmed to 39 for all purposes other than reading back the value of the field.
- Any use of the T*x*SZ value generates a Level 0 Translation fault for the stage of translation at which T*x*SZ is used.

**For a stage 1 translation**

The minimum T*x*SZ value is 16. If T*x*SZ is programmed to a value smaller than 16 then it is IMPLEMENTATION DEFINED whether:

- The implementation behaves as if the field were programmed to 16 for all purposes other than reading back the value of the field.
- Any use of the T*x*SZ value generates a stage 1 Level 0 Translation fault.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**For a stage 2 translation**

*Supported IPA size* defines the effective minimum value of T0SZ, that depends on the supported PA size, and also describes the possible effects of programming T0SZ to a value that is smaller than this effective minimum value.

### *Supported IPA size*

For the Non-secure EL1&0 translation regime, the maximum IPA size is the maximum input address size for the second stage of translation, that must be specified by VTCR_EL2.T0SZ, see *Input address size* on page D4-1752. This value is constrained by the implemented PA size that is specified by ID_AA64MMFR0_EL1.PARange, see *Physical address size* on page D4-1750. This implemented PA size also constrains the maximum value of VTCR_EL2.SL0, that specifies the level of the initial lookup. SL0 also depends on the translation granule, as described in *Overview of the VMSAv8-64 address translation stages* on page D4-1762.

**Table D4-6 PA size implications for the VTCR_EL2.{T0SZ, SL0} fields**

| Supported PA size | Effective minimum T0SZ value | Maximum SL0 value | | |
|---|---|---|---|---|
| | | 4KB granule | 16KB granule | 64KB granule |
| 32 bits | 32 if EL1 is using AArch64<br>24 if EL1 is using AArch32 | 1 | 1 | 1 |
| 36 bits | 28 if EL1 is using AArch64<br>24 if EL1 is using AArch32 | 1 | 1 | 1 |
| 40 bits | 24 | 1 | 1 | 1 |
| 42 bits | 22 | 1 | 2 | 1 |
| 44 bits | 20 | 2 | 2 | 2 |
| 48 bits | 16 | 2 | 2 | 2 |

If VTCR_EL2.SL0 is programmed to a value larger than the maximum value shown in Table D4-6 then any memory access that uses the second stage of translation generates a stage 2 level 0 Translation fault.

If VTCR_EL2.T0SZ is programmed to a value smaller than the effective minimum value shown in Table D4-6 then the implementation consistently does one of the following:

- Treat the VTCR_EL2.T0SZ field as being programmed to the effective minimum value for all purposes other than reading back the value of the field.
- Treat the VTCR_EL2.T0SZ field as being programmed to the effective minimum value for all purposes other than:
  - Reading back the value of the field.
  - Checking whether the value of VTCR_EL2.T0SZ is consistent with the value of VTCR_EL2.SL0.
- Generate a stage 2 level 0 Translation fault on any memory access that uses the second stage of translation.

**Note**

Programming VTCR_EL2.T0SZ to a value smaller than the effective minimum value shown in Table D4-6 can never provide support for a larger address range than the range given by the effective minimum value, because the stage 1 output address will give an Address size fault if it is larger than either:

- The PA size, for a VMSAv8-64 stage 1 translation.
- 40 bits, for a VMSAv8-32 stage 1 translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Atomicity of register changes on changing virtual machine

From the viewpoint of software executing at Non-secure EL1 or EL0, when there is a switch from one virtual machine to another, the registers that control or affect address translation must be changed atomically. This applies to the registers for the Non-secure EL1&0 translation regime. This means that all of the following registers must change atomically:

- The registers associated with the stage 1 translations:
  - MAIR_EL1 and AMAIR_EL1.
  - TTBR0_EL1, TTBR1_EL1, TCR_EL1, and CONTEXTIDR_EL1.
  - SCTLR_EL1.
- The registers associated with the stage 2 translations:
  - VTTBR_EL2 and VTCR_EL2.
  - MAIR_EL2 and AMAIR_EL2.
  - SCTLR_EL2.

**Note**

Only some bits of SCTLR_EL1 affect the stage 1 translation, and only some bits of SCTLR_EL2 affect the stage 2 translation. However, in each case, changing these bits requires a write to the register, and that write must be atomic with the other register updates.

These registers apply to execution using the Non-secure EL1&0 translation regime. However, when updated as part of a switch of virtual machines they are updated by software executing at EL2. This means the registers are *out of context* when they are updated, and no synchronization precautions are required.

### Use of out-of-context translation regimes

The architecture requires that:

- When executing at EL3, EL2, or Secure EL1, the PE must not use the registers associated with the Non-secure EL1&0 translation regime for speculative memory accesses.
- When executing at EL3 or Secure EL1, the PE must not use the registers associated with the EL2 translation regime for speculative memory accesses.
- When executing at EL3, EL2, or Non-secure EL1, the PE must not use the registers associated with the Secure EL1 translation regime for speculative memory accesses.

When entering an Exception level, on completion of a `DSB` instruction, no new memory accesses using any translation table entries from a translation regime of an Exception level lower than the Exception level that has been entered will be observed by any observers, to the extent that those accesses are required to be observed as determined by the shareability and cacheability of those translation table entries.

**Note**

- This does not require that speculative memory accesses cannot be performed using those entries if it is impossible to tell that those memory accesses have been observed by the observers.
- This requirement does not imply that, on taking an exception to a higher Exception level, any translation table walks started before the exception was taken will be completed by the time the higher Exception level is entered, and therefore memory accesses required for such a translation table walk might, in effect, be performed speculatively. However, the execution of a `DSB` on entry to the higher Exception level ensures that these accesses are complete.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D4.2.3 Memory translation granule size

The memory translation granule size defines both:

- The maximum size of a single translation table.
- The memory *page* size. That is, the granularity of a translation table lookup.

VMSAv8-64 supports translation granule sizes of 4KB, 16KB, and 64KB. Support for each granule size is optional, and is indicated as shown in Table D4-7:

**Table D4-7 Identifying supported granule sizes**

| Granule size | Support indicated by: Field | Values | |
|---|---|---|---|
| 4KB | ID_AA64MMFR0_EL1.TGRAN4 | **0b0000** | 4KB granule size supported. |
| | | **0b1111** | 4KB granule size not supported. |
| 16KB | ID_AA64MMFR0_EL1.TGRAN16 | **0b0000** | 16KB granule size not supported. |
| | | **0b0001** | 16KB granule size supported. |
| 64KB | ID_AA64MMFR0_EL1.TGRAN64 | **0b0000** | 64KB granule size supported. |
| | | **0b1111** | 64KB granule size not supported. |

In VMSAv8-64, each address translation stage is configured, independently, to use one of the supported granule sizes.

**Note**

- Using a larger granule size can reduce the maximum required number of levels of address lookup because:
  - The increased translation table size means the translation table holds more entries. This means a single lookup can resolve more bits of the input address.
  - The increased page size means more of the least-significant address bits are required to address a page. These address bits are flat mapped from the input address to the output address, and therefore do not require translation.
- ARM recommends that memory-mapped peripherals are separated by an integer multiple of the largest granule size supported by the operating system or hypervisor, to allow each peripheral to be managed independently.

Table D4-8 summarizes the effects of the different granule sizes.

**Table D4-8 Effect of granule size on a stage of address translation**

| Property | 4KB granule | 16KB granule | 64KB granule | Notes |
|---|---|---|---|---|
| Maximum number of entries in a translation table | 512 | 2048 (2K) | 8192 (8K) | - |
| Address bits resolved in one level of lookup | 9 | 11 | 13 | $2^9$=512, $2^{11}$=2K, $2^{13}$=8K |
| Page size | 4KB | 16KB | 64KB | - |
| Page address range | VA[11:0]= PA[11:0] | VA[13:0]= PA[13:0] | VA[15:0]= PA[15:0] | $2^{12}$=4K, $2^{14}$=16K, $2^{16}$=64K |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### How the granule size affects the address translation process

As Table D4-8 on page D4-1755 shows, the translation granule determines the number of address bits:

- Required to address a memory page.
- That can be resolved in a single translation table lookup.

This means the translation granule determines how the *input address* (IA) is resolved to an *output address* (OA) by the translation process.

Because a single translation table lookup can resolve only a limited number of address bits, the IA to OA resolution requires multiple *levels* of lookup.

Considering the resolution of the maximum IA range of 48 bits, with a translation granule size of $2^n$ bytes:

- The least-significant $n$ bits of the IA address the memory page. This means OA[($n$-1):0]=IA[($n$-1):0].
- The remaining (48-$n$) bits of the IA, IA[47:$n$], must be resolved by the address translation.
- A translation table descriptor is 8 bytes. Therefore:
  - A complete translation table holds $2^{(n-3)}$ descriptors.
  - A single level of translation can resolve a maximum of ($n$-3) bits of address.

  Consider the translation process, working back from the final level of lookup, that resolves the least significant of the address bits that require translation. Because a level of lookup can resolve ($n$-3) bits of address:
  - The final level of lookup resolves IA[(2$n$-4):$n$].
  - The previous level of lookup resolves IA[(3$n$-7):(2$n$-3)].

  However, the level of lookup that resolves the most significant bits of the IA might not require a full-sized translation table. Therefore, in general, the address bits resolved in a level of lookup are:

  IA[Min(47, (($x$-3)($n$-3)+2$n$-4)):($n$+($x$-3)($n$-3))], where:

  **Min($a$, $b$)** Is a function that returns the minimum of $a$ and $b$.

  ***x*** Indicates the level of lookup. This is defined so that the level that resolves the least significant bit of the translated IA bits is level 3.

The following diagrams show this model, for each of the permitted granule sizes.

Figure D4-4 shows how a 48-bit IA is resolved when using the 4KB translation granule.

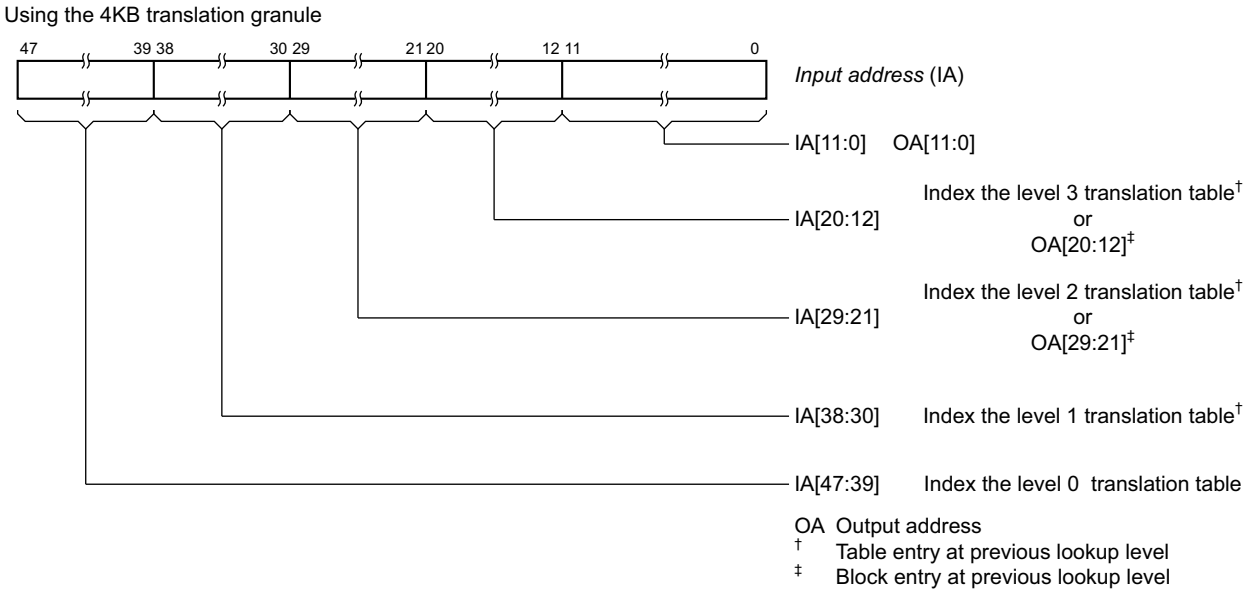



**Figure D4-4 How the IA is resolved when using the 4KB translation granule**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Figure D4-5 shows how a 48-bit IA is resolved when using the 16KB translation granule.

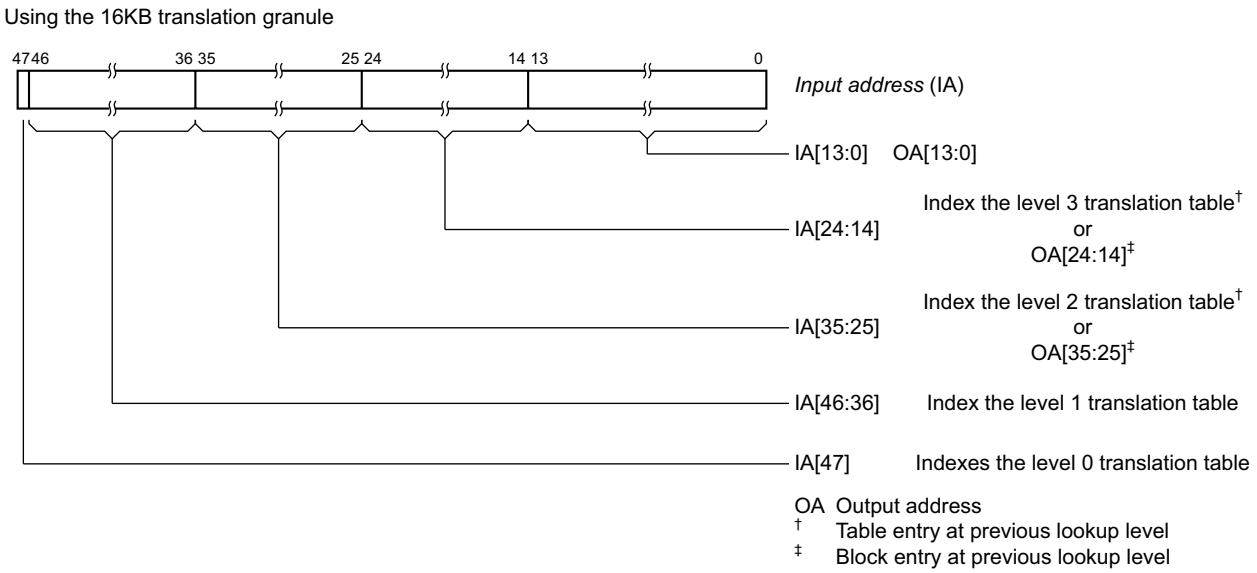



**Figure D4-5 How the IA is resolved when using the 16KB translation granule**

Figure D4-6 shows how a 48-bit IA is resolved when using the 64KB translation granule.

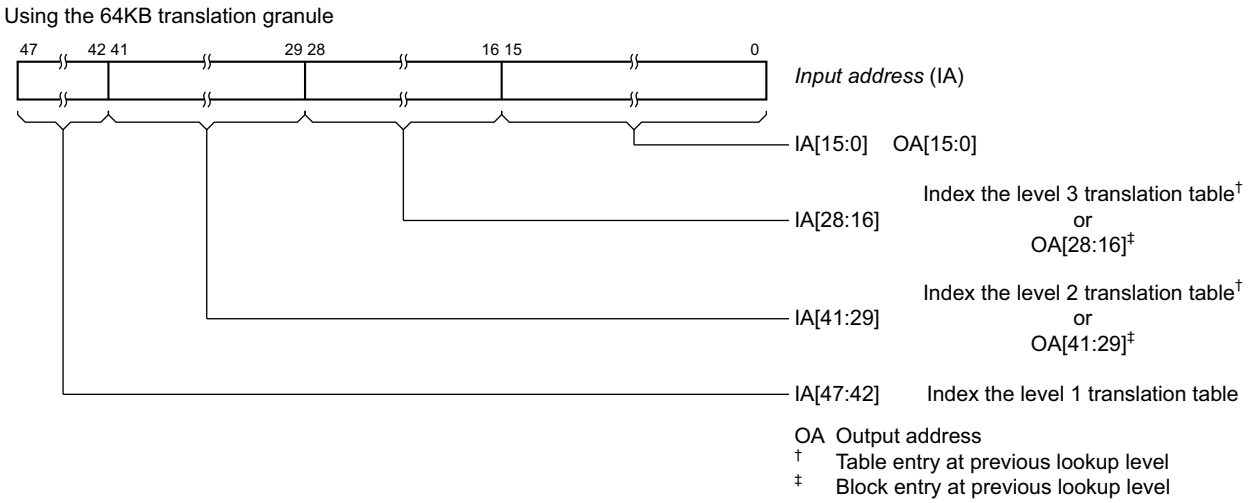



**Figure D4-6 How the IA is resolved when using the 64KB translation granule**

Later sections of this chapter give more information about the translation process, and explain the terminology used in these figures.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Effect of granule size on translation table addressing and indexing

Table D4-9 shows the effect of the translation granule size on the addressing and indexing of the TTBR, and on the input address range that must be resolved.

**Table D4-9 The effect of translation granule size on the translation tables**

| Granule size | Translation table | | Translation resolves[a] | Notes |
|---|---|---|---|---|
| | Addressed by | Indexed by[b] | | |
| 4KB | TTBR[47:12] | IA[($x$ + 8):$x$] | IA[47:12] | One level of lookup resolves up to[c] 9 bits of IA |
| 16KB | TTBR[47:14] | IA[($x$ + 10):$x$] | IA[47:14] | One level of lookup resolves up to[c] 11 bits of IA |
| 64KB | TTBR[47:16] | IA[($x$ + 12):$x$] | IA[47:16] | One level of lookup resolves up to[c] 13 bits of IA |

a. When translating a maximum-sized input address of 48 bits, and accessing a page of memory.
b. Where the value of $x$ depends on the lookup level, see Table D4-10.
c. Depending on the IA size, the initial lookup might resolve fewer bits of the IA.

Table D4-10 shows the IA bits resolved at each level of lookup, and how these correspond to the possible values of $x$ in Table D4-9.

**Table D4-10 IA bits resolved at different levels of lookup**

| Lookup level | 4KB granule size | 16KB granule size | 64KB granule size |
|---|---|---|---|
| Zero | IA[47:39], $x$ = 39 | IA[47[a]], $x$ = 47 | -[b] |
| First | IA[38:30], $x$ = 30 | IA[46:36], $x$ = 36 | IA[47[a]:42], $x$ = 42 |
| Second | IA[29:21], $x$ = 21 | IA[35:25], $x$ = 25 | IA[41:29], $x$ = 29 |
| Third | IA[20:12], $x$ = 12 | IA[24:14], $x$ = 14 | IA[28:16], $x$ = 16 |

a. Smaller value than indicated in Table D4-9, as explained in this section.
b. Level 0 lookup not possible with 64KB granule size

Table D4-9 refers to accessing a complete translation table, of 4KB, 16KB, or 64KB. However, the ARMv8 translation system supports the following possible variations from the information in Table D4-9:

**Reduced IA width**

Depending on the configuration and implementation choices, the required input address width for the initial level of lookup might be smaller than the number of address bits that can be resolved at that level. This means that, for this initial level of lookup:

- The translation table size is reduced. For each 1 bit reduction in the input address size the size of the translation table is halved.

  **Note**

  — This has no effect on the translation table size for subsequent levels of lookup, for which the lookups always use full-sized translation tables.

  — For a stage 2 translation, it might be possible to start the translation at a lower level, see *Concatenated translation tables* on page D4-1759.

- More low-order TTBR bits are needed to hold the translation table base address.

Example D4-1 on page D4-1759 shows how this applies to translating a 35-bit input address range using the 4KB granule.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Example D4-1 Effect of an IA width of 35 bits when using the 4KB granule size**

With a 4KB granule size, a single level of lookup can resolve up to 9 bits of IA. If an implementation has a 35-bit input address range, IA[34:0], Table D4-10 on page D4-1758 shows that lookup must start at level 1, and that the initial lookup must resolve IA[34:30], meaning it resolves 5 bits of address: This 4-bit reduction in the required resolution means:

- The translation table size is divided by $2^4$, giving a size of 256B.
- The TTBR requires 4 more bits for the translation table base address, which becomes TTBR[47:8].

When using the 64KB translation granule to translate the maximum IA size of 48 bits, Table D4-10 on page D4-1758 shows that a level 1 lookup must resolve only IA[47:42]. This is 6 bits of address, compared to the 13 bits that can be resolved at a single level of lookup. This 7-bit reduction in the required resolution means:

- The translation table size is divided by $2^7$, giving a size of 512B.
- The TTBR requires 7 more bits for the translation table base address, which becomes TTBR[47:9].

**Concatenated translation tables**

For stage 2 address translations, for the initial lookup, up to 16 translation tables can be concatenated. This means additional IA bits can be resolved at that lookup level. Each additional IA bit resolved:

- Doubles the number of translation tables required. Resolving an additional *n* bits requires $2^n$ concatenated translation tables at the initial lookup level.
- Reduces by 1 bit the width of the translation table base address held in the TTBR.

This means that, for the initial lookup of a stage 2 translation table, the IA ranges shown in Table D4-10 on page D4-1758 can be extended by up to 4 bits. Example D4-2 shows how concatenation can be used to resolve a 40-bit IA when using the 4KB translation granule.

**Example D4-2 Concatenating translation tables to resolve a 40-bit IA range, with the 4K granule**

Table D4-10 on page D4-1758 shows that, when using the 4KB translation granule, a level 1 lookup can resolve a 39-bit IA, with the first lookup resolving IA[38:30]. For a stage 2 translation, to extend the IA width to 40 bits and resolve IA[39:30] with the first lookup:

- Two translation tables are concatenated, giving a total size of 8KB.
- The TTBR requires 1 fewer bit for the translation table base address, which becomes TTBR[47:13].

For more information, see *Concatenated translation tables for the initial stage 2 lookup* on page D4-1775.

In all cases, the translation table, or block of concatenated translation tables, must be aligned to the actual size of the table or block of concatenated tables.

The translation table base address held in the TTBR is defined in the OA map for that stage of address translation. The information given in this section assumes this stage of translation has an OA size of 48 bits, meaning the translation table base address is:

- TTBR[47:12] if using the 4KB translation granule.
- TTBR[47:14] if using the 16KB translation granule.
- TTBR[47:16] if using the 64KB translation granule.

If the OA address is smaller than 48 bits then the upper bits of this field must be written as zero. For example, for a 40-bit OA range:

- If using the 4KB translation granule:
  - TTBR[47:40] must be set to zero.
  - TTBR[39:12] holds the translation table base address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If using the 16KB translation granule:
  - TTBR[47:40] must be set to zero.
  - TTBR[39:14] holds the translation table base address.
- If using the 64KB translation granule:
  - TTBR[47:40] must be set to zero.
  - TTBR[39:16] holds the translation table base address.

In all cases, if TTBR[47:40] is not zero, any attempt to access the translation table generates an Address size fault.

### D4.2.4 Translation tables and the translation process

The following subsections describe general properties of the translation tables and translation table walks, that are largely independent of the translation table format:

- *Translation table walks*.
- *Security state of translation table lookups* on page D4-1762.
- *Control of translation table walks* on page D4-1762.
- *Security state of translation table lookups* on page D4-1762.

See also *Selection between TTBR0 and TTBR1* on page D4-1774.

#### Translation table walks

A *translation table walk* comprises one or more *translation table lookups*. The translation table walk is the set of lookups that are required to translate the virtual address to the physical address. For the Non-secure EL1&0 translation regime, this set includes lookups for both the stage 1 translation and the stage 2 translation. The information returned by a successful translation table walk is:

- The required physical address. If the access is from Secure state this includes identifying whether the access is to the Secure physical address space or the Non-secure physical address space, see *Security state of translation table lookups* on page D4-1762.
- The memory attributes for the target memory region, as described in *Memory types and attributes* on page B2-93. For more information about how the translation table descriptors specify these attributes see *Memory region attributes* on page D4-1816.
- The access permissions for the target memory regions. For more information about how the translation table descriptors specify these permissions see *Memory access control* on page D4-1808.

The translation table walk starts with a read of the translation table for the initial lookup. The TTBR for the stage of translation holds the base address of this table. Each translation table lookup returns a descriptor, that indicates one of the following:

- The entry is the final entry of the walk. In this case, the entry contains the OA, and the permissions and attributes for the access.
- An additional level of lookup is required. In this case, the entry contains the translation table base address for that lookup. In addition:
  - The descriptor provides hierarchical attributes that are applied to the final translation, see *Hierarchical control of Secure or Non-secure memory accesses* on page D4-1807 and *Hierarchical control of data access permissions* on page D4-1810.
  - If the translation is in a Secure translation regime, the descriptor indicates whether that base address is in the Secure or Non-secure address space, unless a hierarchical control at a previous level of lookup has indicated that it must be in the Non-secure address space.
- The descriptor is invalid. In this case, the memory access generates a Translation fault.

Figure D4-7 on page D4-1761 gives a generalized view of a single stage of address translation, where three levels of lookup are required.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

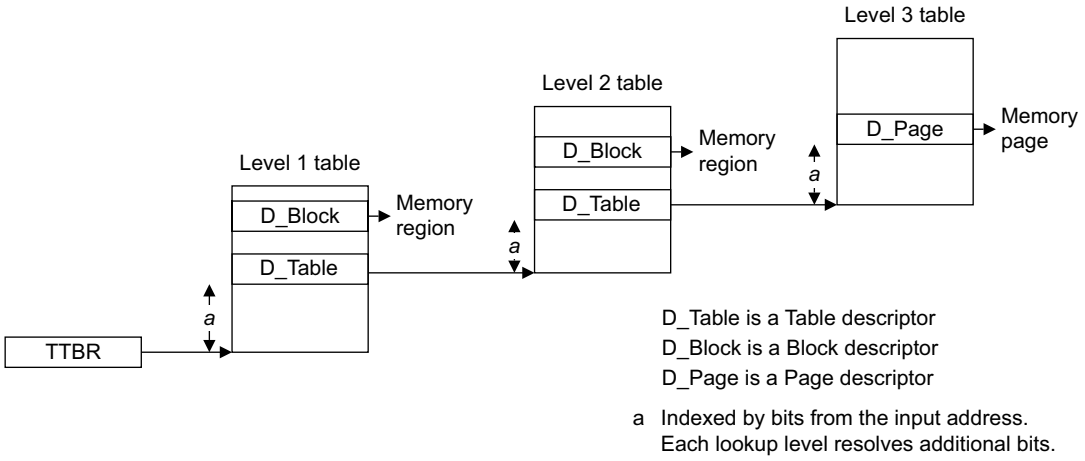



**Figure D4-7 Generalized view of a stage of address translation**

A translation table lookup from VMSAv8-64 performs a single-copy atomic 64-bit access to the translation table entry. This means the translation table entry is treated as a 64-bit object for the purpose of endianness. SCTLR.EE determines the endianness of the translation table lookups.

**Note**

**Dynamically changing translation table endianness**

Because any change to an SCTLR.EE, bit requires synchronization before it is visible to subsequent operations, ARM strongly recommends that any EE bit is changed only when either:

- Executing at an Exception level that does not use the translation tables affected by the EE bit being changed.
- Executing with address translation disabled for any stage of translation affected by the EE bit being changed.

Address translation stages are disabled by setting an SCTLR.M bit to 0. See the appropriate register description for more information.

The appropriate TTBR holds the output address of the base of the translation table used for the initial lookup, and:

- For all address translation stages other than Non-secure EL1&0 stage 1 translations, the output address held in the TTBR, and any translation table base address returned by a translation table descriptor, is the PA of the base of the translation table.
- For Non-secure EL1&0 stage 1 translations, the output address held in the TTBR, and any translation table base address returned by a translation table descriptor, is the IPA of the base of the translation table. This means that if stage 2 address translation is enabled, each of these OAs is subject to second stage translation.

  **Note**

  TLB caching can be used to minimise the number of translation table lookups that must be performed. Because each stage 1 OA generated during a translation table walk is subject to a stage 2 translation, if the caching of translation table entries is ineffective, a VA to PA address translation with two stages of translation can give rise to multiple translation table lookups. The number of lookups required is given by the following equation:

  $(S1+1)*(S2+1) - 1$

  Where, for the Non-secure EL1&0 translation regime, S1 is the number of levels of lookup required for a stage 1 translation, and S2 is the number of levels of lookup required for a stage 2 translation.

The TTBR also determines the memory cacheability and shareability attributes that apply, for that stage of translation, to all translation table lookups generated by that stage of translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Normal memory type is the memory type defined for a translation table lookup for a stage of translation.

**Note**

- In a two stage translation system, a translation table lookup from stage 1, that has the Normal memory type defined at stage 1 by this rule, can still be given the Device memory type as part of the stage 2 translation of that address. ARM strongly recommends against such a remapping of the memory type, and the architecture includes a trap of this behavior to EL2. For more information, see *Stage 2 fault on a stage 1 translation table walk* on page D4-1830.
- The rules about mismatched attributes given in *Mismatched memory attributes* on page B2-104 apply to the relationship between translation table walks and explicit memory accesses to the translation tables in the same way that they apply to the relationship between different explicit memory accesses to the same location. For this reason, ARM strongly recommends that the attributes that the TTBR applies to the translation tables are the same as the attributes that are applied for explicit accesses to the memory that holds the translation tables.

For more information see *Overview of the VMSAv8-64 address translation stages*.

See also *Selection between TTBR0 and TTBR1* on page D4-1774.

### Security state of translation table lookups

For a Non-secure translation regime, all translation table lookups are performed to Non-secure output addresses.

For a Secure translation regimes, the initial translation table lookup is performed to a Secure output address.

If the translation table descriptor returned as a result of that initial lookup points to a second translation table, then the NSTable bit in that descriptor determines whether that translation table lookup is made to Secure or to Non-secure output addresses.

This applies for all subsequent translation table lookups as part of that translation table walk, with the additional rule that any translation table descriptor that is returned from Non-secure memory is treated as if the NSTable bit in that descriptor indicates that the subsequent translation table lookup is to Non-secure memory.

### Control of translation table walks

For the first stage of the EL1&0 translation regime, the TCR_EL1.{EPD0, EPD1} bits determine whether the translation tables for that regime are valid. EPD0 indicates whether the table that TTBR0_EL1 points to is valid, and EPD1 indicates whether the table that TTBR1_EL1 points to is valid. The effect of these bits is:

**EPD*n* == 0** The translation table is valid, and can be used for a translation table lookup.

**EPD*n* == 1** If a TLB miss occurs based on TTBR*n*, a Translation fault is returned, and no translation table walk is performed. The fault is reported as a level 0 fault.

## D4.2.5 Overview of the VMSAv8-64 address translation stages

As shown in *Memory translation granule size* on page D4-1755, the granule size determines significant aspects of the address translation process. *Effect of granule size on translation table addressing and indexing* on page D4-1758 shows, for each granule size:

- How the required input address range determines the required initial lookup levels.
- For stage 2 translations, the possible effect described in *Concatenated translation tables* on page D4-1759.
- The TTBR addressing and indexing for the initial lookup.

The following subsections summarize the multiple levels of lookup that can be required for a single stage of address translation that might require the maximum number of lookups:

- *Overview of VMSAv8-64 address translation using the 4KB translation granule* on page D4-1763.
- *Overview of VMSAv8-64 address translation using the 16KB translation granule* on page D4-1766.
- *Overview of VMSAv8-64 address translation using the 64KB translation granule* on page D4-1769.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Overview of VMSAv8-64 address translation using the 4KB translation granule

The requirements for the level of the initial lookup are different for stage 1 and stage 2 translations.

### *Overview of stage 1 translations, 4KB granule*

For a stage 1 translation, the required initial lookup level is determined only by the required input address range specified by the corresponding TCR.TxSZ field. When using the 4KB translation granule, Table D4-11 shows this requirement.

**Table D4-11 TCR.T*n*SZ values and IA ranges when there is no concatenation of translation tables**

| Initial lookup level | T*n*SZ values for and input address ranges[a] for starting at this level | | | |
|---|---|---|---|---|
| | $T nSZ_{min}$ | $IA_{max}$ | $T nSZ_{max}$ | $IA_{min}$ |
| Zero | 16 | IA[47:12] | 24 | IA[39:12] |
| First | 25 | IA[38:12] | 33 | IA[30:12] |
| Second | 34 | IA[29:12] | 39 | IA[24:12] |

a. The IAs show the address bits to be resolved when addressing a page of memory, see the *Note* that follows.

These configuration options are also permitted for stage 2 translations.

**Note**

- When using the 4KB translation granule, the initial lookup cannot be at level 3.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA, When using the 4KB translation granule, IA[11:0] = OA[11:0] for all translations.

Figure D4-8 shows the stage 1 address translation, for an address translation using the 4KB granule with an input address size greater than 39 bits.

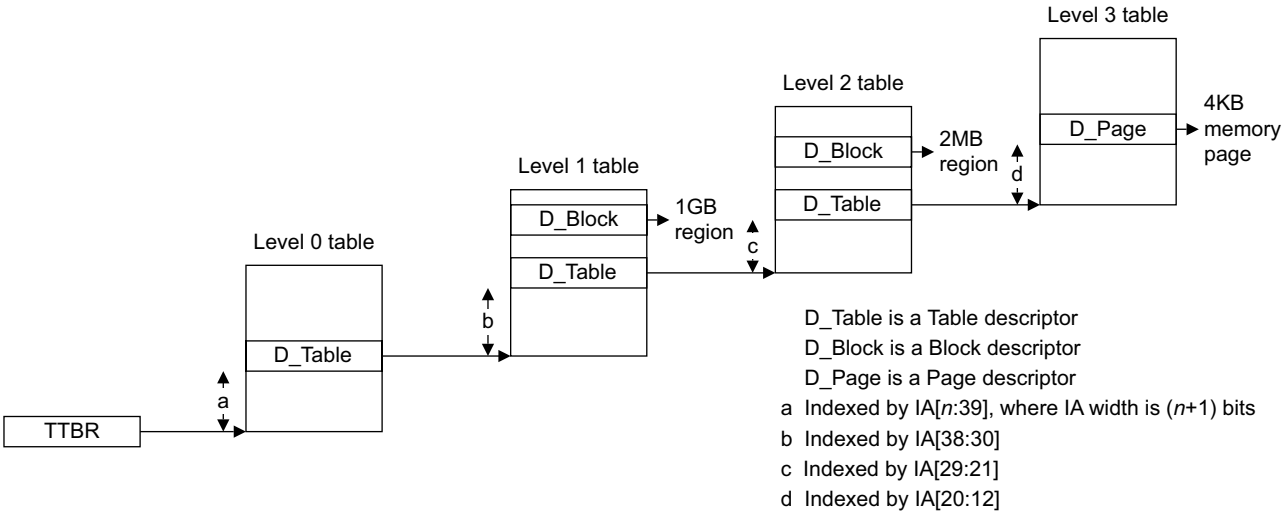



**Figure D4-8 General view of VMSAv8-64 stage 1 address translation, 4KB granule**

### *Overview of stage 2 translations, 4KB granule*

For a stage 2 translation, up to 16 translation tables can be concatenated at the initial lookup level. For certain input address sizes, concatenating tables in this way means that the lookup starts at a lower level than would otherwise be the case. For more information see *Concatenated translation tables for the initial stage 2 lookup* on page D4-1775.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When using the 4KB translation granule, Table D4-12 shows all possibilities for the initial lookup for a stage 2 translation.

**Table D4-12 VTCR_EL2.T0SZ values and IA ranges, including cases where translation tables are concatenated**

| Tables[a] | 1 | | 2 | | 4 | | 8 | | 16 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Initial lookup level | T0SZ values and input address ranges[b] for starting at this level | | | | | | | | | |
| | T0SZ | IA | T0SZ | IA | T0SZ | IA | T0SZ | IA | T0SZ | IA |
| Zero | 16-24 | IA[47:12]-IA[39:12] | - | - | - | - | - | - | - | - |
| First | 25-33 | IA[38:12]-IA[30:12] | 24 | IA[39:12] | 23 | IA[40:12] | 22 | IA[41:12] | 21 | IA[42:12] |
| Second | 34-39 | IA[29:12]-IA[24:12] | 33 | IA[30:12] | 32 | IA[31:12] | 31 | IA[32:12] | 30 | IA[33:12] |

a. Number of concatenated translation tables at the initial lookup level. *1 table* corresponds to no concatenation, see Table D4-11 on page D4-1763.

b. The IAs shown in the table indicate the address bits to be resolved by an address translation addressing a page of memory, see the *Note* that follows.

**Note**

- When using the 4KB translation granule, the initial lookup cannot be at level 3.
- Because concatenating translation tables reduces the number of levels of lookup required, when using the 4KB translation granule, tables cannot be concatenated at level 0.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA. When using the 4KB translation granule, IA[11:0] = OA[11:0] for all translations.

In addition, VTCR_EL2.SL0 indicates the required initial lookup level, as Table D4-13 shows.

**Table D4-13 VTCR_EL2.SL0 values, 4KB granule**

| Initial lookup level | VTCR_EL2.SL0 |
|---|---|
| Zero | 0b10 |
| One | 0b01 |
| Two | 0b00 |

Because the maximum number of concatenated translation tables is 16, there is a relationship between the permitted VTCR_EL2.{T0SZ, SL0} values. If, when a translation table walk is started, the T0SZ value is not consistent with the SL0 value, a stage 2 level 0 Translation fault is generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Figure D4-9 shows the stage 2 address translation, for an input address size of between 40 and 43 bits. For an input address size in this range, the lookup can start at either level 0 or level 1.

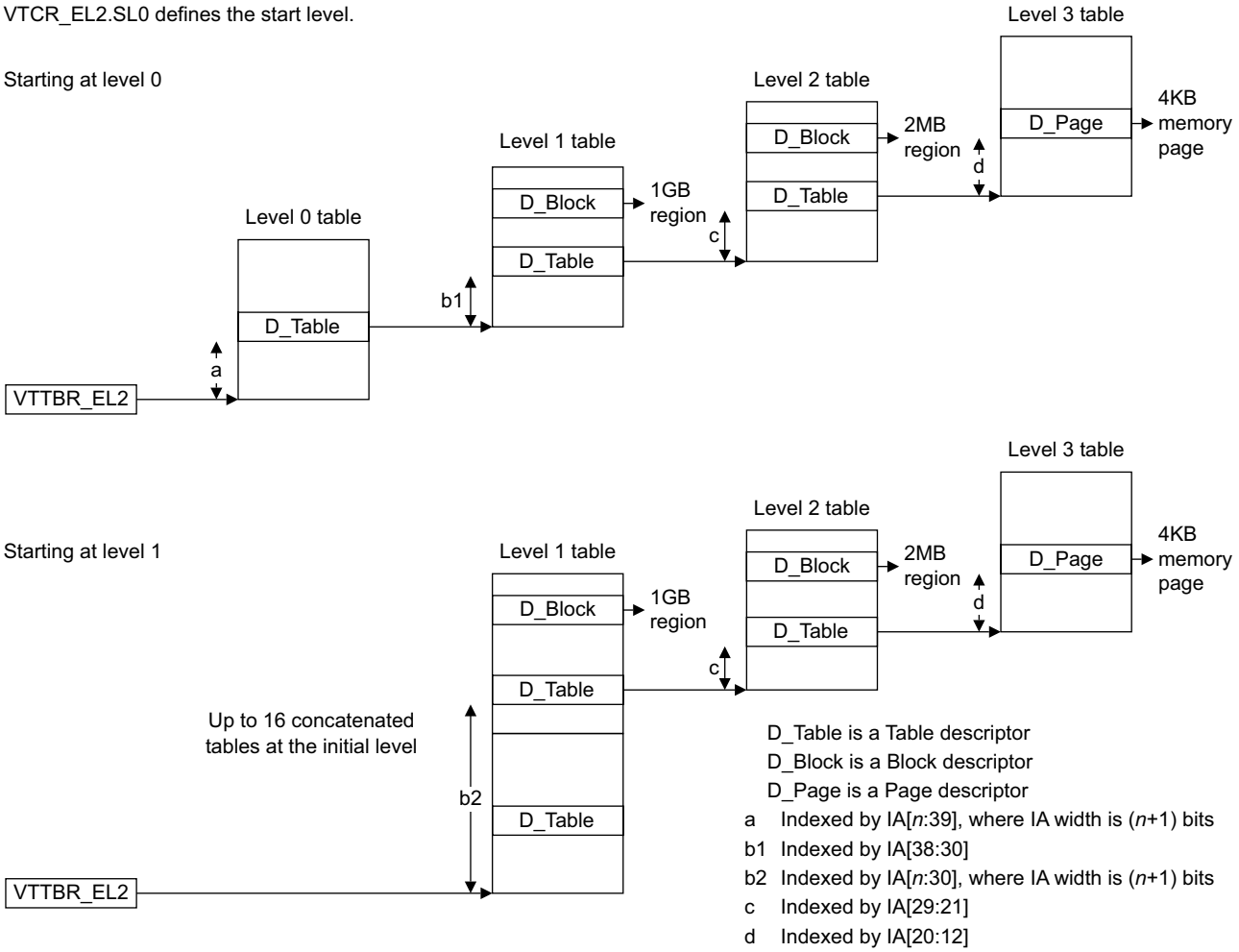



**Figure D4-9 General view of VMSAv8-64 stage 2 address translation, 4KB granule**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Overview of VMSAv8-64 address translation using the 16KB translation granule

The requirements for the level of the initial lookup are different for stage 1 and stage 2 translations.

### *Overview of stage 1 translations, 16KB granule*

For a stage 1 translation, the required initial lookup level is determined only by the required input address range specified by the corresponding TCR.TxSZ field. When using the 16KB translation granule, Table D4-14 shows this requirement.

**Table D4-14 TCR.T*n*SZ values and IA ranges when there is no concatenation of translation tables**

| Initial lookup level | T*n*SZ values for and input address ranges[a] for starting at this level | | | |
|---|---|---|---|---|
| | $T nSZ_{min}$ | $IA_{max}$ | $T nSZ_{max}$ | $IA_{min}$ |
| Zero | 16 | IA[47:14] | - | - |
| First | 17 | IA[46:14] | 27 | IA[36:14] |
| Second | 28 | IA[35:14] | 38 | IA[25:14] |
| Third | 39 | IA[24:14] | - | - |

a. The IAs show the address bits to be resolved when addressing a page of memory, see the *Note* that follows.

The configuration options for an initial lookup at level 1, level 2, or level 3 are also permitted for stage 2 translations, but stage 2 translation does not permit an initial lookup at level 0.

**Note**

- When using the 16KB translation granule, a maximum of 1 bit of IA is resolved by a level 0 lookup.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA, When using the 16KB translation granule, IA[13:0] = OA[13:0] for all translations.

Figure D4-10 shows the stage 1 address translation, for an address translation using the 16KB granule with an input address size of 48 bits.

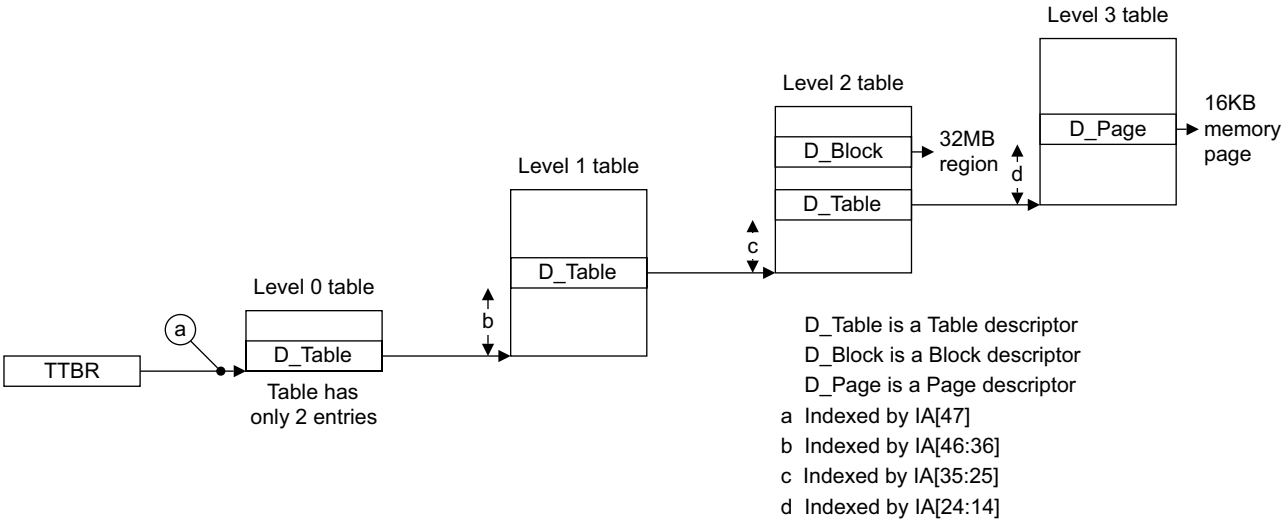



**Figure D4-10 General view of VMSAv8-64 stage 1 address translation, 16KB granule**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Overview of stage 2 translations, 16KB granule*

For a stage 2 translation, up to 16 translation tables can be concatenated at the initial lookup level. For certain input address sizes, concatenating tables in this way means that the lookup starts at a lower level than would otherwise be the case. For more information see *Concatenated translation tables for the initial stage 2 lookup* on page D4-1775.

When using the 16KB granule, for a stage 2 translation with an input address sized of 48 bits, the initial lookup must be at level 1, with two concatenated translation tables at this level.

When using the 16KB translation granule, Table D4-15 shows all possibilities for the initial lookup for a stage 2 translation.

**Table D4-15 VTCR_EL2.T0SZ values and IA ranges, including cases where translation tables are concatenated**

| Tables[a] | 1 | | 2 | | 4 | | 8 | | 16 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Initial lookup level | T0SZ values and input address ranges[b] for starting at this level | | | | | | | | | |
| | T0SZ | IA | T0SZ | IA | T0SZ | IA | T0SZ | IA | T0SZ | IA |
| One | 17-27 | IA[46:14]-IA[36:14] | 16 | IA[47:14] | - | - | - | - | - | - |
| Two | 28-38 | IA[35:14]-IA[25:14] | 27 | IA[36:14] | 26 | IA[37:14] | 25 | IA[38:14] | 24 | IA[39:14] |
| Three | 39 | IA[24:14] | 38 | IA[25:14] | 37 | IA[26:14] | 36 | IA[27:14] | 35 | IA[28:14] |

a. Number of concatenated translation tables at the initial lookup level. *1 table* corresponds to no concatenation, see Table D4-11 on page D4-1763.

b. The IAs shown in the table indicate the address bits to be resolved by an address translation addressing a page of memory, see the *Note* that follows.

**Note**

- When using the 16KB translation granule for a stage 2 translation, the initial lookup cannot be at level 0. When a 48-bit input address is required, translation must start with a level 1 lookup using two concatenated translation tables.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA. When using the 16KB translation granule, IA[13:0] = OA[13:0] for all translations.

In addition, VTCR_EL2.SL0 indicates the required initial lookup level, as Table D4-16 shows.

**Table D4-16 VTCR_EL2.SL0 values, 16KB granule**

| Initial lookup level | VTCR_EL2.SL0 |
|---|---|
| One | `0b10` |
| Two | `0b01` |
| Three | `0b00` |

Because the maximum number of concatenated translation tables is 16, there is a relationship between the permitted VTCR_EL2.{T0SZ, SL0} values. If, when a translation table walk is started, the T0SZ value is not consistent with the SL0 value, a stage 2 level 0 Translation fault is generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When stage 2 translation supports a 48-bit input address range, translation must start with a level 1 lookup using two concatenated translation tables. Figure D4-11 shows the translation for this case.

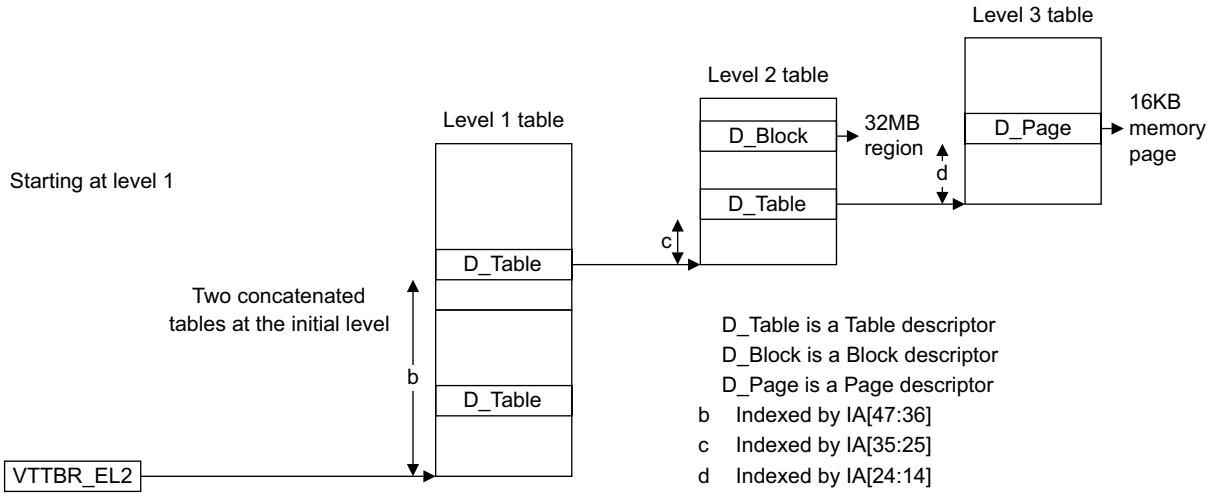



**Figure D4-11 VMSAv8-64 stage 2 address translation, 16KB granule, 48 bit input address**

However, for an input address size of between 37 and 40 bits, Table D4-15 on page D4-1767 shows that translation can start with either a level 1 lookup or a level 2 lookup, and Figure D4-12 shows these options.

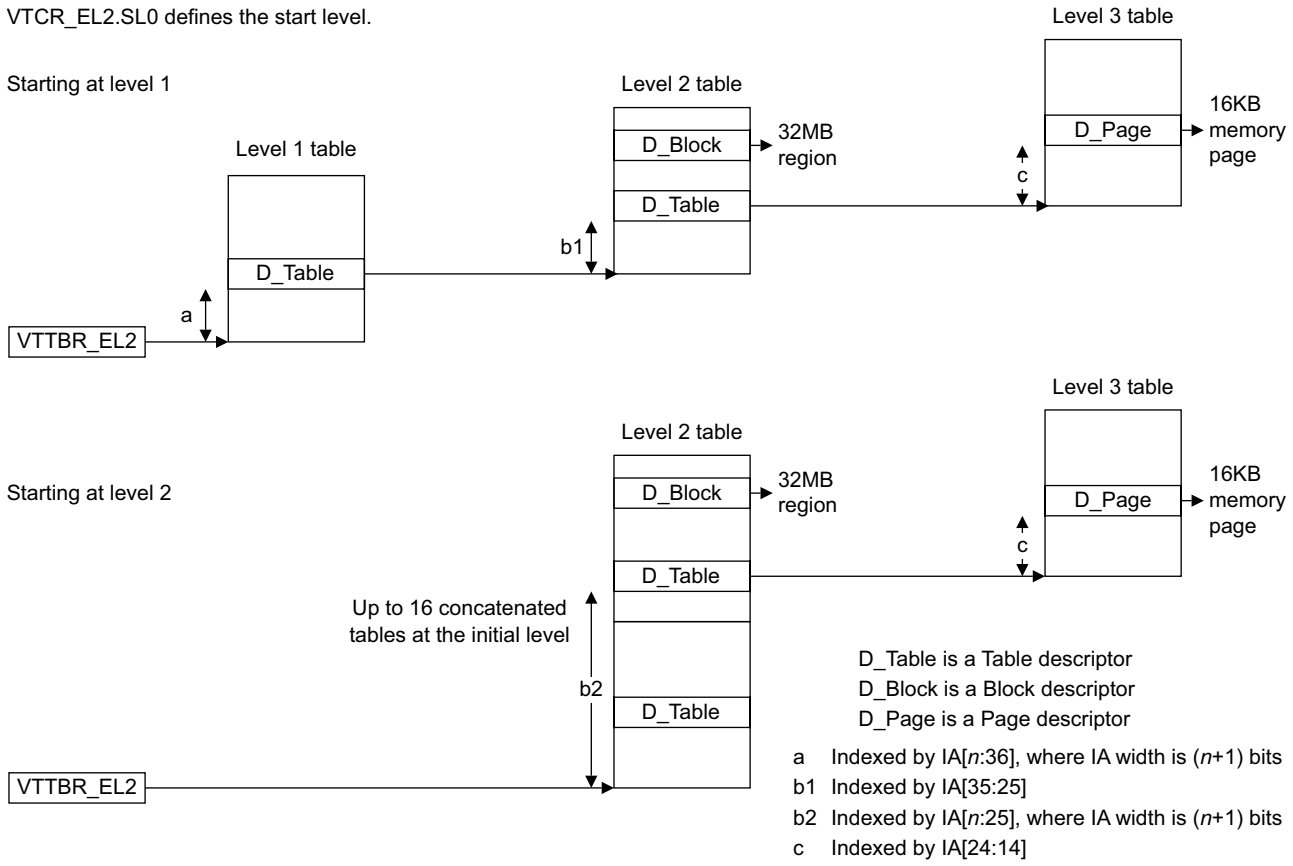



**Figure D4-12 General view of VMSAv8-64 stage 2 address translation, 16KB granule**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Overview of VMSAv8-64 address translation using the 64KB translation granule

The requirements for the level of the initial lookup are different for stage 1 and stage 2 translations.

### *Overview of stage 1 translations, 64KB granule*

For a stage 1 translation, the required initial lookup level is determined only by the required input address range specified by the corresponding TCR.TxSZ field. When using the 64KB translation granule, Table D4-17 shows this requirement.

**Table D4-17 TCR.T*n*SZ values and IA ranges when there is no concatenation of translation tables**

| Lookup level | T*n*SZ values for and input address ranges[a] for starting at this level | | | |
|---|---|---|---|---|
| | $T nSZ_{min}$ | $IA_{max}$ | $T nSZ_{max}$ | $IA_{min}$ |
| One | 16 | IA[47:16] | 21 | IA[42:16] |
| Two | 22 | IA[41:16] | 34 | IA[29:16] |
| Three | 35 | IA[28:16] | 39 | IA[24:16] |

a. The IAs show the address bits to be resolved when addressing a page of memory, see the *Note* that follows.

These configuration options are also permitted for stage 2 translations.

**Note**

- When using the 64KB translation granule, there are no level 0 lookups.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA. When using the 64KB translation granule, IA[15:0] = OA[15:0] for all translations.

Figure D4-13 shows the stage 1 address translation, for an address translation using the 64KB granule with a an input address size greater than 42 bits.

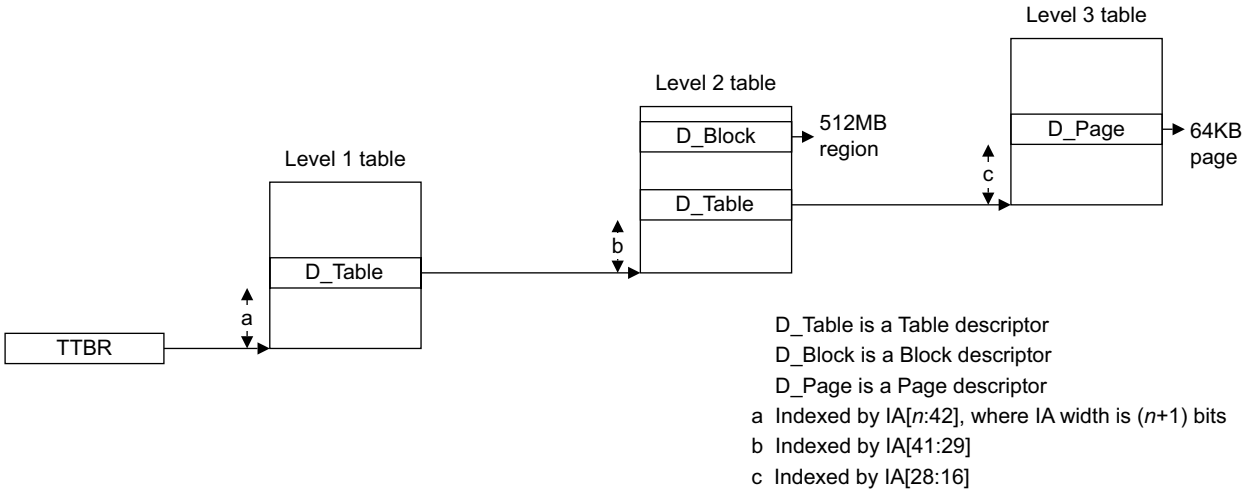



**Figure D4-13 General view of VMSAv8-64 stage 1 address translation, 64KB granule**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Overview of stage 2 translations, 64KB granule

For a stage 2 translation, up to 16 translation tables can be concatenated at the initial lookup level. For certain input address sizes, concatenating tables in this way means that the lookup starts at a lower level than would otherwise be the case. For more information see *Concatenated translation tables for the initial stage 2 lookup* on page D4-1775.

When using the 64KB translation granule, Table D4-18 shows all possibilities for the initial lookup for a stage 2 translation.

**Table D4-18 VTCR_EL2.T0SZ values and IA ranges when translation tables are concatenated**

| Tables[a] | 1 | | 2 | | 4 | | 8 | | 16 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Initial lookup level | T0SZ values and input address ranges[b] for starting at this level | | | | | | | | | |
| | T0SZ | IA | T0SZ | IA | T0SZ | IA | T0SZ | IA | T0SZ | IA |
| One | 16-21 | IA[47:16]-IA[42:16] | - | - | - | - | - | - | - | - |
| Two | 22-34 | IA[41:16]-IA[29:16] | 21 | IA[42:16] | 20 | IA[43:16] | 19 | IA[44:16] | 18 | IA[45:16] |
| Three | 35-39 | IA[28:16]-IA[24:16] | 34 | IA[29:16] | 33 | IA[30:16] | 32 | IA[31:16] | 31 | IA[32:16] |

a. Number of concatenated translation tables at the initial lookup level. *1 table* corresponds to no concatenation, see Table D4-17 on page D4-1769.

b. The IAs shown in the table indicate the address bits to be resolved by an address translation addressing a page of memory, see the *Note* that follows.

**Note**

- When using the 64KB translation granule, there are no level 0 lookups.
- Because concatenating translation tables reduces the number of levels of lookup required, when using the 64KB translation granule, tables cannot be concatenated at level 1.
- Some bits of the IA do not require resolution by the translation table lookup, because they always map directly to the OA. When using the 64KB translation granule, IA[15:0] = OA[15:0] for all translations.

VTCR_EL2.SL0 indicates the required initial lookup level, as Table D4-19 shows.

**Table D4-19 VTCR_EL2.SL0 values, 64K granule**

| Initial lookup level | VTCR_EL2.SL0 |
|---|---|
| One | 0b10 |
| Two | 0b01 |
| Three | 0b00 |

Because the maximum number of concatenated translation tables is 16, there is a relationship between the permitted VTCR_EL2.{T0SZ, SL0} values. If, when a translation table walk is started, the T0SZ value is not consistent with the SL0 value, a stage 2 level 0 Translation fault is generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Figure D4-14 shows the stage 2 address translation, for an input address size of between 43 and 46 bits. This means the lookup can start at either level 1 or level 2.

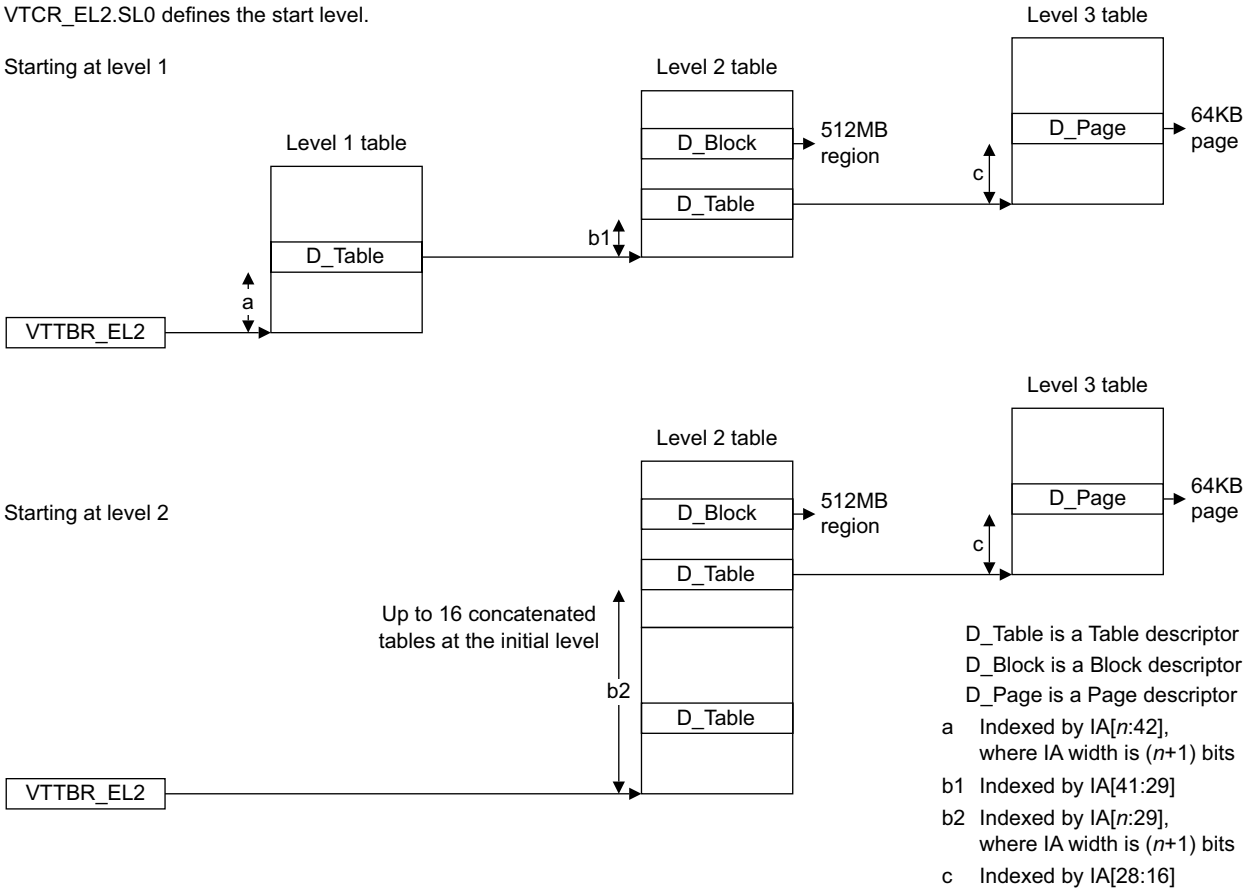



**Figure D4-14 General view of VMSAv8-64 stage 2 address translation, 64KB granule**

### D4.2.6 The VMSAv8-64 translation table format

This section provides the full description of the VMSAv8-64 translation table format, its use for address translations that are controlled by an Exception level using AArch64.

For the address translations that are controlled by an Exception level that is using AArch64:

- The TCR_EL1.{SH0, ORGN0, IRGN0, SH1, ORGN1, IRGN1} fields define memory region attributes for the translation table walk, for each of TTBR0_EL1 and TTBR1_EL1.
- For the Secure and Non-secure EL1&0 stage 1 translations, each of TTBR0_EL1 and TTBR1_EL1 contains an ASID field, and the TCR_EL1.A1 field selects which ASID to use.

For this translation table format, *Overview of the VMSAv8-64 address translation stages* on page D4-1762 summarizes the lookup levels, and *Descriptor encodings, ARMv8 level 0, level 1, and level 2 formats* on page D4-1800 describes the translation table entries.

The following subsections describe the use of this translation table format:

- *Translation granule size and associate block and page sizes* on page D4-1772.
- *Selection between TTBR0 and TTBR1* on page D4-1774.
- *Concatenated translation tables for the initial stage 2 lookup* on page D4-1775.
- *Possible translation table registers programming errors* on page D4-1777.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Translation granule size and associate block and page sizes

Table D4-20 shows the supported granule sizes, block sizes and page sizes, for the different granule sizes. For completeness, this table includes information for AArch32 state. In the table, the OA bit ranges are the OA bits that the translation table descriptor specifies to address the block or page of memory, in an implementation that supports a 48-bit OA range.

**Table D4-20 Translation table granule sizes, with block and page sizes, and output address ranges**

| Granule size | Table level | Block size and OA bit range | Page size and OA bit range |
|---|---|---|---|
| 4KB | Zero | - | - |
| | One | 1GB, OA[47:30] | - |
| | Two | 2MB, OA[47:21] | - |
| | Three | - | 4KB, OA[47:12] |
| 16KB | Zero | - | - |
| | One | - | - |
| | Two | 32MB, OA[47:25] | - |
| | Three | - | 16KB, OA[47:14] |
| 64KB | One | - | - |
| | Two | 512MB, OA[47:29] | - |
| | Three | - | 64KB, OA[47:16] |

Bit[1] of a translation table descriptor identifies whether the descriptor is a block descriptor, and:

- The 4KB granule size supports block descriptors only in level 1and level 2 translation tables.
- The 16KB and 64KB granule sizes support block descriptors only in level2 translation tables,

Setting bit[1] of a descriptor to 0 in a translation table that does not support block descriptors gives a Translation fault.

For translations managed from AArch64 state, the following tables expand the information for each granule size, showing for each lookup level and when accessing a single translation table:

- The maximum IA size, and the address bits that are resolved for that maximum size.
- The maximum OA range resolved by the translation table descriptors at this level, and the corresponding memory region size.
- The maximum size of the translation table. This is the size required for the maximum IA size.

Table D4-21 shows this information for the 4KB translation granule size, Table D4-22 on page D4-1773 shows this information for the 16KB translation granule size, and Table D4-23 on page D4-1773 shows this information for the 64KB translation granule size.

**Table D4-21 Properties of the address lookup levels, 4KB granule size**

| Level | Maximum input address | | Maximum output address | | Number of entries | Block entries supported? |
|---|---|---|---|---|---|---|
| | Size | Address range | Address range | Size of addressed region[a] | | |
| Zero | 256TB | Address[47:39] | Address[47:39] | 512GB | Up to 512 | No |
| One | 512GB | Address[38:30] | Address[47:30] | 1GB | Up to 512 | Yes |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D4-21 Properties of the address lookup levels, 4KB granule size (continued)**

| Level | Maximum input address | | Maximum output address | | Number of entries | Block entries supported? |
|---|---|---|---|---|---|---|
| | Size | Address range | Address range | Size of addressed region[a] | | |
| Two | 1GB | Address[29:21] | Address[47:21] | 2MB | Up to 512 | Yes |
| Three | 2MB | Address[20:12] | Address[47:12] | 4KB | 512 | Page only |

a. That is, the size of the region either addressed by descriptors at this level or to be resolved at this and the subsequent levels of lookup.

**Table D4-22 Properties of the address lookup levels, 16KB granule size**

| Level | Maximum input address | | Maximum output address | | Number of entries | Block entries supported? |
|---|---|---|---|---|---|---|
| | Size | Address range | Address range | Size of addressed region[a] | | |
| Zero | 256TB | Address[47] | Address[47] | 128TB | 2[b] | No |
| One | 128TB | Address[46:36] | Address[47:36] | 64GB | Up to 2048 | No |
| Two | 64GB | Address[35:25] | Address[47:25] | 32MB | Up to 2048 | Yes |
| Three | 32MB | Address[24:14] | Address[47:14] | 16KB | 2048 | Page only |

a. That is, the size of the region either addressed by descriptors at this level or to be resolved at this and the subsequent levels of lookup.
b. The translation table size is less than the maximum for this granule size, and therefore the number of entries is reduced.

**Table D4-23 Properties of the address lookup levels, 64KB granule size**

| Level | Maximum input address | | Maximum output address | | Number of entries | Block entries supported? |
|---|---|---|---|---|---|---|
| | Size | Address range | Address range | Size of addressed region[a] | | |
| One | 256TB | Address[47:42] | Address[47:42] | 4TB | Up to 64[b] | No |
| Two | 4TB | Address[41:29] | Address[47:29] | 512MB | Up to 8192 | Yes |
| Three | 512MB | Address[28:16] | Address[47:16] | 64KB | 8192 | Page only |

a. That is, the size of the region either addressed by descriptors at this level or to be resolved at this and the subsequent levels of lookup.
b. The translation table size is less than the maximum for this granule size, and therefore the number of entries is reduced.

For the initial lookup level:

- If the IA range specified by the TCR.T*x*SZ field is smaller than the maximum size shown in these table then this reduces the number of addresses in the table and therefore reduces the table size. The smaller translation table is aligned to its table size.
- For stage 2 translations, multiple translation tables can be concatenated to extend the maximum IA size beyond that shown in these tables. For more information see the stage 2 translation overviews in *Overview of the VMSAv8-64 address translation stages* on page D4-1762 and *Concatenated translation tables for the initial stage 2 lookup* on page D4-1775.

If a supplied input address is larger than the configured input address size, a Translation fault is generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Larger translation granule sizes typically requires fewer levels of translation tables to translate a particular size of virtual address.

For the TCR programming requirements for the initial lookup, see *Overview of the VMSAv8-64 address translation stages* on page D4-1762.

### Selection between TTBR0 and TTBR1

Every translation table walk starts by accessing the translation table addressed by the TTBR for the stage 1 translation for the required translation regime.

For the EL1&0 translation regime, the VA range is split into two subranges as shown in Figure D4-15, and:

- TTBR0_EL1 points to the initial translation table for the lower VA subrange, that starts at address `0x0000_0000_0000_0000`,
- TTBR1_EL1 points to the initial translation table for the upper VA subrange, that runs up to address `0xFFFF_FFFF_FFFF_FFFF`.

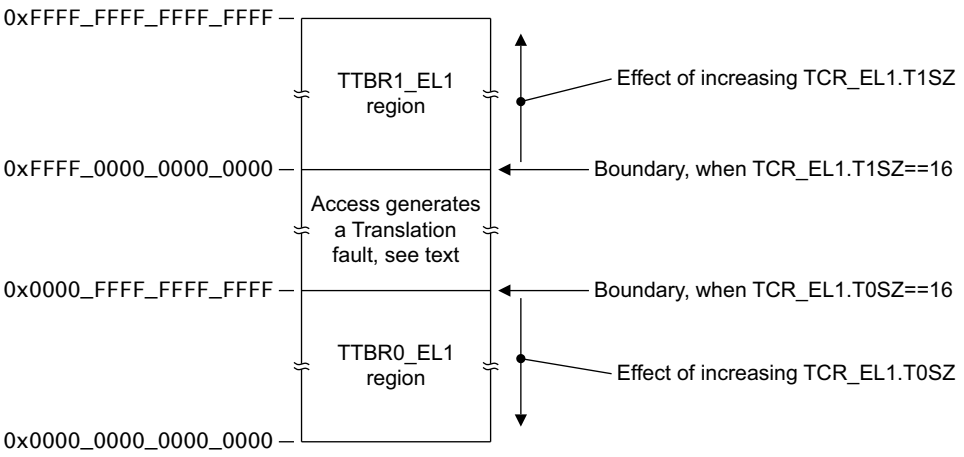



**Figure D4-15 AArch64 TTBR boundaries and VA ranges**

Which TTBR is used depends only on the VA presented for translation:

- If the top bits of the VA are zero, then TTBR0_EL1 is used.
- If the top bits of the VA are one, then TTBR1_EL1 is used.

It is configurable whether this determination depends on the values of VA[63:56] or on the values of VA[55:48], see *Address tagging in AArch64 state* on page D4-1742.

**Note**

The handling of the Contiguous bit can mean that the boundary between the translation regions defined by the TCR_EL1.TnSZ values and the region for which an access generates a Translation fault is wider than shown in Figure D4-15. That is, if the descriptor for an access to the region shown as generating a fault has the Contiguous bit set to 1, the access might not generate a fault. *Possible translation table registers programming errors* on page D4-1777 describes this possibility.

Example D4-3 on page D4-1775 shows a typical application of this VA split.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Example D4-3 Example use of the split VA range, and the TTBR0_EL1 and TTBR1_EL1 controls**

An example of using the split VA range is:

**TTBR0_EL1** Used for process-specific addresses.

Each process maintains a separate level 1 translation table. On a context switch:

- TTBR0_EL1 is updated to point to the level 1 translation table for the new context
- TCR_EL1 is updated if this change changes the size of the translation table
- CONTEXTIDR_EL1 is updated.

**TTBR1_EL1** Used for operating system and I/O addresses, that do not change on a context switch.

For each VA subrange, the input address size is $2^{(64-T_nSZ)}$, where T*n*SZ is one of TCR_EL1.{T0SZ, T1SZ},

This means the two VA subranges are:

**Lower VA subrange** `0x0000_0000_0000_0000` to $(2^{(64-T0SZ)} - 1)$.

**Upper VA subrange** $(2^{64} - 2^{(64-T1SZ)})$ to `0xFFFF_FFFF_FFFF_FFFF`.

The minimum T*n*SZ value is 16, corresponding to the maximum input address range of 48 bits. Example D4-4 shows the two VA subranges when T0SZ and T1SZ are both set to this minimum value.

**Example D4-4 Maximum VA ranges for EL1&0 stage 1 translations**

The maximum VA subranges correspond to T0SZ and T1SZ each having the minimum value of 16. In this case the subranges are:

**Lower VA subrange** `0x0000_0000_0000_0000` to `0x0000_FFFF_FFFF_FFFF`.

**Upper VA subrange** `0xFFFF_0000_0000_0000` to `0xFFFF_FFFF_FFFF_FFFF`.

Figure D4-15 on page D4-1774 indicates the effect of varying the T*n*SZ values.

As described in *Overview of the VMSAv8-64 address translation stages* on page D4-1762, the T*n*SZ values also determine the initial lookup level for the translation.

## Concatenated translation tables for the initial stage 2 lookup

*Overview of the VMSAv8-64 address translation stages* on page D4-1762 introduced the ability to concatenate translation tables for the initial stage 2 translation lookup. This section gives more information about that concatenation.

Where a stage 2 translation would require 16 entries or fewer in its top-level translation table, the system designer can instead:

- Require the corresponding number of concatenated translation tables at the next translation level, aligned to the size of the block of concatenated translation tables.
- Start the translation at that next translation level.

In addition, when using the 16KB translation granule and requiring a 48-bit input address size for the stage 2 translations, lookup must start with two concatenated translation tables at level 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

This translation scheme:

- Avoids the overhead of an additional level of translation.
- Requires the software that is defining the translation to:
  - Define the concatenated translation tables with the required overall alignment.
  - Program VTTBR_EL2 to hold the address of the first of the concatenated translation tables.
  - Program VTCR_EL2 to indicate the required input address range and initial lookup level.

Concatenating additional translation tables at the initial level of look up resolves additional address bits at that level. To resolve *n* additional address bits requires $2^n$ concatenated translation tables. Example D4-5 shows how, for level 1 lookups using the 4KB translation granule, translation tables can be concatenated to resolve three additional address bits.

**Example D4-5 Adding three bits of address resolution at level 1 lookup, using the 4KB granule**

When using the 4KB translation granule, a level1 lookup with a single translation table resolves address bits[38:30]. To add three more address bits requires $2^3$ translation tables, that is, eight translation tables. This means:

- The total size of the concatenated translation tables is 8×4KB=32KB.
- This block of concatenated translation tables must be aligned to 32KB.
- The address range resolved at this lookup level is A[41:30].of which:
  - Bits A[41:39] select the 4KB translation table.
  - Bits A[38:30] index a descriptor within that translation table.

As an example of the concatenation of translation tables at the initial lookup level, when using the 4KB translation granule, Table D4-24 shows the possible uses of concatenated translation tables to permit lookup to start at level 1 rather than at level 0. For completeness, the table starts with the case where the required IPA range means lookup starts at level 1with a single translation table at that level.

**Table D4-24 Possible uses of concatenated translation tables for level 1 lookup, 4KB granule**

| Configured stage 2 IA size | | Lookup starts at level 0 | Lookup starts at level 1 | |
|---|---|---|---|---|
| IPA range | Size | Required level 0 entries | Number of concatenated tables | Required alignment[a] |
| IPA[38:0] | $2^{36}$ bytes | - | 1 | 4KB |
| IPA[39:0] | $2^{37}$ bytes | 2 | 2 | 8KB |
| IPA[40:0] | $2^{38}$ bytes | 4 | 4 | 16KB |
| IPA[41:0] | $2^{39}$ bytes | 8 | 8 | 32KB |
| IPA[42:0] | $2^{40}$ bytes | 16 | 16 | 64KB |

a. Required alignment of the set of concatenated level 2 tables.

**Note**

Because concatenation is permitted only for a stage 2 translation, the input addresses in the table are IPAs.

*Overview of the VMSAv8-64 address translation stages* on page D4-1762 identifies all of the possible uses of concatenation. In all cases, the block of concatenated translation tables must be aligned to the block size.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Possible translation table registers programming errors

This subsection describes possible errors in programming the translation table registers.

### *Misprogramming the VTCR_EL2.{T0SZ, SL0} fields*

For a stage 2 translation, the programming of the VTCR_EL2.{T0SZ, SL0} fields must be consistent, see *Overview of the VMSAv8-64 address translation stages* on page D4-1762.

### *Misprogramming of the Contiguous bit*

For more information about the Contiguous bit, and the range of translation table entries that must have the bit set to 1 to mark the entries as contiguous, see *The Contiguous bit* on page D4-1819.

If one or more of the following errors is made in programming the translation tables, the TLB might contain overlapping entries:

- One or more of the contiguous translation table entries does not have the Contiguous bit set to 1.
- One or more of the contiguous translation table entries holds an output address that is not consistent with all of the entries pointing to the same aligned contiguous address range.
- The attributes and permissions of the contiguous entries are not all the same.

Such misprogramming of the translation tables means the output address, memory permissions, or attributes for a lookup might be corrupted, and might be equal to values that are not consistent with any of the programmed translation table values.

In some implementations, such misprogramming might also give rise to a TLB Conflict abort.

The architecture guarantees that misprogramming of the Contiguous bit cannot provide a mechanism for any of the following to occur:

- Software executing at EL1 or EL0 accessing regions of physical memory that are not accessible by programming the translation tables, from EL1, with arbitrary chosen values that do not misprogram the Contiguous bit.
- Software executing at EL1 or EL0 accessing regions of physical memory with attributes or permissions that are not possible by programming the translation tables, from EL1, with arbitrary chosen values that do not misprogram the Contiguous bit.
- Software executing in Non-secure state accessing Secure physical memory.

**Note**

Hardware implementations must ensure that use of the Contiguous bit cannot provide a mechanism for avoiding output address range checking. This might occur if a Contiguous bit block size of 0.5 GB or 1 GB is used in a system with the output address size configured to 4 GB. The architecture permits the implemented mechanism for preventing any avoidance of output address range checking to suppress the use of the Contiguous bit for such entries in such a system.

Where the Contiguous bit is used to mark a set of blocks as contiguous, if the address range translated by a set of blocks marked as contiguous is larger than the size of the input address supported at a stage of translation used to translate that address at that stage of translation, as defined by the TCR.TxSZ field, then this is a programming error. An implementation is permitted, but not required, to:

- Treat such a block within a contiguous set of blocks as causing a Translation fault, even though the block is valid, and the address accessed within that block is within the size of the input address supported at a stage of translation, as defined by the TCR.TxSZ field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Treat such a block within a contiguous set of blocks as not causing a Translation fault, even though the address accessed within that block is outside the size of the input address supported at a stage of translation, as defined by the TCR.TxSZ field, provided that both of the following apply:
  - The block is valid.
  - At least one address within the block, or contiguous set of blocks, is within the size of the input address supported at a stage of translation.

### D4.2.7 The algorithm for finding the translation table entries

This subsection gives the algorithms for finding the translation table entry that corresponds to a given IA, for each required level of lookup. The algorithms encode the descriptions of address translation given earlier in this section. The algorithm details depend on the translation granule size for the stage of address translation, see:

- *Finding the translation table entry when using the 4KB translation granule* on page D4-1779.
- *Finding the translation table entry when using the 16KB translation granule* on page D4-1780.
- *Finding the translation table descriptor when using the 64KB translation granule* on page D4-1781.

Each subsection uses the following terms:

**BaseAddress** The base address for the level of lookup, as defined by:
- For the initial lookup level, the appropriate TTBR.
- Otherwise, the translation table address returned by the previous level of lookup.

**PAMax** The supported PA width, in bits.

**IA** The supplied IA for this stage of translation.

**T*n*SZ** The translation table size for this stage of translation:

| | |
|---|---|
| **For EL1&0 stage 1** | TCR_EL1.T0SZ or TCR_EL1.T1SZ, as appropriate. |
| **For EL1&0 stage 2** | VTCR_EL2.T0SZ. |
| **For EL2 stage 1** | TCR_EL2.T0SZ. |
| **For EL3 stage 1** | TCR_EL3.T0SZ. |

**SL0** VTCR_EL2.SL0. Applies to the Non-secure EL1&0 stage 2 translation only.

These subsections show only architecturally-valid programming of the TCR. See also *Possible translation table registers programming errors* on page D4-1777.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Finding the translation table entry when using the 4KB translation granule

Table D4-25 shows the translation table descriptor address, for each level of lookup, when using the 4KB translation granule. See the start of *The algorithm for finding the translation table entries* on page D4-1778 for more information about terms used in the table.

**Table D4-25 Translation table entry addresses when using the 4KB translation granule**

| Lookup level | Entry address and conditions | | General conditions |
|---|---|---|---|
| | **Stage 1 translation** | **Stage 2 translation** | |
| Zero | BaseAddr[PAMax-1:*x*]:IA[*y*:39]:0b000<br>if[a] 16 ≤ T*n*SZ ≤ 24 then *x* = (28 - T*n*SZ) | BaseAddr[PAMax-1:*x*]:IA[*y*:39]:0b000<br>if SL0[b] == 2 then<br>  if[a] 16 ≤ T0SZ ≤ 24 then *x* = (28 - T0SZ) | *y* = (*x* + 35) |
| One | BaseAddr[PAMax-1:*x*]:IA[*y*:30]:0b000<br>if[a] 25 ≤ T*n*SZ ≤ 33 then *x* = (37 - T*n*SZ)<br><br>else[c] *x* =12 | BaseAddr[PAMax-1:*x*]:IA[*y*:30]:0b000<br>if SL0[b] == 1 then<br>  if[a] 21 ≤ T0SZ ≤ 33 then *x* = (37 - T0SZ)<br>elsif SL0[b] == 2 then *x* = 12 | *y* = (*x* + 26) |
| Two | BaseAddr[PAMax-1:*x*]:IA[*y*:21]:0b000<br>if[a] 34 ≤ T*n*SZ ≤ 39 then *x* = (46 - T*n*SZ)<br><br>else[c] *x* =12 | BaseAddr[PAMax-1:*x*]:IA[*y*:21]:0b000<br>if SL0[b] == 0 then<br>  if[a] 30 ≤ T0SZ ≤ 39 then *x* = (46 - T0SZ)<br>elsif SL0[b] > 0 then *x* = 12 | *y* = (*x* + 17) |
| Three | BaseAddr[PAMax-1:12]:IA[20:12]:0b000 | BaseAddr[PAMax-1:12]:IA[20:12]:0b000 | - |

a. This line indicates the range of permitted values for T*n*SZ, for a lookup that starts at this level, see *Overview of VMSAv8-64 address translation using the 4KB translation granule* on page D4-1763.

b. SL0 == 0 if the initial lookup is level 2, SL0 == 1 if the initial lookup is level 1, and SL0 ==2 if the initial lookup level is level 0.

c. This is the case where this level of lookup is not the initial level of lookup.

Table D4-7 on page D4-1755 shows how software can determine whether an implementation supports the 4KB granule size.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Finding the translation table entry when using the 16KB translation granule

Table D4-26 shows the translation table descriptor address, for each level of lookup, when using the 16KB translation granule. See the start of *The algorithm for finding the translation table entries* on page D4-1778 for more information about terms used in the table.

**Table D4-26 Translation table entry addresses when using the 16KB translation granule**

| Lookup level | Entry address and conditions | | General conditions |
|---|---|---|---|
| | Stage 1 translation | Stage 2 translation | |
| Zero | BaseAddr[PAMax-1:4]:IA[47]:0b000<br>[a] 16 ≤ T*n*SZ | - | Only applies to stage 1 |
| One | BaseAddr[PAMax-1:*x*]:IA[*y*:36]:0b000<br>if[a] 17 ≤ T*n*SZ ≤ 27 then *x* = (31 - T*n*SZ)<br><br>else[b] *x* =14 | BaseAddr[PAMax-1:*x*]:IA[*y*:36]:0b000<br>if SL0[c] == 2 then<br>  if[a] 16 ≤ T0SZ ≤ 27 then *x* = (31 - T0SZ) | *y* = (*x* + 32) |
| Two | BaseAddr[PAMax-1:*x*]:IA[*y*:25]:0b000<br>if[a] 28 ≤ T*n*SZ ≤ 38 then *x* = (42 - T*n*SZ)<br><br>else[b] *x* =14 | BaseAddr[PAMax-1:*x*]:IA[*y*:25]:0b000<br>if SL0[c] == 1 then<br>  if[a] 24 ≤ T0SZ ≤ 38 then *x* = (42 - T0SZ)<br>elsif SL0[c] == 2 then *x* = 14 | *y* = (*x* + 21) |
| Three | BaseAddr[PAMax-1:14]:IA[24:14]:0b000 | BaseAddr[PAMax-1:*x*]:IA[*y*:14]:0b000<br>if SL0[c] == 0 then<br>  if[a] 35 ≤ T0SZ ≤ 39 then *x* = (53 - T0SZ)<br>elsif SL0[c] >0 then *x* = 14 | *y* = (*x* + 10) |

a. This line indicates the range of permitted values for T*n*SZ, for a lookup that starts at this level, see *Overview of VMSAv8-64 address translation using the 16KB translation granule* on page D4-1766.

b. This is the case where this level of lookup is not the initial level of lookup.

c. SL0 == 0 if the initial lookup is level 3, SL0 == 1 if the initial lookup is level 2, and SL0 ==2 if the initial lookup level is level 1.

Table D4-7 on page D4-1755 shows how software can determine whether an implementation supports the 16KB granule size.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Finding the translation table descriptor when using the 64KB translation granule

Table D4-27 shows the translation table descriptor address, for each level of lookup, when using the 64KB translation granule. See the start of *The algorithm for finding the translation table entries* on page D4-1778 for more information about terms used in the table.

**Table D4-27 Translation table entry addresses when using the 64KB translation granule**

| Lookup level | Entry address and conditions | | General conditions |
|---|---|---|---|
| | Stage 1 translation | Stage 2 translation | |
| One | BaseAddr[PAMax-1:*x*]:IA[*y*:42]:0b000<br>if[a] 16 ≤ T*n*SZ ≤ 21 then *x* = (25 - T*n*SZ) | BaseAddr[PAMax-1:*x*]:IA[*y*:42]:0b000<br>if SL0[b] == 2 then<br>  if[a] 16 ≤ T0SZ ≤ 21 then *x* = (25 - T0SZ) | *y* = (*x* + 38) |
| Two | BaseAddr[PAMax-1:*x*]:IA[*y*:29]:0b000<br>if[a] 22 ≤ T*n*SZ ≤ 34 then *x* = (38 - T*n*SZ)<br>else[c] *x* =16 | BaseAddr[PAMax-1:*x*]:IA[*y*:29]:0b000<br>if SL0[b] == 1 then<br>  if[a] 18 ≤ T0SZ ≤ 34 then *x* = (38 - T0SZ)<br>elsif SL0[b] == 2 then *x* = 16 | *y* = (*x* + 25) |
| Three | BaseAddr[PAMax-1:*x*]:IA[*y*:16]:0b000<br>if[a] 35 ≤ T*n*SZ ≤ 39 then *x* = (51 - T*n*SZ)<br>else[c] *x* =16 | BaseAddr[PAMax-1:*x*]:IA[*y*:16]:0b000<br>if SL0[b] == 0 then<br>  if[a] 31 ≤ T0SZ ≤ 39 then *x* = (51 - T0SZ)<br>elsif SL0[b] > 0 then *x* = 16 | *y* = (*x* + 12) |

a. This line indicates the range of permitted values for T*n*SZ, for a lookup that starts at this level, see *Overview of VMSAv8-64 address translation using the 64KB translation granule* on page D4-1769.

b. SL0 == 0 if the initial lookup is level 3, SL0 == 1 if the initial lookup is level 2, and SL0 ==2 if the initial lookup level is at level 1.

c. This is the case where this level of lookup is not the initial level of lookup.

Table D4-7 on page D4-1755 shows how software can determine whether an implementation supports the 64KB granule size.

## D4.2.8 The effects of disabling a stage of address translation

The following sections describe the effect on MMU behavior of disabling each stage of translation:

- *Behavior when stage 1 address translation is disabled*.
- *Behavior when stage 2 address translation is disabled* on page D4-1782.
- *Behavior of instruction fetches when all associated stages of translation are disabled* on page D4-1783.

### Behavior when stage 1 address translation is disabled

When a stage 1 address translation is disabled, memory accesses that would otherwise be translated by that stage of translation are treated as follows:

**Non-secure EL1 and EL0 accesses if the HCR_EL2.DC bit is set to 1**

For the Non-secure EL1&0 translation regime, when the value of HCR_EL2.DC is 1, the stage 1 translation assigns the Normal Non-shareable, Inner Write-Back Read-Write-Allocate, Outer Write-Back Read-Write-Allocate memory attributes.

**Note**

This applies for both instruction and data accesses.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**All other accesses**

For all other accesses, when stage 1 address translation is disabled, the assigned attributes depend on whether the access is a data access or an instruction access, as follows:

**Data access**

The stage 1 translation assigns the Device-nGnRnE memory type.

**Instruction access**

The stage 1 translation assigns the Normal memory attribute, with the cacheability and shareability attributes determined by the value of the SCTLR.I bit for the translation regime, as follows:

**When the value of I is 0**

The stage 1 translation assigns the Non-cacheable and Outer Shareable attributes.

**When the value of I is 1**

The stage 1 translation assigns the Cacheable, Inner Write-Through no Write-Allocate Read-Allocate, Outer Write-Through no Write-Allocate Read Allocate Outer Shareable attribute.

For this stage of translation, no memory access permission checks are performed, and therefore no MMU faults can be generated for this stage of address translation.

**Note**

Alignment checking is performed, and therefore Alignment faults can occur.

For every access, the input address of the stage 1 translation is flat-mapped to the output address.

For a Non-secure EL1 or EL0 access, if EL1&0 stage 2 address translation is enabled, the stage 1 memory attribute assignments and output address can be modified by the stage 2 translation.

When the value of HCR_EL2.DC is 1, in Non-secure state:

- The SCTLR_EL1.M bit behaves as if it is 0, for all purposes other than reading the value of the bit. This means Non-secure EL1&0 stage 1 address translation is disabled.
- The HCR_EL2.VM bit behaves as if it is 1, for all purposes other than reading the value of the bit. This means that Non-secure EL1&0 stage 2 address translation is enabled.

See also *Behavior of instruction fetches when all associated stages of translation are disabled* on page D4-1783.

#### *Effect of disabling address translation on maintenance and address translation instruction instructions*

Cache maintenance instructions act on the target cache regardless of whether any stages of address translation are disabled, and regardless of the values of the memory attributes. However, if a stage of address translation is disabled, they use the flat address mapping for that translation stage.

TLB invalidate operations act on the target TLB regardless of whether any stage of address translation is disabled.

The value of HCR_EL2.DC affect some address translation instructions, see *Address translation instructions, AT** on page D4-1796.

### Behavior when stage 2 address translation is disabled

When stage 2 address translation is disabled:

- The IPA output from the stage 1 translation maps flat to the PA.
- The memory attributes and permissions from the stage 1 translation apply to the PA.

When both stages of address translation are disabled, see also *Behavior of instruction fetches when all associated stages of translation are disabled* on page D4-1783.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Behavior of instruction fetches when all associated stages of translation are disabled

When EL3 is using AArch64, this section applies to:

- The Secure EL1&0 translation regime when Secure EL1&0 stage 1 address translation is disabled.
- The Secure EL3 translation regime, when Secure EL3 stage 1 address translation is disabled.
- The Non-secure EL2 translation regime, when Non-secure EL2 stage 1 address translation is disabled
- The Non-secure EL1&0 translation regime, when both stages of address translation are disabled.

**Note**

- The behaviors in Non-secure state apply regardless of the Execution state that EL3 is using.
- When the value of HCR_EL2.DC is 1, then the behavior of the Non-secure EL1&0 translation regime is as if stage 1 translation is disabled and stage 2 translation is enabled, as described in *Behavior when stage 1 address translation is disabled* on page D4-1781.

In these cases, a memory location might be accessed as a result of an instruction fetch if the following condition is met:

- The memory location is in the same block of memory as, or in the next contiguous block of memory to, an instruction that a simple sequential execution of the program either requires to be fetched now or has required to be fetched since any of:
  - The last reset.
  - The last synchronization of instruction cache maintenance targeting the address of the instruction.
  - When executing in AArch32 state, the last synchronization of branch predictor maintenance targeting the address of the instruction.

  In this description, the blocks of memory referred to are of the size of the minimum implemented translation granule and are aligned to that size.

These accesses can be caused by speculative instruction fetches, regardless of whether the prefetched instruction is committed for execution.

**Note**

To ensure architectural compliance, software must ensure that both of the following apply:

- Instructions that will be executed when all associated stages of address translation are disabled are located in blocks of the address space, of the translation granule size, that contain only memory that is tolerant to speculative accesses.
- Each block of the address space, of the translation granule size, that immediately follows a similar block that holds instructions that will be executed when all associated stages address translation are disabled, contains only memory that is tolerant to speculative accesses.

## D4.2.9 The implemented Exception levels and the resulting translation stages and regimes

Elsewhere, this chapter describes an implementation that includes all Exception levels, and describes the control of address translation by Exception levels that are using AArch64. This subsection describes how the address translation scheme changes if an implementation does not include all of the Exception levels.

If an implementation does not include EL3, it has only a single Security state, with MMU controls equivalent to the Secure state MMU controls.

If an implementation does not include EL2 then:

- If it also does not include EL3, the MMU provides only a single EL1&0 stage 1 translation regime.
- If it includes EL3, the MMU provides an EL1&0 stage 1 translation regime in each Security state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Figure D4-2 on page D4-1746 shows the set of translation regimes for an implementation that implements all of the Exception levels. Table D4-28 shows how the supported translation stages depend on the implemented Exception levels, and in some cases on the Execution state being used by the highest implemented Exception level.

**Table D4-28 The relation between the implemented translation stages and Exception levels for AArch64**

| Translation stage | Requires |
|---|---|
| Secure EL3 stage 1 | EL3 implemented and using AArch64. |
| Secure EL1&0 stage 1 | Either:<br>• EL3 implemented and using AArch64.<br>• Only EL1 and EL0 implemented, all operation is in Secure state, and EL1 is using AArch64. |
| Non-secure EL2 stage 1 | EL2 implemented. |
| Non-secure EL1&0 stage 2 | EL2 implemented. |
| Non-secure EL1&0 stage 1 | Any implementation except:<br>• Only EL1 and EL0 implemented, with all operation in the Secure state. |

## D4.2.10 Pseudocode description of VMSAv8-64 address translation

The following subsections gives a pseudocode description of the translation table walk:

- *Definitions required for address translation*.
- *Performing the full address translation*.
- *Stage 1 translation* on page D4-1785.
- *Stage 2 translation* on page D4-1787.
- *Translation table walk* on page D4-1788.
- *Support functions* on page D4-1793.

### Definitions required for address translation

In pseudocode, the result of a translation table lookup, in either Execution state, is returned in a `TLBRecord` structure.

```
type TLBRecord is (
    Permissions       perms,
    bit               nG,             // '0' = Global, '1' = not Global
    bits(4)           domain,         // AArch32 only
    boolean           contiguous,     // Contiguous bit from page table
    integer           level,          // In AArch32 Short-descriptort format, indicates Section/Page
    integer           blocksize,      // Describes size of memory translated in KBytes
    AddressDescriptor addrdesc
)
```

*Memory data type definitions* on page D3-1730 includes definitions of the `Permissions` and `AddressDescriptor` parameters.

### Performing the full address translation

The function `AArch64.FullTranslate()` performs a full translation table walk. For any translation regime it performs a stage 1 translation for the supplied virtual address, and for the Non-secure EL1&0 translation regime it then performs a stage 2 translation of the returned address.

```
// AArch64.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch64.FullTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch64.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !IsFault(S1) && HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        s2fs1walk = FALSE;
        result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                              size);
    else
        result = S1;

    return result;
```

## Stage 1 translation

The function `AArch64.FirstStageTranslate()` performs a stage 1 translation, calling the function `AArch64.TranslationTableWalk()`, described in *Translation table walk* on page D4-1788, to perform the required translation table walk. However, if stage 1 translation is disabled, it calls the function `AArch64.TranslateAddressS1Off()`, described in this section, to set the memory attributes.

```
// AArch64.FirstStageTranslate()
// =============================
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.FirstStageTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                              boolean wasaligned, integer size)

    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';
    else
        s1_enabled = SCTLR[].M == '1';

    ipaddress = bits(48) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                          // First stage enabled
        S1 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);
        permissioncheck = TRUE;
    else
        S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;

    // Check for unaligned data accesses to Device memory
    if (!wasaligned && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
        acctype != AccType_IFETCH) then
        S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level,
                                                    S1.addrdesc.paddress.NS,
                                                    acctype, iswrite);

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
        acctype == AccType_IFETCH) then
        S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                                acctype, iswrite,
                                                secondstage, s2fs1walk);

    return S1.addrdesc;
```

When stage 1 translation is disabled, the function `AArch64.TranslateAddressS1Off()` sets the memory attributes.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// AArch64.TranslateAddressS1Off()
// ===============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch64.TranslateAddressS1Off(bits(64) vaddress, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    Top = AddrTop(vaddress);
    if !IsZero(vaddress<Top:PAMax()>) then
        level = 0;
        ipaddress = bits(48) UNKNOWN;
        secondstage = FALSE;
        s2fs1walk = FALSE;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    default_cacheable = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} && HCR_EL2.DC == '1');

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.type = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;      // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
    elsif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.type = MemType_Device;
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
    else
        // Instruction cacheability controlled by SCTLR_ELx.I
        cacheable = SCTLR[].I == '1';
        result.addrdesc.memattrs.type = MemType_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            result.addrdesc.memattrs.inner.hints = MemHint_No;
        result.addrdesc.memattrs.shareable = TRUE;
        result.addrdesc.memattrs.outershareable = TRUE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.physicaladdress = vaddress<47:0>;
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch64.NoFault();

    return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Stage 2 translation

In the Non-secure EL1&0 translation regime, a descriptor address returned by stage 1 lookup is in the IPA address space, and must be mapped to a PA by a stage 2 translation. Function `AArch64.SecondStageWalk()` performs this translation, by calling the `AArch64.SecondStageTranslate()` function.When called from `AArch64.SecondStageWalk()`, the `AArch64.SecondStageTranslate()` function performs a second stage translation, from IPA to PA, of the supplied address, including checking that the access has read permission at the second stage. If the access does not have second stage read permission it generates a second stage Permission fault on the first stage translation table walk. The second stage translation might hit in a TLB, or might involve a translation table walk, which will use the algorithm described in this section.

```
// AArch64.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype,
                                          boolean iswrite, integer size)
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                        size);
```

The `AArch64.SecondStageTranslate()` function performs the stage 2 address translation.

```
// AArch64.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.SecondStageTranslate(AddressDescriptor S1, bits(64) vaddress,
                                               AccType acctype, boolean iswrite, boolean wasaligned,
                                               boolean s2fs1walk, integer size)
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};

    s2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';
    secondstage = TRUE;

    if s2_enabled then                          // Second stage enabled
        ipaddress = S1.paddress.physicaladdress<47:0>;
        S2 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if (!wasaligned && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype != AccType_IFETCH) then
            S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                          acctype, iswrite, s2fs1walk);

        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype == AccType_IFETCH) then
            S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    acctype, iswrite,
                                                    secondstage, s2fs1walk);

        // Check for protected table walk
        if (s2fs1walk && !IsFault(S2.addrdesc) && HCR_EL2.PTW == '1' &&
            S2.addrdesc.memattrs.type == MemType_Device) then
            S2.addrdesc.fault = AArch64.PermissionFault(ipaddress, S2.level, acctype,
                                                        iswrite, secondstage, s2fs1walk);
        result = CombineS1S2Desc(S1, S2.addrdesc);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        result = S1;

    return result;
```

### Translation table walk

The function AArch64.TranslationTableWalk() returns the result, in the form of a TLBRecord, of a translation table walk made for a memory access from an Exception level that is using AArch64.

```
// AArch64.TranslationTableWalk()
// ==============================
// Returns a result of a translation table walk
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch64.TranslationTableWalk(bits(48) ipaddress, bits(64) vaddress,
                                       AccType acctype, boolean iswrite, boolean secondstage,
                                       boolean s2fs1walk, integer size)
    if !secondstage then
        assert !ELUsingAArch32(S1TranslationRegime());
    else
        assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && PSTATE.EL != EL2;

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(64) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2

    descaddr.memattrs.type = MemType_Normal;

    // Derived parameters for the page table walk:
    //  grainsize = Log2(Size of Table)         - Size of Table is 4KB, 16KB or 64KB in AArch64
    //  stride = Log2(Address per Level)        - Bits of address consumed at each level
    //  firstblocklevel = First level where a block entry is allowed
    //  ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTCR_EL2.PS
    //  inputsize = Log2(Size of Input Address) - Input Address size in bits
    //  level = Level to start walk from
    // This means that the number of levels after start level = 3-level

    if !secondstage then
        // First stage translation
        inputaddr = ZeroExtend(vaddress);
        top = AddrTop(inputaddr);

        if PSTATE.EL == EL3 then
            inputsize = 64 - UInt(TCR_EL3.T0SZ);
            if inputsize > 48 then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = 48;
            if inputsize < 25 then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = 25;
            largegrain = TCR_EL3.TG0 == '01';
            midgrain = TCR_EL3.TG0 == '10';
            ps = TCR_EL3.PS;
            basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<top:inputsize>);
            disabled = FALSE;
            baseregister = TTBR0_EL3;
            descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0);
            reversedescriptors = SCTLR_EL3.EE == '1';
            lookupsecure = TRUE;
            singlepriv = TRUE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
elsif PSTATE.EL == EL2 then
    inputsize = 64 - UInt(TCR_EL2.T0SZ);
    if inputsize > 48 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = 48;
    if inputsize < 25 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = 25;
    largegrain = TCR_EL2.TG0 == '01';
    midgrain = TCR_EL2.TG0 == '10';
    ps = TCR_EL2.PS;
    basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<top:inputsize>);
    disabled = FALSE;
    baseregister = TTBR0_EL2;
    descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0);
    reversedescriptors = SCTLR_EL2.EE == '1';
    lookupsecure = FALSE;
    singlepriv = TRUE;

else
    if inputaddr<top> == '0' then
        inputsize = 64 - UInt(TCR_EL1.T0SZ);
        if inputsize > 48 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 48;
        if inputsize < 25 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 25;
        largegrain = TCR_EL1.TG0 == '01';
        midgrain = TCR_EL1.TG0 == '10';
        basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<top:inputsize>);
        disabled = TCR_EL1.EPD0 == '1';
        baseregister = TTBR0_EL1;
        descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORGN0, TCR_EL1.IRGN0);
    else
        inputsize = 64 - UInt(TCR_EL1.T1SZ);
        if inputsize > 48 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 48;
        if inputsize < 25 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 25;
        largegrain = TCR_EL1.TG1 == '11';       // TG1 and TG0 encodings differ
        midgrain = TCR_EL1.TG1 == '01';
        basefound = inputsize >= 25 && inputsize <= 48 && IsOnes(inputaddr<top:inputsize>);
        disabled = TCR_EL1.EPD1 == '1';
        baseregister = TTBR1_EL1;
        descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGN1, TCR_EL1.IRGN1);
    ps = TCR_EL1.IPS;
    reversedescriptors = SCTLR_EL1.EE == '1';
    lookupsecure = IsSecure();
    singlepriv = FALSE;

if largegrain then
    grainsize = 16;                             // Log2(64KB page size)
    firstblocklevel = 2;                        // Largest block is 512MB (2^29 bytes)
elsif midgrain then
    grainsize = 14;                             // Log2(16KB page size)
    firstblocklevel = 2;                        // Largest block is 32MB (2^25 bytes)
else // Small grain
    grainsize = 12;                             // Log2(4KB page size)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            firstblocklevel = 1;                        // Largest block is 1GB (2^30 bytes)
        stride = grainsize - 3;                         // Log2(page size / 8 bytes)
        // The starting level is the number of strides needed to consume the input address
        level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));

    else
        // Second stage translation
        inputaddr = ZeroExtend(ipaddress);
        inputsize = 64 - UInt(VTCR_EL2.T0SZ);
        if inputsize > 48 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 48;
        if inputsize < 25 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 25;
        largegrain = VTCR_EL2.TG0 == '01';
        midgrain = VTCR_EL2.TG0 == '10';
        ps = VTCR_EL2.PS;
        basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<63:inputsize>);
        disabled = FALSE;
        baseregister = VTTBR_EL2;
        descaddr.memattrs = WalkAttrDecode(VTCR_EL2.IRGN0, VTCR_EL2.ORGN0, VTCR_EL2.SH0);
        reversedescriptors = SCTLR_EL2.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;

        startlevel = UInt(VTCR_EL2.SL0);
        if largegrain then
            grainsize = 16;                             // Log2(64KB page size)
            level = 3 - startlevel;
            firstblocklevel = 2;                        // Largest block is 512MB (2^29 bytes)
        elsif midgrain then
            grainsize = 14;                             // Log2(16KB page size)
            level = 3 - startlevel;
            firstblocklevel = 2;                        // Largest block is 32MB (2^25 bytes)
        else // Small grain
            grainsize = 12;                             // Log2(4KB page size)
            level = 2 - startlevel;
            firstblocklevel = 1;                        // Largest block is 1GB (2^30 bytes)
        stride = grainsize - 3;                         // Log2(page size / 8 bytes)

        // Limits on IPA controls based on implemented PA size. Level 0 is only
        // supported by small grain translations
        if largegrain then                              // 64KB pages
            // Level 1 only supported if implemented PA size is greater than 2^42 bytes
            if level == 0 || (level == 1 && PAMax() <= 42) then basefound = FALSE;
        elsif midgrain then                             // 16KB pages
            // Level 1 only supported if implemented PA size is greater than 2^40 bytes
            if level == 0 || (level == 1 && PAMax() <= 40) then basefound = FALSE;
        else                                            // Small grain, 4KB pages
            // Level 0 only supported if implemented PA size is greater than 2^42 bytes
            if level < 0 || (level == 0 && PAMax() <= 42) then basefound = FALSE;

        // If the inputsize exceeds the PAMax value, the behavior is CONSTRAINED UNPREDICTABLE
        inputsizecheck = inputsize;
        if inputsize > PAMax() && (!ELUsingAArch32(EL1) || inputsize > 40) then
            case ConstrainUnpredictable() of
                when Constraint_FORCE
                    // Restrict the inputsize to the PAMax value
                    inputsize = PAMax();
                    inputsizecheck = PAMax();
                when Constraint_FORCENOSLCHECK
                    // As FORCE, except use the configured inputsize in the size checks below
                    inputsize = PAMax();
                when Constraint_FAULT
                    // Generate a translation fault
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                basefound = FALSE;
            otherwise
                Unreachable();

    // Number of entries in the starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    startsizecheck = inputsizecheck - ((3 - level)*stride + grainsize); // Log2(Num of entries)

    // Check for starting level table with fewer than 2 entries or longer than 16 pages.
    // Lower bound check is:  startsizecheck < Log2(2 entries)
    // Upper bound check is:  startsizecheck > Log2(pagesize/8*16)
    if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

if !basefound || disabled then
    level = 0;           // AArch32 reports this as a level 1 fault
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

case ps of
    when '000'  outputsize = 32;
    when '001'  outputsize = 36;
    when '010'  outputsize = 40;
    when '011'  outputsize = 42;
    when '100'  outputsize = 44;
    when '101'  outputsize = 48;
    otherwise   outputsize = 48;

if outputsize > PAMax() then outputsize = PAMax();

if outputsize != 48 && !IsZero(baseregister<47:outputsize>) then
    level = 0;
    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

// Bottom bound of the Base address is:
//     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
//     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
baseaddress = baseregister<47:baselowerbound>:Zeros(baselowerbound);

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

addrselecttop = inputsize - 1;

repeat
    addrselectbottom = (3-level)*stride + grainsize;

    bits(48) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
    descaddr.paddress.physicaladdress = baseaddress OR index;
    descaddr.paddress.NS = ns_table;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if !HaveEL(EL2) || secondstage || IsSecure() || PSTATE.EL == EL2 then
        descaddr2 = descaddr;
    else
        descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
        // Check for a fault on the stage 2 walk
        if IsFault(descaddr2) then
            result.addrdesc.fault = descaddr2.fault;
            return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Update virtual address for abort functions
    descaddr2.vaddress = ZeroExtend(vaddress);

    desc = _Mem[descaddr2, 8, AccType_PTW];
    if reversedescriptors then desc = BigEndianReverse(desc);

    if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
         // Fault (00), Reserved (10), or Block (01) at level 3
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Valid Block, Page, or Table entry
    if desc<1:0> == '01' || level == 3 then                 // Block (01) or Page (11)
        blocktranslate = TRUE;
    else                                                    // Table (11)
        if outputsize != 48 && !IsZero(desc<47:outputsize>) then
            result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        baseaddress = desc<47:grainsize>:Zeros(grainsize);

        if !secondstage then
            // Unpack the upper and lower table attributes
            ns_table    = ns_table    OR desc<63>;
            ap_table<1> = ap_table<1> OR desc<62>;       // read-only
            xn_table    = xn_table    OR desc<60>;
            // pxn_table and ap_table[0] apply only in EL1&0 translation regimes
            if !singlepriv then
                ap_table<0> = ap_table<0> OR desc<61>;   // privileged
                pxn_table   = pxn_table   OR desc<59>;

        level = level + 1;
        addrselecttop = addrselectbottom - 1;
        blocktranslate = FALSE;
until blocktranslate;

// Check block size is supported at this level
if level < firstblocklevel then
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                     iswrite, secondstage, s2fs1walk);
    return result;

// Check for misprogramming of the contiguous bit
if largegrain then
    contiguousbitcheck = level == 2 && inputsize < 34;
elsif midgrain then
    contiguousbitcheck = level == 2 && inputsize < 30;
else
    contiguousbitcheck = level == 1 && inputsize < 34;

if contiguousbitcheck && desc<52> == '1' then
    if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

// Check the output address is inside the supported range
if outputsize != 48 && !IsZero(desc<47:outputsize>) then
    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                     iswrite, secondstage, s2fs1walk);
    return result;

// Check the access flag
if desc<10> == '0' then
    result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return result;

    // Unpack the descriptor into address and upper and lower block attributes
    outputaddress = desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
    xn = desc<54>;
    pxn = desc<53>;
    contiguousbit = desc<52>;
    nG = desc<11>;
    sh = desc<9:8>;
    ap = desc<7:6>:'1';
    mattr = desc<5:2>;                                      // AttrIndx and NS bit in stage 1

    result.domain = bits(4) UNKNOWN;                        // Domains not used
    result.level = level;
    result.blocksize = 2^((3-level)*stride + grainsize);

    // Stage 1 translation regimes also inherit attributes from the tables
    if !secondstage then
        result.perms.xn      = xn OR xn_table;
        result.perms.ap<2>   = ap<2> OR ap_table<1>;        // Force read-only

        // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
        if !singlepriv then
            result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
            result.perms.pxn   = pxn OR pxn_table;
            // Pages from Non-secure tables are marked non-global in Secure EL1&0
            if IsSecure() then
                result.nG = nG OR ns_table;
            else
                result.nG = nG;
        else
            result.perms.ap<1> = '1';
            result.perms.pxn   = '0';
            result.nG          = '0';
        result.perms.ap<0>   = '1';
        result.addrdesc.memattrs = AArch64.S1AttrDecode(sh, mattr<2:0>, acctype);
        result.addrdesc.paddress.NS = mattr<3> OR ns_table;
    else
        result.perms.ap<2:1> = ap<2:1>;
        result.perms.ap<0>   = '1';
        result.perms.xn      = xn;
        result.perms.pxn     = '0';
        result.nG            = '0';
        result.addrdesc.memattrs = S2AttrDecode(sh, mattr, acctype);
        result.addrdesc.paddress.NS = '1';

    result.addrdesc.paddress.physicaladdress = outputaddress;
    result.addrdesc.fault = AArch64.NoFault();
    result.contiguous = contiguousbit == '1';

    return result;
```

## Support functions

In the translation table walk functions, the `WalkAttrDecode()` function determines the attributes for a translation table lookup.

```
// WalkAttrDecode()
// ================

MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN)

    MemoryAttributes memattrs;

    AccType acctype = AccType_NORMAL;

    memattrs.type = MemType_Normal;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    mematttrs.inner = ShortConvertAttrsHints(IRGN, acctype);
    mematttrs.outer = ShortConvertAttrsHints(ORGN, acctype);
    mematttrs.shareable = SH<1> == '1';
    mematttrs.outershareable = SH == '10';

    return MemAttrDefaults(mematttrs);
```

The function AArch64.S1AttrDecode() decodes the attributes from a stage 1 translation table lookup.

```
// AArch64.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch64.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes mematttrs;

    mair = MAIR[];
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();

    if attrfield<7:4> == '0000' then            // Device
        mematttrs.type = MemType_Device;
        case attrfield<3:0> of
            when '0000'  mematttrs.device = DeviceType_nGnRnE;
            when '0100'  mematttrs.device = DeviceType_nGnRE;
            when '1000'  mematttrs.device = DeviceType_nGRE;
            when '1100'  mematttrs.device = DeviceType_GRE;
            otherwise    Unreachable();         // Reserved, handled above

    elsif attrfield<3:0> != '0000'  then        // Normal
        mematttrs.type = MemType_Normal;
        mematttrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        mematttrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        mematttrs.shareable = SH<1> == '1';
        mematttrs.outershareable = SH == '10';

    else
        Unreachable();                          // Reserved, handled above

    return MemAttrDefaults(mematttrs);
```

The function AArch64.CheckPermission() checks the access permissions returned by a stage 1 translation table lookup, see *Access permission checking* on page D3-1735.

The function AArch64.CheckS2Permission() checks the access permissions returned by a stage 2 translation table lookup.

```
// AArch64.CheckS2Permission()
// ===========================
// Function used for permission checking from AArch64 stage 2 translations

FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, bits(48) ipaddress,
                                      integer level, AccType acctype, boolean iswrite,
                                      boolean s2fs1walk)
    assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && PSTATE.EL != EL2;

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    xn = perms.xn == '1';

    // Stage 1 walk is checked as a read, regardless of the original type
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if acctype == AccType_IFETCH && !s2fs1walk then
    fail = xn;
elsif iswrite && !s2fs1walk then
    fail = !w;
    failedread = FALSE;
else
    fail = !r;
    failedread = TRUE;

if fail then
    domain = bits(4) UNKNOWN;
    secondstage = TRUE;
    return AArch64.PermissionFault(ipaddress, level, acctype,
                                   !failedread, secondstage, s2fs1walk);
else
    return AArch64.NoFault();
```

The `AddrTop()` function returns the bit number of the most significant valid bit of a VA in the current translation regime. If EL1 is using AArch64 and EL0 is using AArch32 then an address from EL0 is zero-extended to 64 bits.

```
integer AddrTop(bits(64) address);
```

### D4.2.11 Address translation instructions

Each of the ARMv8 instruction sets provides instructions that return the result of translating an input address, supplied as an argument to the instruction, using a specified translation stage or regime.

The available instructions only perform translations that are accessible from the Security state and Exception level at which the instruction is executed. That is:

- No instruction executed in Non-secure state can return the result of a Secure address translation stage.
- No instruction can return the result of an address translation stage that is controlled by an Exception level that is higher than the Exception level at which the instruction is executed.

*Address translation instructions, AT** on page D4-1796 summarizes the A64 address translation instructions.

See also *A64 system instructions for address translation* on page C5-358.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Address translation instructions, AT*

The A64 assembly language syntax for address translation instructions is:

```
AT <operation>, <Xt>
```

Where:

<operation> Is one of S1E1R, S1E1W, S1E0R, S1E0W, S12E1R, S12E1W, S12E0R, S12E0W, S1E2R, S1E2W, S1E3R, or S1E3W.
<operation> has a structure of <stages><level><read|write>, where:

<stages> Is one of:

- S1 Stage 1 translation.
- S12 Stage 1 translation followed by stage 2 translation.

<level> Describes the Exception Level that the translation applies to. Is one of:

- E0 EL0.
- E1 EL1.
- E2 EL2.
- E3 EL3.

If <level> is higher than the current Exception Level the instruction is UNDEFINED.

<read|write>

Is one of:

- R Read.
- W Write.

<Xt> The address to be translated. No alignment restrictions apply for the address.

If EL2 is not implemented, the AT S1E2R and AT S1E2W instructions are UNDEFINED.

**Note**

If EL2 is not implemented but EL3 is implemented, the AT S12E* instructions are not UNDEFINED, but behave the same way as the equivalent AT S1E* instructions. This is consistent with the behavior if EL2 is implemented but stage 2 translation is disabled.

In each case, the address being translated is held in the 64-bit address argument register, Xt. If the address translation instruction uses a translation regime that is using AArch32, meaning it requires a VA of only 32 bits, then VA[63:32] is RES0.

If the address translation is successful, the resulting PA is returned in PAR_EL1.PA, and PAR_EL1.F is set to 0 to indicate that the translation was successful. Otherwise, see *Synchronous faults generated by address translation instructions* on page D4-1797.

**Note**

The architecture provides a single PAR, PAR_EL1, that is used regardless of:

- The Exception level at which the instruction was executed.
- The Exception level that controls the stage or stages of translation used by the instruction.

For all of these instructions, the current context information determines which entries in TLB caching structures are used, and how the translation table walk is performed. However, it is IMPLEMENTATION DEFINED whether the Address translation instructions return the values held in a TLB or the result of a translation table walk. Therefore, ARM recommends that these instructions are not used at a time when the TLB entries might be different from the underlying translation tables held in memory.

When Non-secure EL1&0 stage 1 address translation is disabled, any AT S1E0*, AT S1E1*, AT S12E0*, or ATS12E1* address translation instruction that accesses the Non-secure state translation reflects the effect of the HCR_EL2.DC bit as described in *Behavior when stage 1 address translation is disabled* on page D4-1781.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Executing AT S1E2R or AT S1E2W at EL3 with SCR_EL3.NS==0 is UNDEFINED.

**Note**

AT S12E* instructions at EL3 with SCR_EL3.NS==0 are not UNDEFINED but behave the same way as the equivalent AT S1E* instructions.

### *Synchronous faults generated by address translation instructions*

The address translation instructions use the translation mechanism, and that mechanism can generate the following synchronous faults:

- Translation fault.
- Access flag fault.
- Permission fault.
- Domain fault, when translating using the AArch32 translation systems.
- Address size fault.
- TLB conflict fault.
- Synchronous external aborts during a translation table walk.

In addition:

- If the address translation instruction requires two stages of translation then these faults could arise from either stage 1 or stage 2.
- For a stage 1 translation for the Non-secure EL1&0 translation regime, the fault might be generated on the stage 2 translation of an address accessed as part of the stage 1 translation table walk, see *Stage 2 fault on a stage 1 translation table walk* on page D4-1830.

Except as described in this section, these faults are not taken as an exception for the address translation instructions, but instead the PAR_EL1.FST field holds the fault status information. In these cases the PAR_EL1.PA field does not hold the output address of the translation.

The exceptions to this reporting the fault in PAR_EL1 are:

- Synchronous external aborts during a translation table walk are taken as a Data Abort exception.

  For an address translation instruction executed at a particular Exception level, if the synchronous external abort is generated on a stage 1 translation table walk, the Data Abort exception is taken to the Exception level to which a synchronous external abort on a stage 1 translation table walk for a memory access from that Exception level would be taken.

  If the synchronous external abort is generated on a stage 2 translation table walk then:

  — If the address translation instruction was executed at EL3, the synchronous Data Abort exception is taken to EL3.

  — If the address translation instruction was executed at EL2 or EL1, the Data Abort exception is taken to the Exception level to which a synchronous external abort on a stage 2 translation table walk for a memory access from that Exception level would be taken.

  In any case where the address translation instruction causes a synchronous Data Abort exception to be taken:

  — The PAR_EL1 is UNKNOWN.

  — The ESR_ELx of the target Exception Level of the exception indicates that the fault was due to a translation table walk for a cache maintenance instruction.

  — The FAR_ELx of the target Exception Level holds the virtual address for the translation request.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For the `AT S1E0*` and `AT S1E1*` instructions executed from the Non-secure EL1 Exception level, if there is a synchronous stage 2 fault on a memory access made as part of the translation table walk then if the value of SCR_EL3.EA is 1 then a synchronous external abort on a stage 2 translation table walk is taken to EL3. In all other cases of a synchronous stage 2 fault on a memory access made as part of the translation table walk, the fault is taken as an exception to EL2, and:
  - PAR_EL1 is UNKNOWN
  - ESR_EL2 indicates that the fault occurred on a translation table walk, and that the operation that faulted was a cache maintenance instruction.
  - HPFAR_EL2 holds the IPA that faulted
  - FAR_EL2 holds the VA that the executing software supplied to the address translation instruction.

  This fault can occur for any of the following reasons:
  - Stage 2 Translation fault.
  - Stage 2 Access fault.
  - Stage 2 Permission fault.
  - Stage 2 Address size fault.
  - Synchronous external abort on a stage 2 translation table walk.

#### *Synchronization requirements of the address translation instructions*

Where an instruction results in an update to a System register, as is the case with the `AT *` address translation instructions, explicit synchronization must be performed before the result is guaranteed to be visible to subsequent direct reads of the PAR_EL1.

**Note**

This is consistent with the AArch32 requirement, where the VA to PA translation instructions are expressed as CP15 register writes, and the effect of those writes to other registers require explicit synchronization before the result is guaranteed to be visible to subsequent instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D4.3 VMSAv8-64 translation table format descriptors

In general, a descriptor is one of:

- An invalid or fault entry.
- A table entry, that points to the next-level translation table.
- A block entry, that defines the memory properties for the access.
- A reserved format.

Bit[1] of the descriptor indicates the descriptor type, and bit[0] indicates whether the descriptor is valid.

The following sections describe the ARMv8 translation table descriptor formats:

- *VMSAv8-64 translation table level 0, level 1, and level 2 descriptor formats*.
- *ARMv8 translation table level 3 descriptor formats* on page D4-1802.

*Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D4-1803 then gives more information about the descriptor attribute fields, and *Control of Secure or Non-secure memory access* on page D4-1806 describe how the NS and NSTable together control whether a memory access from Secure state accesses the Secure memory map or the Non-secure memory map.

### D4.3.1 VMSAv8-64 translation table level 0, level 1, and level 2 descriptor formats

In the VMSAv8-64 translation table format, the difference in the formats of the level 0, level 1 and level 2 descriptors is:

- Whether a block entry is permitted.
- If a block entry is permitted, the size of the memory region described by that entry.

These differences depend on the translation granule, as follows:

**4KB granule** A level 0 descriptor does not support block translation.

A block entry:

- In a level 1 table describes the mapping of the associated 1GB input address range.
- In a level 2 table describes the mapping of the associated 2MB input address range.

**16KB granule** Level 0 and level 1 descriptors do not support block translation.

A block entry in a level 2 table describes the mapping of the associated 32MB input address range.

**64KB granule** Level 0 lookup is not supported.

A level 1 descriptor does not support block translation.

A block entry in a level 2 table describes the mapping of the associated 512MB input address range.

Figure D4-16 on page D4-1800 shows the ARMv8 level 0, level 1, and level 2 descriptor formats:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

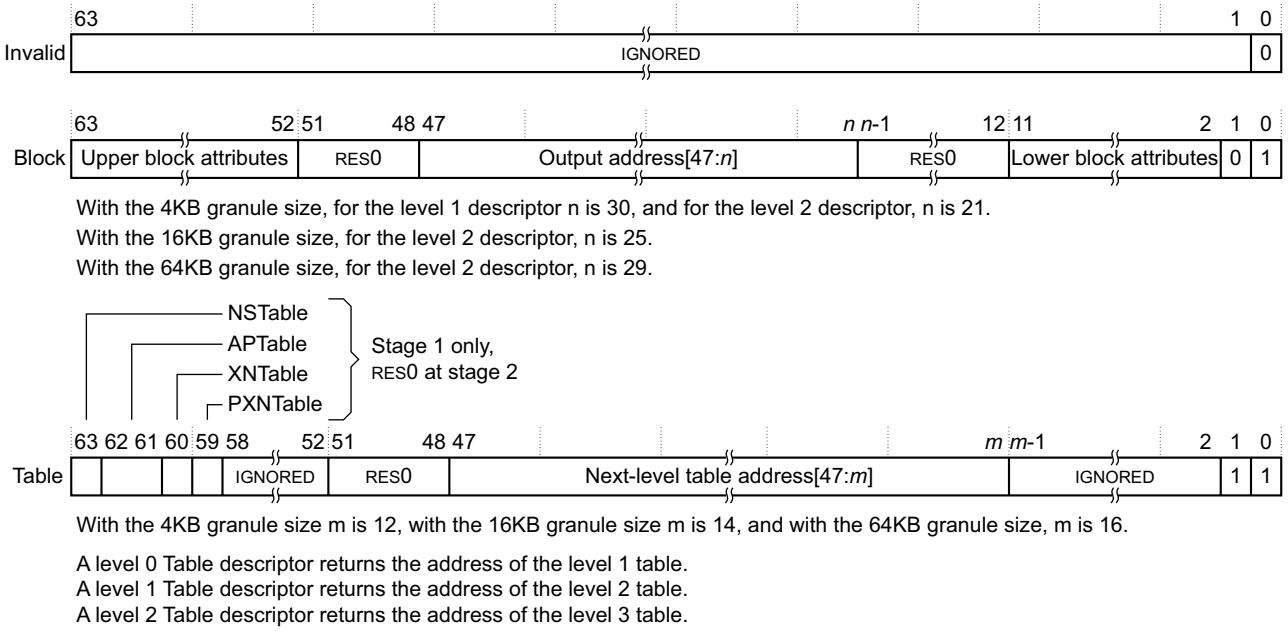



**Figure D4-16 VMSAv8-64 level 0, level 1, and level 2 descriptor formats**

## Descriptor encodings, ARMv8 level 0, level 1, and level 2 formats

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

**0, Block** The descriptor gives the base address of a block of memory, and the attributes for that memory region.

**1, Table** The descriptor gives the address of the next level of translation table, and for a stage 1 translation, some attributes for that translation.

The other fields in the valid descriptors are:

**Block descriptor**

Gives the base address and attributes of a block of memory, as follows:

**4KB translation granule**

- For a level 1 Block descriptor, bits[47:30] are bits[47:30] of the output address. This output address specifies a 1GB block of memory.
- For a level 2 descriptor, bits[47:21] are bits[47:21] of the output address.This output address specifies a 2MB block of memory.

**16KB translation granule**

For a level 2 Block descriptor, bits[47:25] are bits[47:25] of the output address.This output address specifies a 32MB block of memory.

**64KB translation granule**

For a level 2 Block descriptor, bits[47:29] are bits[47:29] of the output address.This output address specifies a 512MB block of memory.

Bits[63:52, 11:2] provide attributes for the target memory block, see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D4-1803. The position and contents of these bits are identical in the level 2 block descriptor and in the level 3 page descriptor.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table descriptor**

Gives the translation table address for the next-level lookup, as follows:

**4KB translation granule**

- Bits[47:12] are bits[47:12] of the address of the required next-level table, which is:
  - For a level 0 Table descriptor, the address of a level 1 table.
  - For a level 1Table descriptor, the address of a level 2 table.
  - For a level 2 Table descriptor, the address of a level 3 table.
- Bits[11:0] of the table address are zero.

**16KB translation granule**

- Bits[47:14] are bits[47:14] of the address of the required next-level table, which is:
  - For a level 0 Table descriptor, the address of a level 1 table.
  - For a level 1 Table descriptor, the address of a level 2 table.
  - For a level 2 Table descriptor, the address of a level 3 table.
- Bits[13:0] of the table address are zero.

**64KB translation granule**

- Bits[47:16] are bits[47:16] of the address of the required next-level table, which is:
  - For a level 1 Table descriptor, the address of a level 2 table.
  - For a level 2 Table descriptor, the address of a level 3 table.
- Bits[15:0] of the table address are zero.

For a stage 1 translation only, bits[63:59] provide attributes for the next-level lookup, see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D4-1803.

If the translation table defines the Non-secure EL1&0 stage 1 translations, then the output address in the descriptor is the IPA of the target block or table. Otherwise, it is the PA of the target block or table.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### D4.3.2 ARMv8 translation table level 3 descriptor formats

For the 4KB granule size, each entry in a level 3 table describes the mapping of the associated 4KB input address range.

For the 16KB granule size, each entry in a level 3 table describes the mapping of the associated 16KB input address range.

For the 64KB granule size, each entry in a level 3 table describes the mapping of the associated 64KB input address range.

Figure D4-17 shows the ARMv8 level 3 descriptor formats.

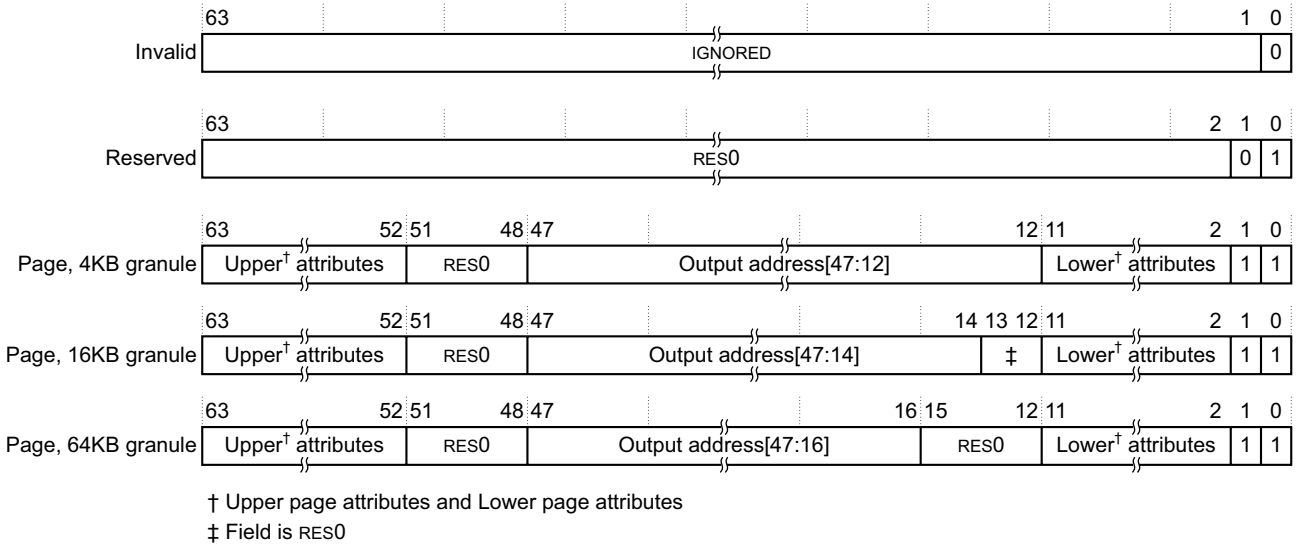



**Figure D4-17 VMSAv8-64 level 3 descriptor format**

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

**0, Reserved, invalid**

Behaves identically to encodings with bit[0] set to 0.

This encoding must not be used in level 3 translation tables.

**1, Page** Gives the address and attributes of a 4KB, 16KB, or 64KB page of memory.

At this level, the only valid format is the Page descriptor. The other fields in the Page descriptor are:

**Page descriptor**

Gives the output address of a page of memory, as follows:

**4KB translation granule**

Bits[47:12] are bits[47:12] of the output address for a page of memory.

**16KB translation granule**

Bits[47:14] are bits[47:14] of the output address for a page of memory.

**64KB translation granule**

Bits[47:16] are bits[47:16] of the output address for a page of memory.

Bits[63:52, 11:2] provide attributes for the target memory page, see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D4-1803.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Note**

The position and contents of bits[63:52, 11:2] are identical to bits[63:52, 11:2] in the level 0, level 1, and level 2 block descriptors.

For the Non-secure EL1&0 stage 1 translations, the output address in the descriptor is the IPA of the target page. Otherwise, it is the PA of the target page.

### D4.3.3 Memory attribute fields in the VMSAv8-64 translation table format descriptors

*Memory region attributes* on page D4-1816 describes the region attribute fields. The following subsections summarize the descriptor attributes as follows:

**Table descriptor**

Table descriptors for stage 2 translations do not include any attribute field. For a summary of the attribute fields in a stage 1 table descriptor, that define the attributes for the next lookup level, see *Next-level attributes in stage 1 VMSAv8-64 Table descriptors*.

**Block and page descriptors**

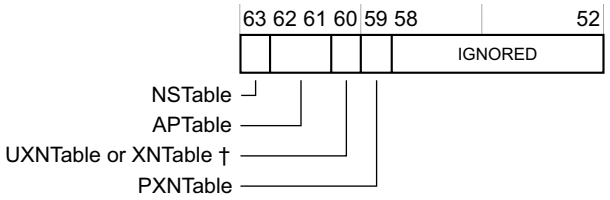
These descriptors define memory attributes for the target block or page of memory. Stage 1 and stage 2 translations have some differences in these attributes, see:

- *Attribute fields in stage 1 VMSAv8-64 Block and Page descriptors* on page D4-1804
- *Attribute fields in stage 2 VMSAv8-64 Block and Page descriptors* on page D4-1805.

#### Next-level attributes in stage 1 VMSAv8-64 Table descriptors

In a Table descriptor for a stage 1 translation, bits[63:59] of the descriptor define the attributes for the next-level translation table access, and bits[58:52] are IGNORED:

Next-level descriptor attributes, stage 1 only




† UXNTable for the EL1&0 translation regime, XNTable for the other regimes.

These attributes are:

**NSTable, bit[63]**

For memory accesses from Secure state, specifies the Security state for subsequent levels of lookup, see *Hierarchical control of Secure or Non-secure memory accesses* on page D4-1807.

For memory accesses from Non-secure state, this bit is RES0 and is ignored by the PE.

This field is RES1 in the AArch64 EL2 translation regime.

**APTable, bits[62:61]**

Access permissions limit for subsequent levels of lookup, see *Hierarchical control of data access permissions* on page D4-1810.

APTable[0] is RES0:

- In the EL2 translation regime.
- In the EL3 translation regime.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**UXNTable or XNTable, bit[60]**

XN limit for subsequent levels of lookup, see *Hierarchical control of instruction fetching* on page D4-1814.

This bit is called UXNTable in the EL1&0 translation regime, where it only determines whether execution at EL0 of instructions fetched from the region identified at a lower level of lookup permitted. In the other translation regimes the bit is called XNTable.

**PXNTable, bit[59]**

PXN limit for subsequent levels of lookup, see *Hierarchical control of instruction fetching* on page D4-1814.

This bit is reserved, SBZ:

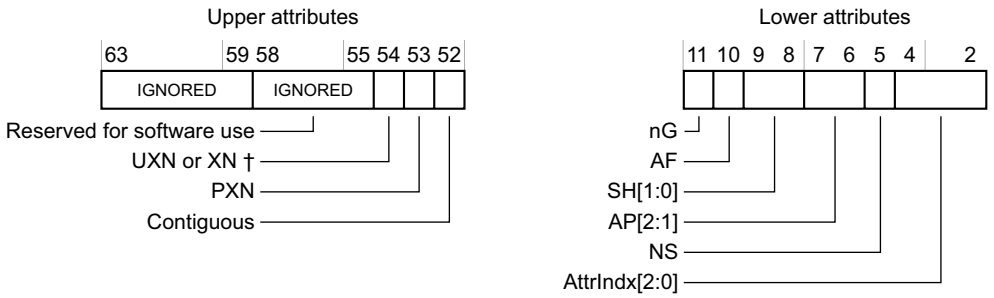
- In the EL2 translation regime.
- In the EL3 translation regime.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see *IGNORED* on page Glossary-5766. For more information about these fields see *Other fields in the VMSAv8-64 translation table format descriptors* on page D4-1819.

## Attribute fields in stage 1 VMSAv8-64 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block, as shown for a stage 1 translation:

Attribute fields for VMSAv8-64 stage 1 Block and Page descriptors




† UXN for the EL1&0 translation regime, XN for the other regimes.

For a stage 1 descriptor, the attributes are:

**UXN or XN, bit[54]**

The Execute-never bit. Determines whether the region is executable, see *Access permissions for instruction execution* on page D4-1811.

This bit is called UXN (Unprivileged execute never) in the EL1&0 translation regime, where it only determines whether execution at EL0 of instructions fetched from the region is permitted. In the other translation regimes the bit is called XN (Execute never).

**PXN, bit[53]** The Privileged execute-never bit. Determines whether the region is executable at EL1, see *Access permissions for instruction execution* on page D4-1811.

This bit is RES0 in the EL2 and EL3 translation regimes.

**Contiguous, bit[52]**

A hint bit indicating that the translation table entry is one of a contiguous set or entries, that might be cached in a single TLB entry, see *The Contiguous bit* on page D4-1819.

**nG, bit[11]** The not global bit. Determines whether the TLB entry applies to all ASID values, or only to the current ASID value, see *Global and process-specific translation table entries* on page D4-1834.

Valid only to the EL1&0 translation regime. This bit is RES0 in all other translation regimes.

**AF, bit[10]** The Access flag, see *The Access flag* on page D4-1815.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SH, bits[9:8]** Shareability field, see *Memory region attributes* on page D4-1816.

**AP[2:1], bits[7:6]**

Data Access Permissions bits, see *Memory access control* on page D4-1808.

**Note**

The ARMv8 translation table descriptor format defines AP[2:1] as the Access Permissions bits, and does not define an AP[0] bit.

AP[1] is valid only to the EL1&0 translation regime, and is RES1 in all other translation regimes.

**NS, bit[5]** Non-secure bit. For memory accesses from Secure state, specifies whether the output address is in the Secure or Non-secure address map, see *Control of Secure or Non-secure memory access* on page D4-1806.

For memory accesses from Non-secure state, this bit is RES0 and is ignored by the PE.

This field is RES1 in the AArch64 EL2 translation regime.

**AttrIndx[2:0], bits[4:2]**

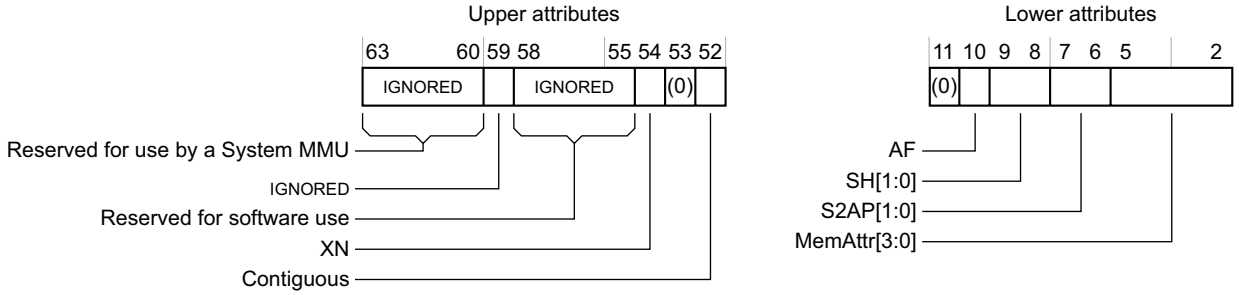
Stage 1 memory attributes index field, for the MAIR_ELx, see *Memory region type and attributes, for stage 1 translations* on page D4-1816.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see *IGNORED* on page Glossary-5766. For more information about these fields see *Other fields in the VMSAv8-64 translation table format descriptors* on page D4-1819.

## Attribute fields in stage 2 VMSAv8-64 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block, as shown for a stage 2 translation:

Attribute fields for VMSAv8-64 stage 2 Block and Page descriptors




For a stage 2 descriptor, the attributes are:

**XN, bit[54]** The Execute-never bit. Determines whether the region is executable, see *Access permissions for instruction execution* on page D4-1811.

**Contiguous, bit[52]**

A hint bit indicating that the translation table entry is one of a contiguous set or entries, that might be cached in a single TLB entry, see *The Contiguous bit* on page D4-1819.

**AF, bit[10]** The Access flag, see *The Access flag* on page D4-1815.

**SH, bits[9:8]** Shareability field, see *The memory region attributes for stage 2 translations, EL1&0 translation regime* on page D4-1817.

**S2AP, bits[7:6]**

Stage 2 data Access Permissions bits, see *The S2AP data access permissions, Non-secure EL1&0 translation regime* on page D4-1810.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

In the original VMSAv7-32 Long-descriptor attribute definition, this field was called HAP[2:1], for consistency with the AP[2:1] field in the stage 1 descriptors and despite there being no HAP[0] bit. ARMv8 renames the field for greater clarity.

**MemAttr, bits[5:2]**

Stage 2 memory attributes, see *The memory region attributes for stage 2 translations, EL1&0 translation regime* on page D4-1817.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see *IGNORED* on page Glossary-5766. For more information about these fields see *Other fields in the VMSAv8-64 translation table format descriptors* on page D4-1819.

### D4.3.4 Control of Secure or Non-secure memory access

As this section describes, the NS bit in the translation table entries:

- For accesses from Secure state, if the translation table entry was held in secure memory, determines whether the access is to Secure or Non-secure memory.
- Is ignored by:
  - Accesses from Non-secure state.
  - Accesses from Secure state if the translation table entry was held in Non-secure memory.

In the VMSAv8-64 translation table format:

- The NS bit relates only to the memory block or page at the output address defined by the descriptor.
- The descriptors also include an NSTable bit, that affects accesses at lower levels of lookup, see *Hierarchical control of Secure or Non-secure memory accesses* on page D4-1807.

The NS and NSTable bits are valid only for memory accesses from Secure state described by translation table descriptors that are fetched from Secure memory, and:

- In the translation table descriptors in a Non-secure translation table, the NS and NSTable bits are SBZ.
- Memory accesses from Non-secure state, including all accesses from EL2, ignore the values of these bits.

In the Secure translation regimes, for translation table descriptors that are fetched from Secure memory, the NS bit in a descriptor indicates whether the descriptor refers to the Secure or the Non-secure address map, as follows:

**NS == 0** Access the Secure physical address space.

**NS == 1** Access the Non-secure physical address space.

For Non-secure translation regimes, and for translation table descriptors fetched from Non-secure memory, the corresponding bit is RES0 and is ignored by the PE. The access is made to Non-secure memory, regardless of the value of the bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Hierarchical control of Secure or Non-secure memory accesses

For VMSAv8-64 table descriptors for stage 1 translations, the descriptor includes an NSTable bit, that indicates whether the table identified in the descriptor is in Secure or Non-secure memory. For accesses from Secure state, the meaning of the NSTable bit is:

**NSTable == 0** The defined table address is in the Secure physical address space. In the descriptors in that translation table, NS bits and NSTable bits have their defined meanings.

**NSTable == 1** The defined table address is in the Non-secure physical address space. Because this table is fetched from the Non-secure address space, the NS and NSTable bits in the descriptors in this table must be ignored. This means that, for this table:

- The value of the NS bit in any block or page descriptor is ignored. The block or page address refers to Non-secure memory.
- The value of the NSTable bit in any table descriptor is ignored, and the table address refers to Non-secure memory. When this table is accessed, the NS bit in any block or page descriptor is ignored, and all descriptors in the table refer to Non-secure memory.

In addition, an entry fetched in Secure state is treated as non-global if it is read from Non-secure memory. That is, these entries must be treated as if nG==1, regardless of the value of the nG bit. For more information about the nG bit, see *Global and process-specific translation table entries* on page D4-1834.

The effect of NSTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore a change to NSTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D4.4 Access controls and memory region attributes

In addition to an output address, a translation table entry that refers to a page or region of memory includes fields that define properties of the target memory region. These fields can be classified as address map control, access control, and region attribute fields. *Control of Secure or Non-secure memory access* on page D4-1806 describes the address map control, and the following sections describe the other fields:

- *Memory access control*.
- *Memory region attributes* on page D4-1816.
- *Combining the stage 1 and stage 2 attributes, Non-secure EL1&0 translation regime* on page D4-1821.

**Note**

This section describes the access controls and memory region attributes for each of the translation regimes, and for both stages of translation in the Non-secure EL1&0 translation regime. In general, attribute assignment is simpler in the EL2 and EL3 translation regimes, and in these regimes behavior is consistent with fields in the translation tables being treated as follows:

- APTable[0] is RES0 and is ignored by the PE, meaning it is treated as if it is 0.
- AP[1] is RES1 and is ignored by the PE, meaning it is treated as if it is 1.
- The PXNTable bit is RES0 and is ignored by the PE, meaning it is treated as if it is 0.
- The PXN field is RES0 and is ignored by the PE, meaning it is treated as if it is 0.

### D4.4.1 Memory access control

The access control fields in the translation table descriptors determine whether the PE, in its current state, is permitted to perform the required access to the output address given in the translation table descriptor. If a translation stage does not permit the access then an MMU fault is generated for that translation stage, and no memory access is performed.

The following sections describe the memory access controls:

- *About the access permissions*.
- *The data access permission controls* on page D4-1809.
- *Access permissions for instruction execution* on page D4-1811.
- *The Access flag* on page D4-1815.

#### About the access permissions

**Note**

This section gives a general description of memory access permissions. In an implementation that includes EL2, software executing at EL1 in Non-secure state can see only the access permissions defined by the Non-secure EL1&0 stage 1 translations. However, software executing at EL2 can modify these permissions. This modification is invisible to the Non-secure software executing at EL1 or EL0.

The access permission bits control access to the corresponding memory region. The VMSAv8-64 translation table format:

- In stage 1 translations, uses AP[2:1] to define the data access permissions, see *The AP[2:1] data access permissions, for stage 1 translations* on page D4-1809.

  **Note**

  The description of the access permission field as AP[2:1] is for consistency with the VMSAv8-32 Short-descriptor translation table format, see *The VMSAv8-32 Short-descriptor translation table format* on page G4-4009. The VMSAv8-64 translation table format does not define an AP[0] bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In stage 2 translations, uses S2AP[1:0] to define the data access permissions, see *The S2AP data access permissions, Non-secure EL1&0 translation regime* on page D4-1810.
- Uses the UXN, XN and PXN bits to define access controls for instruction fetches, see *Access permissions for instruction execution* on page D4-1811.

An attempt to perform a memory access that the translation table access permission bits do not permit generates a Permission fault, for the corresponding stage of translation.

**Note**

In an implementation that includes EL2, each stage of the translation of a memory access made from Non-secure EL1 or EL0 has its own, independent, permission check.

## The data access permission controls

The following subsubsections describe the data access permission controls:

- *The AP[2:1] data access permissions, for stage 1 translations*.
- *The S2AP data access permissions, Non-secure EL1&0 translation regime* on page D4-1810.
- *Hierarchical control of data access permissions* on page D4-1810.

### *The AP[2:1] data access permissions, for stage 1 translations*

For the VMSAv8-64 EL1&0 translation regime, the AP[2:1] bits control the stage 1 data access permissions, and:

**AP[2]** Selects between read-only and read/write access.

**AP[1]** Selects between Application level (EL0) and System level (EL1) control.

This provides four permission settings for data accesses:

- Read-only at all levels.
- Read/write at all levels.
- Read-only at EL1, no access by software executing at EL0.
- Read/write at EL1, no access by software executing at EL0.

For translation regimes other than the EL1&0 translation regimes, AP[2] determines the stage 1 data access permissions, and AP[1] is:

- SBO.
- Ignored by hardware and is treated as if it is 1.

Table D4-29 shows the effect of the data access permission bits for stage 1 of the EL1&0 translation regime. In this table, an entry of None indicates that any access from that Exception level faults.

**Table D4-29 Data access permissions for stage 1 of the EL1&0 translation regime,**

| AP[2:1] | Access from EL1 | Access from EL0 |
|---|---|---|
| 00 | Read/write | None |
| 01 | Read/write | Read/write |
| 10 | Read-only | None |
| 11 | Read-only | Read-only |

For the Non-secure EL1&0 translation regime:

- The stage 2 translation also defines data access permissions, see *The S2AP data access permissions, Non-secure EL1&0 translation regime* on page D4-1810.
- When both stages of translation are enabled, *Combining the stage 1 and stage 2 data access permissions* on page D4-1821 describes how these permissions are combined.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table D4-30 shows the effect of the AP[2] data access permission bit for the EL2 and EL3 translation regimes.

**Table D4-30 Data access permissions for the EL2 or EL3 translation regime**

| AP[2] | Access from EL2 or EL3 |
|---|---|
| 0 | Read/write |
| 1 | Read-only |

### *The S2AP data access permissions, Non-secure EL1&0 translation regime*

In the Non-secure EL1&0 translation regime, when stage 2 address translation is enabled, the S2AP field in the stage 2 translation table descriptors define the data access permissions as Table D4-31 shows. In this table, an entry of None indicates that any access generates a permission fault.

**Table D4-31 Data access permissions for stage 2 of the Non-secure EL1&0 translation regime**

| S2AP | Access from Non-secure EL1 or Non-secure EL0 |
|---|---|
| 00 | None |
| 01 | Read-only |
| 10 | Write-only |
| 11 | Read/write |

The S2AP access permissions make no distinction between Non-secure accesses from EL1 and Non-secure accesses from EL0. However, when both stages of address translation are enabled, these permissions are combined with the stage 1 access permissions defined by AP[2:1], see *Combining the stage 1 and stage 2 data access permissions* on page D4-1821.

*Combining the stage 1 and stage 2 attributes, Non-secure EL1&0 translation regime* on page D4-1821 gives more information about the use of the stage 1 and stage 2 access permissions in an implementation of virtualization.

### *Hierarchical control of data access permissions*

The VMSAv8-64 translation table format includes mechanisms by which entries at one level of translation table lookup can set limits on the permitted entries at subsequent levels of lookup. This subsection describes how these controls apply to the data access permissions.

**Note**

Similar hierarchical controls apply to instruction fetching, see *Hierarchical control of instruction fetching* on page D4-1814.

The restrictions apply only to subsequent levels of lookup for the same stage of translation. The APTable[1:0] field restricts the access permissions, as Table D4-32 shows.

As stated in the table footnote, for the EL2 translation regime, APTable[0] is reserved, SBZ, and is ignored by the hardware.

**Table D4-32 Effect of APTable[1:0] on subsequent levels of lookup**

| APTable[1:0] | Effect |
|---|---|
| 00 | No effect on permissions in subsequent levels of lookup. |
| 01 [a] | Access at EL0 not permitted, regardless of permissions in subsequent levels of lookup. |
| 10 | Write access not permitted, at any Exception level, regardless of permissions in subsequent levels of lookup. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D4-32 Effect of APTable[1:0] on subsequent levels of lookup (continued)**

| APTable[1:0] | Effect |
|---|---|
| 11 [a] | Regardless of permissions in subsequent levels of lookup:<br>• Write access not permitted, at any Exception level.<br>• Read access not permitted at EL0. |

a. Not valid for the EL2 and EL3 translation regime. In the translation tables for that regime, APTable[0] is SBZ and is ignored by hardware.

**Note**

The APTable[1:0] settings are combined with the translation table access permissions in the translation tables descriptors accessed in subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The VMSAv8-64 provides APTable[1:0] control only for the stage 1 translations. The corresponding bits are SBZ in the stage 2 translation table descriptors.

The effect of APTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to APTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

### Access permissions for instruction execution

Execute-never (XN) controls determine whether instructions can be executed from a memory region. These controls are:

**UXN, Unprivileged Execute never**

Defined only for stage 1 of the EL1&0 translation regime.

**PXN, Privileged execute never**

Used only for stage 1 of the EL1&0 translation regime:

- For the EL2 and EL3 translation regimes, the descriptors define a PXN bit that is reserved, SBZ, and is ignored by hardware.
- For stage 2 of the Non-secure EL1&0 translation regime, the corresponding bit position is reserved, SBZ, and is ignored by hardware.

**XN, Execute never**

Defined for stage 2 of the EL1&0 translation regime and for stage 1 of the EL2 and EL3 translation regimes.

Each of theses bits is set to 1 to indicate that instructions cannot be executed from the target memory region. In addition:

- For the EL1&0 translation regime, if the value of the AP[2:1] bits is `0b01`, permitting write access from EL0, then the PXN bit is treated as if it has the value 1, regardless of its actual value.
- For each translation regime, if the value of the corresponding SCTLR_ELx.WXN bit is 1 then any memory region that is writable is treated as XN, regardless of the value of the corresponding UXN, XN, or PXN bit. For more information see *Preventing execution from writable locations* on page D4-1815.
- The SCR_EL3.SIF bit prevents execution in Secure state of any instruction fetched from Non-secure memory, see *Restriction on Secure instruction fetch* on page D4-1815.

The execute-never controls apply to speculative instruction fetching, meaning speculative instruction fetch from a memory region that is execute-never at the current Exception level is prohibited.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- Although the execute-never controls apply to speculative fetching, on a speculative instruction fetch from an execute-never location, no Permission fault is generated unless the PE attempts to execute the instruction that would have been fetched from that location. This means that, if a speculative fetch from an execute-never location is attempted, but there is no attempt to execute the corresponding instruction, a Permission fault is not generated.
- The software that defines a translation table must mark any region of memory that is read-sensitive as execute-never, to avoid the possibility of a speculative fetch accessing the memory region. This means it must mark any memory region that corresponds to a read-sensitive peripheral as execute-never. Hardware does not prevent speculative accesses to a region of any Device memory type unless that region is also marked as execute-never for all Exception levels from which it can be accessed.
- When no stage of address translation for the translation regime is enabled, memory regions cannot have UXN, XN, or PXN attributes assigned. *Behavior of instruction fetches when all associated stages of translation are disabled* on page D4-1783 describes how disabling all stages of address translation affects instruction fetching.

The following subsubsections describe the data access permission controls:

- *Instruction access and execution permissions for stage 1 translations*.
- *Instruction execution permissions for stage 2 translations* on page D4-1814.
- *Hierarchical control of instruction fetching* on page D4-1814.
- *Preventing execution from writable locations* on page D4-1815.
- *Restriction on Secure instruction fetch* on page D4-1815.

### *Instruction access and execution permissions for stage 1 translations*

Table D4-33 and Table D4-34 on page D4-1813 include the AP[2:1] read and write permissions shown in Table D4-29 on page D4-1809 and Table D4-30 on page D4-1810. These permissions are shown as:

**R** Indicates Read permission granted.

**W** Indicates Write permission granted.

Table D4-33 shows the access permissions for instruction execution for stage 1 of the EL1&0 translation regime.

**Table D4-33 Access permissions for instruction execution for stage 1 of the EL1&0 translation regime**

| UXN | PXN | AP[2:1] | SCTLR_EL1.WXN | Access from EL1 | Access from EL0 |
|---|---|---|---|---|---|
| 0 | 0 | 00 | 0 | R, W, Executable | Executable |
| | | | 1 | R, W, Not executable[a] | Executable |
| | | 01 | 0 | R, W, Not executable[b] | R, W, Executable |
| | | | 1 | R, W, Not executable | R, W, Not executable[c] |
| | | 10 | x | R, Executable | Executable |
| | | 11 | x | R, Executable | R, Executable |
| 0 | 1 | 00 | x | R, W, Not executable | Executable |
| | | 01 | 0 | R, W, Not executable | R, W, Executable |
| | | | 1 | R, W, Not executable | R, W, Not executable[c] |
| | | 10 | x | R, Not executable | Executable |
| | | 11 | x | R, Not executable | R, Executable |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D4-33 Access permissions for instruction execution for stage 1 of the EL1&0 translation regime (continued)**

| UXN | PXN | AP[2:1] | SCTLR_EL1.WXN | Access from EL1 | Access from EL0 |
|---|---|---|---|---|---|
| 1 | 0 | 00 | 0 | R, W, Executable | Not executable |
| | | | 1 | R, W, Not executable[a] | Not executable |
| | | 01 | x | R, W, Not executable[b] | R, W, Not executable |
| | | 10 | x | R, Executable | Not executable |
| | | 11 | x | R, Executable | R, Not executable |
| | 1 | 00 | x | R, W, Not executable | Not executable |
| | | 01 | x | R, W, Not executable | R, W, Not executable |
| | | 10 | x | R, Not executable | Not executable |
| | | 11 | x | R, Not executable | R, Not executable |

a. Not executable because of SCTLR_EL1.WXN control, because region is writable at EL1.
b. Not executable, because AArch64 execution treats all regions writable at EL0 as being PXN.
c. Not executable because of SCTLR_EL1.WXN control, because region is writable at EL0.

Table D4-34 shows the access permissions for instruction execution for the EL2 and EL3 translation regimes.

**Table D4-34 Access permissions for instruction execution, EL2 and EL3 translation regimes**

| XN | AP[2] | SCTLR_EL2.WXN or[a] SCTLR_EL3.WXN | Access from EL2 or EL3 |
|---|---|---|---|
| 0 | 0 | 0 | R, W, Executable |
| | | 1 | R, W, Not executable[b] |
| | 1 | x | R, Executable |
| 1 | 0 | x | R, W, Not executable |
| | 1 | x | R, Not executable |

a. SCTLR_EL2 for the EL2 translation regime, SCTLR_EL3 for the EL3 translation regime.
b. Not executable because of the SCTLR_ELx.WXN control, because region is writable at this Exception level.

**Note**

The Access permissions for the AArch64 EL2 and EL3 translation regimes are consistent with the following fields in the translation table entries being treated as shown:

- AP[1] treated as RES1.
- APTable treated as RES0.
- PXN treated as RES0.
- PXNTable treated as RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Instruction execution permissions for stage 2 translations

For the Non-secure EL1&0 stage 2 translation, the XN bit in the stage 2 translation table descriptors controls the execution permission, and this control is completely independent of the S2AP access permissions.

**Table D4-35 Access permissions for instruction execution for stage 2 of the Non-secure EL1&0 translation regime,**

| XN | Access from Non-secure EL1or Non-secure EL0 |
|---|---|
| 0 | Executable |
| 1 | Not executable |

The stage 2 XN access permissions make no distinction between Non-secure accesses from EL1 and Non-secure accesses from EL0. However, when both stages of address translation are enabled, these permissions are combined with the stage 1 access permissions defined at stage 1 of the translation, see *Combining the stage 1 and stage 2 instruction execution permissions* on page D4-1821.

### Hierarchical control of instruction fetching

The VMSAv8-64 translation table format includes mechanisms by which entries at one level of translation table lookup can set limits on the permitted entries at subsequent levels of lookup. This subsection describes how these controls apply to the instruction fetching controls.

**Note**

Similar hierarchical controls apply to data accesses, see *Hierarchical control of data access permissions* on page D4-1810.

The restrictions apply only to subsequent levels of lookup at the same stage of translation, and:

- UXNTable or XNTable restricts the XN control:
  - When the value of the XNTable bit is 1, the XN bit is treated as 1 in all subsequent levels of lookup, regardless of its actual value.
  - When the value of the UXNTable bit is 1, the UXN bit is treated as 1 in all subsequent levels of lookup, regardless of its actual value.
  - When the value of a UXNTable or XNTable bit is 0 the bit has no effect.
- For the EL1&0 translation regime, PXNTable restricts the PXN control:
  - When PXNTable is set to 1, the PXN bit is treated as 1 in all subsequent levels of lookup, regardless of the actual value of the bit.
  - When PXNTable is set to 0 it has no effect.

**Note**

The UXNTable, XNTable, and PXNTable settings are combined with the UXN, XN, and PXN bits in the translation table descriptors accessed at subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The UXNTable, XNTable, and PXNTable controls are provided only for stage 1 translations. The corresponding bits are SBZ in the stage 2 translation table descriptors.

The effect of UXNTable, XNTable, or PXNTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to UXNTable, XNTable, or PXNTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Preventing execution from writable locations*

ARMv8 provides control bits that, when corresponding stage 1 address translation is enabled, force writable memory to be treated as UXN, PXN, or XN, regardless of the value of the UXN, PXN, or XN bit:

- For the EL1&0 translation regime, when the value of SCTLR_EL1.WXN is 1:
  - All regions that are writable from EL0 at stage 1 of the address translation are treated as UXN.
  - All regions that are writable from EL1 at stage 1 of the address translation are treated as PXN
- For the EL2 translation regime, when the value of SCTLR_EL2.WXN is 1, all regions that are writable at stage 1 of the address translation are treated as XN.
- For the EL3 translation regime, when the value of SCTLR_EL3.WXN is 1, all regions that are writable at stage 1 of the address translation are treated as XN.

**Note**

- The SCTLR_ELx.WXN controls are intended to be used in systems with very high security requirements.
- Setting a WXN bit to 1 changes the interpretation of the translation table entry, overriding a zero value of a UXN, XN, or PXN field. It does not cause any change to the translation table entry.

For any given virtual machine, ARM expects WXN to remain static in normal operation. In particular, it is IMPLEMENTATION DEFINED whether TLB entries associated with a particular VMID reflect the effect of the values of these bits. This means that any change of these bits without a corresponding change of VMID might require synchronization and TLB invalidation, as described in *TLB maintenance requirements and the TLB maintenance instructions* on page D4-1837.

#### *Restriction on Secure instruction fetch*

EL3 provides a Secure instruction fetch bit, SCR_EL3.SIF. When the value of this bit is 1, and execution is using the EL3 translation regime or the Secure EL1 translation regime, any attempt to execute an instruction fetched from Non-secure physical memory causes a Permission fault. TLB entries might reflect the value of this bit, and therefore any change to the value of this bit requires synchronization and TLB invalidation, as described in *TLB maintenance requirements and the TLB maintenance instructions* on page D4-1837.

### The Access flag

The Access flag indicates when a page or section of memory is accessed for the first time since the Access flag in the corresponding translation table descriptor was set to 0.

The AF bit in the translation table descriptors is the Access flag.

#### *Software management of the Access flag*

ARMv8 requires that software manages the Access flag. This means an Access flag fault is generated whenever an attempt is made to read into the TLB a translation table descriptor entry for which the value of Access flag is 0.

The Access flag mechanism expects that, when an Access flag fault occurs, software resets the Access flag to 1 in the translation table entry that caused the fault. This prevents the fault occurring the next time that memory location is accessed. Entries with the Access flag set to 0 are never held in the TLB, meaning software does not have to flush the entry from the TLB after setting the flag.

**Note**

If a system incorporates components that can autonomously update translation table entries that are shared with the ARM PE, then the software must be aware of the possibility that such components can update the access flag autonomously.

In such a system, system software should perform any changes of translation table entries with an Access flag of 0, other than changes to the Access flag value, by using an Load-Exclusive/Store-Exclusive loop, to allow for the possibility of simultaneous updates.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D4.4.2 Memory region attributes

The memory region attribute fields control the memory type, accesses to the caches, and whether the memory region is Shareable and therefore is coherent. This section also describes some additional translation table fields, that this manual groups with the memory region attributes.

In the EL1&0 translation regime, each enabled stage of address translation assigns memory region attributes, as described in this section. When both stages of translation are enabled, *Combining the stage 1 and stage 2 attributes, Non-secure EL1&0 translation regime* on page D4-1821 describes how the assignments from the two stages are combined.

**Note**

In a virtualization implementation, a hypervisor, executing at EL2, might usefully:

- Reduce the permitted cacheability of a region.
- Increase the required shareability of a region.

The combining of attributes from stage 1 and stage 2 translations supports both of these options.

The following sections describe these attributes:

- *The memory region attributes for stage 1 translations*.
- *The memory region attributes for stage 2 translations, EL1&0 translation regime* on page D4-1817.
- *Other fields in the VMSAv8-64 translation table format descriptors* on page D4-1819.

#### The memory region attributes for stage 1 translations

The description of the memory region attributes in a translation descriptor divides into:

**Memory type and attributes**

These are described indirectly, by registers referenced by bits in the table descriptor. This is described as *remapping* the memory type and attribute description. *Memory region type and attributes, for stage 1 translations* describes this encoding.

**Shareability** The SH[1:0] field in the translation table descriptor encodes shareability information. *Shareability for Normal memory, for stage 1 translations* on page D4-1817 describes this encoding.

##### *Memory region type and attributes, for stage 1 translations*

In the VMSAv8-64 translation table format, the AttrIndx[2:0] field in a block or page translation table descriptor for a stage 1 translation indicates the 8-bit field in the MAIR_ELx that specifies the attributes for the corresponding memory region. The required field is Attr*n*, where *n* = AttrIndx[2:0]. For more information about AttrIndx[2:0] see *Attribute fields in stage 1 VMSAv8-64 Block and Page descriptors* on page D4-1804

**Note**

Each MAIR_ELx is a 64-bit register that is architecturally mapped to a pair of AArch32 registers. See the MAIR_ELx register descriptions for more information.

Each MAIR_ELx.Attr*n* field defines, for the corresponding memory region:

- The memory type, Device or Normal.
- For Device memory, the Device memory type, one of:
  - Device-nGnRnE.
  - Device-nGnRE.
  - Device-nGRE.
  - Device-GRE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For Normal memory:
  - The inner and outer cacheability, Non-cacheable, Write-Through, or Write-Back
  - For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate policy hints, each of which is *Allocate* or *Do not allocate*, and the Transient allocation hints.

For more information about the memory type and attributes, see *Memory types and attributes* on page B2-93.

#### *Shareability for Normal memory, for stage 1 translations*

When using the VMSAv8-64 translation table format, the SH[1:0] field in a block or page translation table descriptor specifies the Shareability attributes of the corresponding memory region. Table D4-36 shows the encoding of this field.

**Table D4-36 SH[1:0] field encoding for Normal memory, VMSAv8-64 translation table format**

| SH[1:0] | Normal memory |
|---|---|
| 00 | Non-shareable |
| 01 | Reserved, CONSTRAINED UNPREDICTABLE[a] |
| 10 | Outer Shareable |
| 11 | Inner Shareable |

a. See *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501 for the permitted CONSTRAINED UNPREDICTABLE behavior.

**Note**

The shareability field is only relevant if the memory is a Normal Cacheable memory type. All Device and Normal Non-cacheable memory regions are always treated as Outer Shareable, regardless of the translation table shareability attributes

See *Combining the stage 1 and stage 2 shareability attributes for Normal memory* on page D4-1823 for constraints on the Shareability attributes of a Normal memory region that is Inner Non-cacheable, Outer Non-cacheable.

### The memory region attributes for stage 2 translations, EL1&0 translation regime

In the stage 2 translation table descriptors for memory regions and pages, the MemAttr[3:0] and SH[1:0] fields describe the stage 2 memory region attributes:

- *Memory region type and attributes for stage 2 translations* on page D4-1818 describes how the MemAttr[3:0] field defines these attributes.
- The SH[1:0] field in the translation table descriptor encodes shareability information. *Shareability for Normal memory, for stage 2 translations* on page D4-1819 describes this encoding.

The following sections describe how, when both stages of address translation are enabled, the memory region attributes assigned at stage 2 of the translation are combined with those assigned at stage 1:

- *Combining the stage 1 and stage 2 memory type attributes* on page D4-1822.
- *Combining the stage 1 and stage 2 cacheability attributes for Normal memory* on page D4-1822.
- *Combining the stage 1 and stage 2 shareability attributes for Normal memory* on page D4-1823.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Memory region type and attributes for stage 2 translations*

Table D4-37 shows how MemAttr[3:2] gives a top-level definition of the memory type, and of the outer cacheability of a Normal memory region.

**Table D4-37 VMSAv8-64 MemAttr[3:2] encoding, stage 2 translation**

| MemAttr[3:2] | Memory type | Outer cacheability |
|---|---|---|
| 00 | Device. MemAttr[1:0] encodes the Device memory type. | Not applicable |
| 01 | Normal. MemAttr[1:0] encodes the Inner Cacheability. | Outer Non-cacheable |
| 10 | | Outer Write-Through Cacheable |
| 11 | | Outer Write-Back Cacheable |

The encoding of MemAttr[1:0] depends on the Memory type indicated by MemAttr[3:2]:

- When MemAttr[3:2]==`0b00`, indicating Device memory, Table D4-38 shows the encoding of MemAttr[1:0].

**Table D4-38 MemAttr[1:0] encoding for Device memory**

| MemAttr[1:0] | Meaning when MemAttr[3:2] == `0b00` |
|---|---|
| 00 | Region is Device-nGnRnE memory |
| 01 | Region is Device-nGnRE memory |
| 10 | Region is Device-nGRE memory |
| 11 | Region is Device-GRE memory |

- When MemAttr[3:2]!=`0b00`, indicating Normal memory, Table D4-39 shows the encoding of MemAttr[1:0].

**Table D4-39 MemAttr[1:0] encoding for Normal memory**

| MemAttr[1:0] | Meaning when MemAttr[3:2] != `0b00` |
|---|---|
| 00 | Reserved, CONSTRAINED UNPREDICTABLE[a] |
| 01 | Inner Non-cacheable |
| 10 | Inner Write-Through Cacheable |
| 11 | Inner Write-Back Cacheable |

a. See *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501 for the permitted CONSTRAINED UNPREDICTABLE behavior.

**Note**

The stage 2 translation does not assign any allocation hints.

**Note**

The following stage 2 translation table attribute settings leave the stage 1 settings unchanged:

- MemAttr[3:2] == `0b11`, Normal memory, Outer Write-Back Cacheable.
- MemAttr[1:0] == `0b11`, Inner Write-Back Cacheable.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Shareability for Normal memory, for stage 2 translations*

When using the VMSAv8-64 translation table format, the SH[1:0] field in a block or page translation table descriptor specifies the Shareability attributes of the corresponding memory region. Table D4-40 shows the encoding of this field.

**Table D4-40 SH[1:0] field encoding for Normal memory, VMSAv8-64 translation table format**

| SH[1:0] | Normal memory |
|---|---|
| 00 | Non-shareable |
| 01 | Reserved, CONSTRAINED UNPREDICTABLE[a] |
| 10 | Outer Shareable |
| 11 | Inner Shareable |

a. See *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501 for the permitted CONSTRAINED UNPREDICTABLE behavior.

**Note**

- This encoding is the same as the shareability encoding described in *Shareability for Normal memory, for stage 1 translations* on page D4-1817.
- The shareability field is only relevant if the memory is a Normal Cacheable memory type. All Device and Normal Non-cacheable memory regions are always treated as Outer Shareable, regardless of the translation table shareability attributes.

See *Combining the stage 1 and stage 2 shareability attributes for Normal memory* on page D4-1823 for constraints on the Shareability attributes of a Normal memory region that is Inner Non-cacheable, Outer Non-cacheable.

### Other fields in the VMSAv8-64 translation table format descriptors

The following subsections describe the other fields in the translation table block and page descriptors:

- *The Contiguous bit*.
- *IGNORED fields* on page D4-1820.
- *Field reserved for software use* on page D4-1820.

#### *The Contiguous bit*

When the value of the Contiguous bit is 1, it indicates that the entry is one of a number of adjacent translation table entries that point to a *contiguous output address range*. The required number of adjacent entries depends on the current translation granule size, as follows:

**4KB granule** 16 adjacent translation table entries point to a contiguous output address range that has the same permissions and attributes. These 16 entries must be aligned in the translation table. If accessing a full-sized 4KB translation table, this means that the top 5 of the 9 input addresses bits that index the descriptor positions in the translation table are the same for all of the entries.

The contiguous output address range must be aligned to size of 16 translation table entries at the same translation table level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**16KB granule** This bit indicates that adjacent translation table entries point to contiguous output address range that has the same permissions and attributes. With the 16KB granule, the number of contiguous entries indicated by setting this bit to 1 depends on the lookup level of the translation table:

**Level 2 lookup** The bit indicates 32 contiguous entries, giving a 1GB block of memory. These entries must be aligned in the translation table. When accessing a full-sized 16KB translation table, this means the top 6 of the 11 input addresses bits that index the descriptor positions in the translation table are the same for all of the entries.

The contiguous output address range must be aligned to size of 32 translation table entries at the same translation table level.

**Level 3 lookup** The bit indicates 128 contiguous entries, giving a 2MB block of memory. These entries must be aligned in the translation table. When accessing a full-sized 16KB translation table, this means the top 4 of the 11 input addresses bits that index the descriptor positions in the translation table are the same for all of the entries.

The contiguous output address range must be aligned to size of 128 translation table entries at the same translation table level.

**64KB granule** 32 adjacent translation table entries point to a contiguous output address range that has the same permissions and attributes. These 32 entries must be aligned in the translation table. If accessing a full-sized 64KB translation table, this means that the top 8 of the 13 input addresses bits that index the descriptor positions in the translation table are the same for all of the entries.

The contiguous output address range must be aligned to size of 32 translation table entries at the same translation table level.

Setting this bit to 1 means that the TLB can cache a single entry to cover the contiguous translation table entries.

This section defines the requirements for programming the Contiguous bit. *Possible translation table registers programming errors* on page D4-1777 describes the effect of not meeting these requirements.

The architecture does not require a PE to cache TLB entries in this way. To avoid TLB coherency issues, any TLB maintenance by address must not assume any optimization of the TLB tables that might result from use of the Contiguous bit.

TLB maintenance must be performed based on the size of the underlying translation table entries, to avoid TLB coherency issues.

### *IGNORED fields*

In the VMSAv8-64 translation table descriptors, the following fields are identified as IGNORED, meaning the architecture guarantees that a PE makes no use of these fields:

- In the stage 1 table descriptors, bits[58:52].
- In the stage 1 and stage 2 block and page descriptors, bits[63:59] and bits[58:55].

Of these fields:

- In the stage 1 and stage 2 block and page descriptors, bits[58:55] are reserved for software use, see *Field reserved for software use*.
- In the stage 2 block and page descriptors, bits[63:60] are reserved for use by a System MMU control.

### *Field reserved for software use*

The architecture reserves a 4-bit IGNORED field in the Block and Page table descriptors, bits[58:52], for software use. The definition of IGNORED means the architecture guarantees that hardware makes no use of this field.

**Note**

This means there is no need to invalidate the TLB if these bits are changed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D4.4.3 Combining the stage 1 and stage 2 attributes, Non-secure EL1&0 translation regime

The Non-secure EL1&0 translation regime comprises two stage of translation, each of which can be enabled independently:

- Stage 1 translation is configured and controlled from EL1. When enabled, stage 1 translation can define access permissions independently for access from EL0 and for accesses from EL1.

  Stage 1 MMU faults are taken to EL1.

- When stage 2 translation is enabled, the stage 2 access controls defined at EL2:
  - Affect only the Non-secure stage 1 access permissions settings.
  - Take no account of whether the accesses are at EL1 or EL0.
  - Permit software executing at EL2 to assign a write-only attribute to a memory region.

  Stage 2 MMU faults are taken to EL2.

**Note**

In an implementation of virtualization, the attributes defined in the stage 2 translation tables mean a hypervisor can define additional access restrictions to those defined by a Guest OS in the stage 1 translation tables. For a particular access, the actual access permission is the more restrictive of the permissions defined by:

- The Guest OS, in the stage 1 translation tables.
- The hypervisor, in the stage 2 translation tables.

The effects of the combination of attributes defined by the Hypervisor are functionally transparent to the Guest OS.

#### Combining the stage 1 and stage 2 data access permissions

When both stages of translation are enabled, the following access permissions are combined:

- The stage 1 permissions described in *The AP[2:1] data access permissions, for stage 1 translations* on page D4-1809.
- The stage 2 permissions described in *The S2AP data access permissions, Non-secure EL1&0 translation regime* on page D4-1810.

The stage 1 and stage 2 permissions are combined as follows:

1. If an access is not permitted by the stage 1 permissions, then it generates a Stage 1 Permission fault, regardless of the stage 2 permissions.
2. If an access is permitted by the stage 1 permissions, but is not permitted by the stage 2 Permissions, then it generates a Stage 2 Permission fault.
3. If an access is permitted by both the stage 1 permissions and the stage 2 permissions, then it does not generate a Permission fault.

#### Combining the stage 1 and stage 2 instruction execution permissions

When both stages of translation are enabled, the following access permissions are combined:

- The stage 1 permissions described in *Instruction access and execution permissions for stage 1 translations* on page D4-1812.
- The stage 2 permissions described in *Instruction execution permissions for stage 2 translations* on page D4-1814.

The stage 1 and stage 2 permissions are combined as follows:

1. If an instruction fetch is not permitted by the stage 1 permissions, then it generates a Stage 1 Permission fault, regardless of the stage 2 permissions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2. If an instruction fetch is permitted by the stage 1 permissions, but is not permitted by the stage 2 Permissions, then it generates a Stage 2 Permission fault.

3. If an instruction fetch is permitted by both the stage 1 permissions and the stage 2 permissions, then it does not generate a Permission fault.

### Combining the stage 1 and stage 2 memory type attributes

Table D4-41 shows the rules for combining the stage 1 and stage 2 memory type assignments.

**Table D4-41 Combining the stage 1 and stage 2 memory type assignments**

| Rule | If either stage of translation assigns: | The resultant memory type is: |
|---|---|---|
| Device has precedence over Normal | Any Device memory type | A Device memory type |
| Non-Gathering has precedence over Gathering | A Device-nGxx memory type | A Device-nGxx memory type |
| Non-Reordering has precedence over Reordering | A Device-nGnRx memory type | A Device-nGnRx memory type |
| No Early write acknowledge has precedence over Early write acknowledge | The Device-nGnRnE memory type | The Device-nGnRnE memory type |

Regardless of any shareability attribute obtained as described in *Combining the stage 1 and stage 2 shareability attributes for Normal memory* on page D4-1823:

- Any location for which the resultant memory type is any type of Device memory is always treated as Outer Shareable.
- Any location for which the resultant memory type is Normal Inner Non-cacheable, Outer Non-cacheable is always treated as Outer Shareable.

  For information about how the cacheability attribute is obtained from the attributes assigned at each stage of translation see *Combining the stage 1 and stage 2 cacheability attributes for Normal memory*.

The combining of the memory type attributes from the two stages of translation means a translation table walk for stage 1 translation can be made to a type of Device memory. If this occurs then:

- If the value of HCR_EL2.PTW is 0, then the translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be done speculatively.
- If the value of HCR_EL2.PTW is 1, then the memory access generates a stage 2 Permission fault.

### Combining the stage 1 and stage 2 cacheability attributes for Normal memory

For a Normal memory region, Table D4-42 shows how the stage 1 and stage 2 cacheability assignments are combined. This combination applies, independently, for the Inner cacheability and Outer cacheability attributes.

**Table D4-42 Combining the stage 1 and stage 2 cacheability assignments for Normal memory**

| Assignment in stage 1 | Assignment in stage 2 | Resultant cacheability |
|---|---|---|
| Non-cacheable | Any | Non-cacheable |
| Any | Non-cacheable | Non-cacheable |
| Write-Through Cacheable | Write-Through or Write-Back Cacheable | Write-Through Cacheable |
| Write-Through or Write-Back Cacheable | Write-Through Cacheable | Write-Through Cacheable |
| Write-Back Cacheable | Write-Back Cacheable | Write-Back Cacheable |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Combining the stage 1 and stage 2 shareability attributes for Normal memory

A memory region is treated as Outer Shareable, regardless of any shareability assignments at either stage of translation, if either:

- The resultant memory type attribute, described in *Combining the stage 1 and stage 2 memory type attributes* on page D4-1822, is any type of Device memory.
- The resultant memory type attribute, described in *Combining the stage 1 and stage 2 memory type attributes* on page D4-1822, is Normal memory, and the resultant cacheability, described in *Combining the stage 1 and stage 2 cacheability attributes for Normal memory* on page D4-1822, is Inner Non-cacheable, Outer Non-cacheable.

For a memory region with a resultant memory type attribute of Normal, that is not Inner Non-cacheable, Outer Non-cacheable, Table D4-43 shows how the stage 1 and stage 2 shareability assignments are combined.

**Table D4-43 Combining the stage 1 and stage 2 shareability assignments for Normal memory[a]**

| Assignment in stage 1 | Assignment in stage 2 | Resultant shareability |
|---|---|---|
| Outer Shareable | Any | Outer Shareable |
| Inner Shareable | Outer Shareable | Outer Shareable |
| Inner Shareable | Inner Shareable | Inner Shareable |
| Inner Shareable | Non-shareable | Inner Shareable |
| Non-shareable | Outer Shareable | Outer Shareable |
| Non-shareable | Inner Shareable | Inner Shareable |
| Non-shareable | Non-shareable | Non-shareable |

a. Applies only if the Normal memory is not Inner Non-cacheable, Outer Non-cacheable, see text.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D4.5 MMU faults

In a VMSAv8-64 implementation, the following mechanisms cause a PE to take an exception on a failed memory access:

**Debug exception** An exception caused by the debug configuration, see Chapter D2 *AArch64 Self-hosted Debug*.

**Alignment fault** An Alignment fault is generated if the address used for a memory access does not have the required alignment for the operation. For more information see *Alignment support* on page B2-75.

**MMU fault** An MMU fault is a fault generated by the fault checking sequence for the current translation regime. The remainder of this section describes MMU faults.

**External abort** Any memory system fault other than a Debug exception, an Alignment fault, or an MMU fault.

Collectively, these mechanisms are called *aborts*.

MMU faults are synchronous exceptions that fall into two categories in AArch64:

- Data Aborts.
- Instruction Aborts

**Note**

The Instruction Abort exception applies to any synchronous memory abort on an instruction fetch. It is not restricted to speculative instruction fetches.

External aborts can be reported synchronously or asynchronously. In AArch64 state, asynchronous external aborts are reported using the SError interrupt.

An access that causes an abort is said to be aborted, and uses the *Fault Address Registers* (FARs) and *Exception Syndrome Registers* (ESRs) to record context information.

For more information, see *Synchronous exception types, routing and priorities* on page D1-1563.

The Exception level that the MMU fault is taken to depends on the translation regime that generated the fault. The fault context saved in the appropriate ESR_ELx register, where ELx is the Exception level that the fault is taken to, is dependent on whether:

- The MMU fault is due to an Instruction or Data Abort.
- The exception is taken from the same or a lower Exception level.

Software stepping, a debug feature, and a misaligned PC exception are the only exceptions that are higher than an Instruction Abort. Only watchpoints are at a lower priority than Data Aborts in the exception priority hierarchy. For details on the exception model priorities, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

The following sections describe the abort mechanisms:

- *Types of MMU faults* on page D4-1825.
- *The MMU fault-checking sequence* on page D4-1827.
- *AArch64 prioritization of synchronous aborts from a single stage of address translation* on page D4-1830.
- *Pseudocode description of the MMU faults* on page D4-1832.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D4.5.1 Types of MMU faults

This section describes the faults that might be detected during one of the fault-checking sequences described in *The MMU fault-checking sequence* on page D4-1827.

The following list includes all the types of exceptions that can occur:

- Alignment fault.
- Permission fault.
- Translation fault.
- Address size fault.
- Synchronous external abort on a translation table walk.
- Access flag fault.
- TLB conflict abort.

When an MMU fault generates an abort for a region of memory, no memory access is made if that region is or could be marked as Device.

The following subsections describe the MMU faults:

- *Permission fault*.
- *Translation fault* on page D4-1826.
- *Address size fault* on page D4-1826.
- *External abort on a translation table walk* on page D4-1826.
- *Access flag fault* on page D4-1827.

#### Permission fault

A Permission fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. See *About the access permissions* on page D4-1808 for information about conditions that cause a Permission fault.

A TLB might hold a translation table entry that cause a Permission fault. Therefore, if the handling of a Permission fault results in an update to the associated translation tables, the software that updates the translation tables must invalidate the appropriate TLB entry, to prevent the stale information in the TLB being used on a subsequent memory access.

This maintenance requirement applies to Permission faults in both stage 1 and stage 2 translations.

Cache maintenance instructions cannot generate Permission faults, except that:

- A stage 1 translation table walk performed as part of a cache maintenance instruction can generate a stage 2 Permission fault as described in *Stage 2 fault on a stage 1 translation table walk* on page D4-1830.
- When the value of SCTLR_EL1.UCI is 1, enabling EL0 execution of the DC CVAU, DC CVAC, DC CIVAC, and IC IVAU instructions:
  - Executing a DC CVAU, DC CVAC, or DC CIVAC instruction at EL0 to a location that does not have read permission at EL0 generates a Permission fault.
  - It is IMPLEMENTATION DEFINED whether executing a IC IVAU instruction at EL0 to a location that does not have read permission at EL0 generates a Permission fault.
- A DC IVAC instruction requires write permission to the address it invalidates, otherwise it generates a Permission fault.

  **Note**

  - Execution of the DCIMVAC instruction in AArch32 state does not have this write permission requirement.
  - When Non-secure EL1&0 stage 2 address translation is enabled, a DC IVAC instruction executed in Non-secure state operates as a DC CIVAC instruction, as described in *Effects of virtualization and security on the cache maintenance instructions* on page D3-1720.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Data Cache Zero instruction, DC ZVA, operates as set of stores to each byte within the block being accessed, and therefore it generates a Permission fault if the translation system does not permit writes to these locations.

### Translation fault

A Translation fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. A Translation fault is generated if bits[1:0] of a translation table descriptor identify the descriptor as either a Fault encoding or a reserved encoding. For more information see *VMSAv8-64 translation table format descriptors* on page D4-1799.

In addition, a Translation fault is generated if the input address for a translation either does not map on to an address range of a TTBR, or the TTBR range that it maps on to is disabled. In these cases the fault is reported as a level 0 Translation fault on the translation stage at which the mapping to a region described by a TTBR failed.

A data or unified cache maintenance by VA instruction can generate a Translation fault. It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate a Translation fault.

The architecture guarantees that any translation table entry that causes a Translation fault is not cached, meaning the TLB never holds such an entry. Therefore, when a Translation fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

### Address size fault

An Address size fault can be generated at any level of lookup.

An Address size fault is generated if one of the following has nonzero address bits above the output address size, for the current stage of translation:

- The TTBR used for the translation.
- A translation table entry.
- The output address of the translation.

For an Address size fault generated because the TTBR used for the translation has nonzero address bits above the output address size, the reported fault code indicates a fault at level 0. Otherwise, the reported fault code indicates the lookup level at which the fault occurred.

A data or unified cache maintenance by VA instruction can generate an Address size fault. It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate an Address size fault.

The architecture guarantees that any translation table entry that causes an Address size fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Address size fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

For more information on Address size faults, see *Output address size* on page D4-1751.

### External abort on a translation table walk

An external abort on a translation table walk can be either synchronous or asynchronous. An external abort on a translation table walk is reported:

- If the external abort is synchronous, using:
  - A synchronous Instruction Abort exception if the translation table walk is for an instruction fetch.
  - A synchronous Data Abort exception if the translation table walk is for a data access.
- If the external abort is asynchronous, using the SError interrupt exception.

#### *Behavior of external aborts on a translation table walk caused by address translation instructions*

The address translation instructions summarized in *Address translation instructions, functional group* on page G4-4172 require translation table walks. An external abort can occur in the translation table walk. This is reported as follows:

- If the external abort is synchronous, using a synchronous Data Abort exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If the external abort is asynchronous, using the SError interrupt exception.

For more information, see *Synchronous faults generated by address translation instructions* on page D4-1797.

### Access flag fault

An Access flag fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. An Access flag fault is generated only if a translation table descriptor with the Access flag bit set to 0 is used.

For more information about the Access flag bit, see *VMSAv8-64 translation table format descriptors* on page D4-1799.

The architecture guarantees that any translation table entry that causes an Access flag fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Access flag fault occurs, the fault handler does not have to execute any TLB maintenance instructions to remove the faulting entry.

Whether any cache maintenance by VA instructions can generate Access flag faults is IMPLEMENTATION DEFINED.

For more information, see *The Access flag* on page D4-1815.

## D4.5.2 The MMU fault-checking sequence

This section describes the MMU checks made for the memory accesses required for instruction fetches and for explicit memory accesses:

- If an instruction fetch faults it generates an Instruction Abort.
- If an data memory access faults it generates a Data Abort.

MMU fault checking is performed for each stage of address translation.

The fault-checking sequence shows a translation from an Input address to an Output address. For more information about this terminology, see *About address translation* on page D4-1748.

**Note**

The descriptions in this section do not include the possibility that the attempted address translation generates a TLB conflict abort, as described in *TLB conflict aborts* on page D4-1836.

*Types of MMU faults* on page D4-1825 describes the faults that an MMU fault-checking sequence can report.

Figure D4-18 on page D4-1828 shows the process of fetching a descriptor from the translation table. For the top-level fetch for any translation, the descriptor is fetched only if the input address passes any required alignment check. As the figure shows, if the translation is stage 1 of the Non-secure EL1&0 translation regime, then the descriptor address is in the IPA address space, and is subject to a stage 2 translation to obtain the required PA. This stage 2 translation requires a recursive entry to the fault checking sequence.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

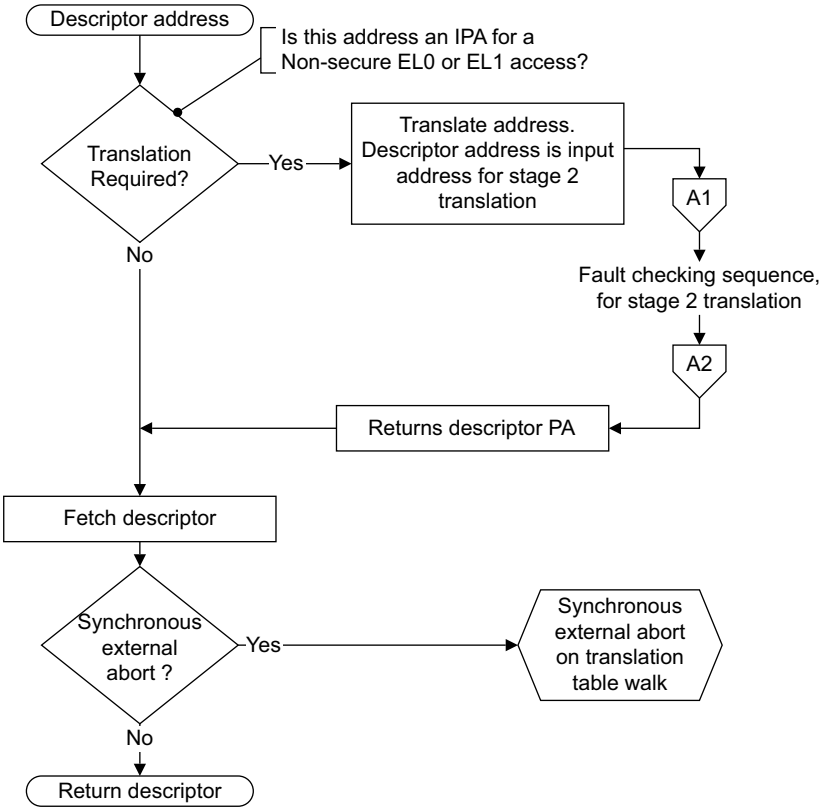
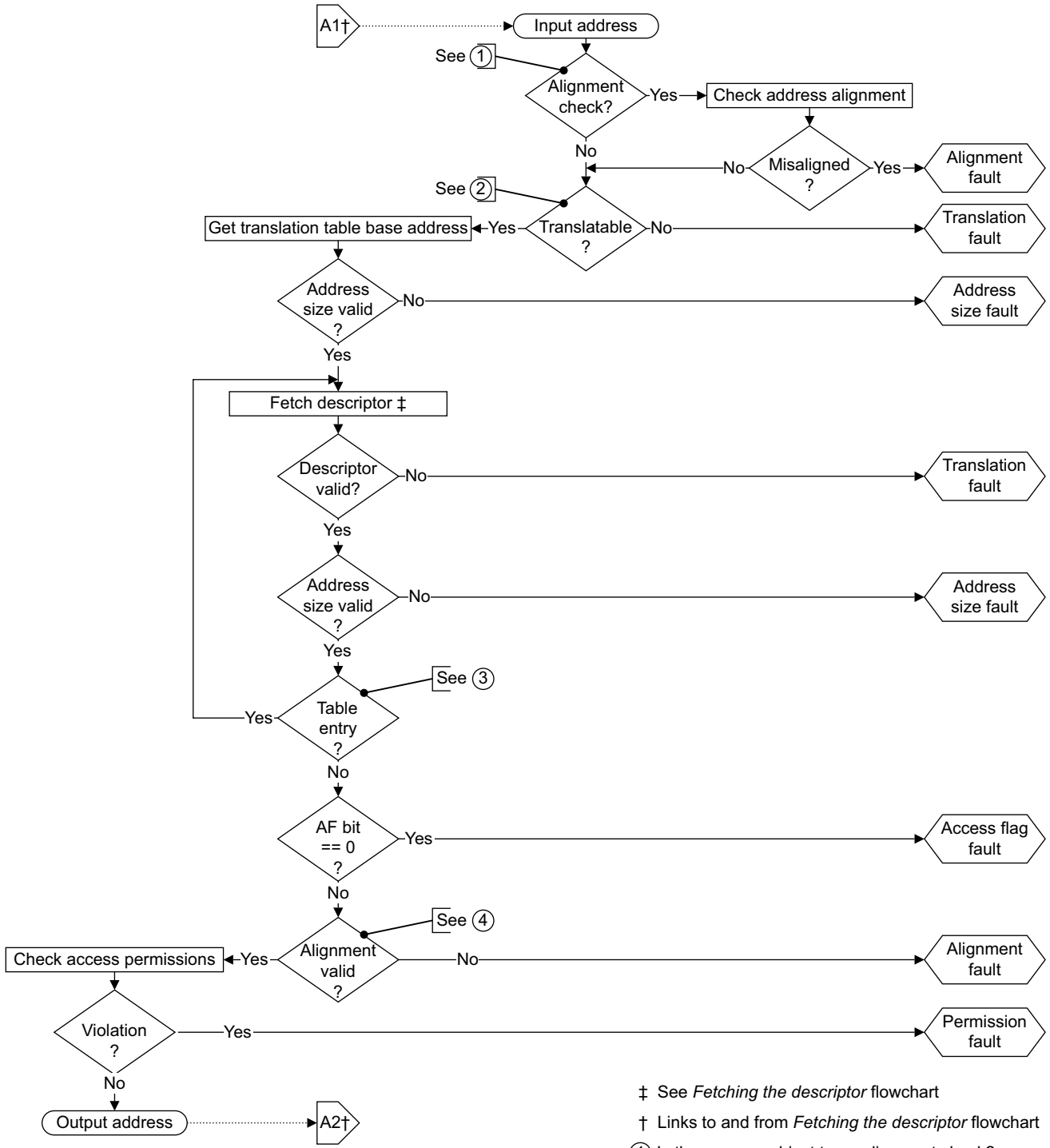



**Figure D4-18 Fetching the descriptor in a VMSAv8-64 translation table walk**

Figure D4-19 on page D4-1829 shows the full VMSA fault checking sequence, including the alignment check on the initial access.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




‡ See *Fetching the descriptor* flowchart

† Links to and from *Fetching the descriptor* flowchart

① Is the access subject to an alignment check?

② Does the address map to a TTBR?

③ Not permitted at the lowest lookup level

④ Fault any unaligned access to Device memory

**Figure D4-19 VMSAv8-64 fault checking sequence**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Stage 2 fault on a stage 1 translation table walk

On performing a translation table walk for the stage 1 translations, the descriptor addresses must be translated from IPA to PA, using a stage 2 translation. This means that a memory access made as part of a stage 1 translation table lookup might generate, on a stage 2 translation:

- A Translation fault, Access flag fault, or Permission fault.
- A synchronous external abort on the memory access.

If SCR_EL3.EA is set to 1, a synchronous external abort is taken to Secure Monitor mode., Otherwise, these faults are reported as stage 2 memory aborts. ESR_EL2.ISS[7] is set to 1, to indicate a stage 2 fault during a stage 1 translation table walk, and the part of the ISS field that might contain details of the instruction is invalid. For more information see *Use of the ESR_EL1, ESR_EL2, and ESR_EL3* on page D1-1539.

Alternatively, a memory access made as part of a stage 1 translation table lookup might target an area of memory with the Device attribute assigned on the stage 2 translation of the address accessed. When the HCR_EL2.PTW bit is set to 1, such an access generates a stage 2 Permission fault.

**Note**

On most systems, such a mapping to Device memory on the stage 2 translation is likely to indicate a Guest OS error, where the stage 1 translation table is corrupted. Therefore, it is appropriate to trap this access to the hypervisor.

A TLB might hold entries that depend on the effect of HCR_EL2.PTW. Therefore, if HCR_EL2.PTW is changed without changing the current VMID, the TLBs must be invalidated before executing in a Non-secure EL1 or EL0 mode. For more information see *Changing HCR_EL2.PTW* on page D4-1849.

A cache maintenance instruction executed at Non-secure EL1 can cause a stage 1 translation table walk that might generate a stage 2 Permission fault, as described in this section. This is an exception to the general rule that a cache maintenance instruction cannot generate a Permission fault.

## D4.5.3 AArch64 prioritization of synchronous aborts from a single stage of address translation

*Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 describes the prioritization of exceptions taken to an Exception level that is using AArch64. This section gives additional information about the prioritization of MMU faults from VMSAv8-64 translation regimes.

**Note**

The priority numbering in this list only shows the relative priorities of aborts from a single stage of address translation in a VMSAv8-64 translation regime. This numbering has no global significance and, for example, does not correlate with the equivalent AArch32 list in *AArch32 prioritization of synchronous aborts from a single stage of address translation* on page G4-4088.

For a single stage of translation in a VMSAv8-64 translation regime, the priority of the memory management faults on a memory access is as follows, ordered from highest priority to lowest priority. For memory accesses that undergo two stages of translation, the *italic entries show where the faults from second stage translations can occur*. A second stage fault within a second stage translation follows the same priority of faults:

1. Alignment fault not caused by memory type. This is possible for a stage 1 translation only.
2. Translation fault due to the input address being out of the address range to be translated or requiring a TTBR that is disabled. This includes VTCR_EL2.T0SZ being inconsistent with VTCR_EL2.SL0.
3. Address size fault on a TTBR caused by either:
   - The check on TCR_EL1.IPS, TCR_EL2.PS, TCR_EL3.PS, or VTCR_EL2.PS.
   - The physical address being out of the range implemented.
4. *Second stage abort on a level 0 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the physical address being out of the range implemented*. This is second stage abort during a first stage translation table walk.
5. Synchronous parity or ECC error on a level 0 lookup of a translation table walk.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

6. Synchronous external abort on a level 0 lookup level of a translation table walk.
7. Translation fault on a level 0 translation table entry.
8. Address Size fault a level 0 lookup translation table entry caused by either:
   - The check on TCR_EL1.IPS, TCR_EL2.PS, TCR_EL3.PS, or VTCR_EL2.PS.
   - The output address being out of the range implemented.
9. *Second stage abort on a level 1 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the physical address being out of the range implemented*. This is second stage abort during a first stage translation table walk.
10. Synchronous parity or ECC error on a level 1 lookup of a translation table walk.
11. Synchronous external abort on a level 1 lookup level of a translation table walk.
12. Translation fault on a level 1 translation table entry.
13. Address size fault on a level 1 lookup translation table entry caused by either:
   - The check on TCR_EL1.IPS, TCR_EL2.PS, TCR_EL3.PS, or VTCR_EL2.PS.
   - The output address being out of the range implemented.
14. *Second stage abort on a level 2 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the physical address being out of the range implemented*. This is second stage abort during a first stage translation table walk.
15. Synchronous parity or ECC error on a level 2 lookup of a translation table walk.
16. Synchronous external abort on a level 2 lookup level of a translation table walk.
17. Translation fault on a level 2 translation table entry.
18. Address size fault on a level 2 lookup translation table entry caused by either:
   - The check on TCR_EL1.IPS, TCR_EL2.PS, TCR_EL3.PS, or VTCR_EL2.PS.
   - The output address being out of the range implemented.
19. *Second stage abort on a level 3 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the physical address being out of the range implemented*. This is second stage abort during a first stage translation table walk.
20. Synchronous parity or ECC error on a level 3 lookup of a translation table walk.
21. Synchronous external abort on a level 3 lookup level of a translation table walk.
22. Translation fault on a level 3 translation table entry.
23. Address size fault on a level 3 lookup translation table entry caused by either:
   - The check on TCR_EL1.IPS, TCR_EL2.PS, TCR_EL3.PS, or VTCR_EL2.PS.
   - The output address being out of the range implemented.
24. Access Flag fault.
25. Alignment fault caused by the memory type.
26. Permission fault.
27. *A fault from the stage 2 translation of the memory access. When stage 2 address translation is enabled this includes an Address size fault caused by the physical address being out of the range implemented*.
28. Synchronous parity or ECC error on the memory access.
29. Synchronous External Abort on the memory access.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- The prioritization of TLB Conflict aborts is IMPLEMENTATION DEFINED, as the exact cause of these aborts depends on the form of TLBs implemented. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.
- The prioritization of IMPLEMENTATION DEFINED MMU faults for a Load-Exclusive or Store-Exclusive to an unsupported memory type is IMPLEMENTATION DEFINED.

### D4.5.4 Pseudocode description of the MMU faults

The following functions generate fault records that describe MMU faults.

```
// AArch64.AccessFlagFault()
// =========================

FaultRecord AArch64.AccessFlagFault(bits(48) ipaddress, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)


    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);

// AArch64.TranslationFault()
// ==========================

FaultRecord AArch64.TranslationFault(bits(48) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Translation, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);

// AArch64.AddressSizeFault()
// ==========================

FaultRecord AArch64.AddressSizeFault(bits(48) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);

// AArch64.PermissionFault()
// =========================

FaultRecord AArch64.PermissionFault(bits(48) ipaddress, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Permission, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

*Abort exceptions* on page D3-1736 describes how fault records are used.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D4.6 Translation Lookaside Buffers (TLBs)

*Translation Lookaside Buffers* (TLBs) reduce the average cost of a memory access by caching the results of translation table walks. TLBs behave as caches of the translation table information, and the VMSA provides TLB maintenance instructions for the management of TLB contents.

**Note**

The ARM architecture permits TLBs to hold any translation table entry that does not directly cause a Translation fault, an Address size fault, or an Access flag fault.

To reduce the need for TLB maintenance on context switches, for EL1&0 stage 1 translations the VMSA can distinguish between *Global pages* and *Process-specific pages*. The *Address Space Identifier* (ASID) identifies pages associated with a specific process and provides a mechanism for changing process-specific tables without having to maintain the TLB structures. Similarly, for the Non-secure EL1&0 translation regime, the *virtual machine identifier* (VMID) identifies the current virtual machine, with its own independent ASID space. The TLB entries include this VMID information, meaning TLBs do not require explicit invalidation when changing from one virtual machine to another, if the virtual machines have different VMIDs.

The following sections describe the architectural requirements for Translation Lookaside Buffers (TLBs) and their maintenance:

- *About ARMv8 Translation Lookaside Buffers (TLBs)*.
- *TLB maintenance requirements and the TLB maintenance instructions* on page D4-1837.

In these descriptions, TLB entries for a translation regime for a particular Exception level are *out of context* when executing at a higher Exception level.

### D4.6.1 About ARMv8 Translation Lookaside Buffers (TLBs)

*Translation Lookaside Buffers* (TLBs) are an implementation technique that caches translations or translation table entries. TLBs avoid the requirement for every memory access to perform a translation table walk in memory. The ARM architecture does not specify the exact form of the TLB structures for any design. In a similar way to the requirements for caches, the architecture only defines certain principles for TLBs:

- The architecture has a concept of an entry locked down in the TLB. The method by which lockdown is achieved is IMPLEMENTATION DEFINED, and an implementation might not support lockdown.
- The architecture does not guarantee that an unlocked TLB entry remains in the TLB.
- The architecture guarantees that a locked TLB entry remains in the TLB. However, a locked TLB entry might be updated by subsequent updates to the translation tables. Therefore, when a change is made to the translation tables, the architecture does not guarantee that a locked TLB entry remains incoherent with an entry in the translation table.
- The architecture guarantees that a translation table entry that generates a Translation fault, an Address size fault, or an Access flag fault is not held in the TLB. However a translation table entry that generates a Permission fault might be held in the TLB.
- Any translation table entry that does not generate a Translation fault, an Address size fault, or an Access flag fault and is not from a translation regime for an Exception level that is lower than the current Exception level might be allocated to an enabled TLB at any time. The only translation table entries guaranteed not to be held in the TLB are those that generate a Translation fault, an Address size fault, or an Access flag fault.

  **Note**

  An enabled TLB can hold a translation table entry that does not itself generate a Translation fault but that points to a subsequent table in the translation table walk. This is referred to as *intermediate caching* of TLB entries.

- Software can rely on the fact that between disabling and re-enabling a stage of address translation, entries in the TLB relating to that stage of translation have not have been corrupted to give incorrect translations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following sections give more information about TLB implementation:

- *Global and process-specific translation table entries*.
- *TLB matching* on page D4-1835.
- *TLB behavior at reset* on page D4-1835.
- *TLB lockdown* on page D4-1835.
- *TLB conflict aborts* on page D4-1836.

See also *TLB maintenance requirements and the TLB maintenance instructions* on page D4-1837.

## Global and process-specific translation table entries

In a VMSA implementation, system software can divide the virtual memory map used by memory accesses at EL1 and EL0 into global and non-global regions, indicated by the nG bit in the translation table descriptors:

**nG == 0** The translation is global, meaning the region is available for all processes.

**nG == 1** The translation is non-global, or process-specific, meaning it relates to the current ASID, as defined in either the TTBR0_EL1 or the TTBR1_EL1.

As indicated by the nG field definitions, each non-global region has an associated *Address Space Identifier* (ASID). These identifiers mean different translation table mappings can co-exist in a caching structure such as a TLB. This means that software can create a new mapping of a non-global memory region without removing previous mappings.

TTBR0_EL1 and TTBR1_EL1 each have an ASID field, and TCR_EL1.A1 determines which of these fields defines the current ASID. See also *ASID size*.

**Note**

The selected ASID applies to the translation of any address for which the value of the nG bit is 1, regardless of whether the address is translated based on TTBR0_EL1 or on TTBR1_EL1.

For a symmetric multiprocessor cluster where a single operating system is running on the set of processing elements, the ARM architecture requires all ASID values to be assigned uniquely within any single Inner Shareable domain. In other words, each ASID value must have the same meaning to all processing elements in the system.

The EL2 translation regime and the EL3 translation regime do not support ASIDs, and all descriptors in these regimes are treated as global.

In a translation regime that supports global and non-global translations, translation table entries from lookup levels other than the final level of lookup are treated as being non-global, regardless of the value of the nG bit.

When a PE is using the VMSAv8-64 translation table format, and is in Secure state, a translation must be treated as non-global, regardless of the value of the nG bit, if NSTable is set to 1 at any level of the translation table walk.

For more information see *Control of Secure or Non-secure memory access* on page D4-1806.

### *ASID size*

In VMSAv8-64, the ASID size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits, and ID_AA64MMFR0_EL1.ASIDBits identifies the supported size. When an implementation supports a 16-bit ASID, TCR_EL1.AS selects whether the top 8 bits of the ASID are used. When the value of TCR_EL1.AS is 0, ASID[15:8]:

- Are ignored by hardware for every purpose other than reads of ID_AA64MMFR0_EL1.
- Are treated as if they are all zeros when used for allocating and matching entries in the TLB.

**Note**

VMSAv8-32 uses an 8-bit ASID. For backwards compatibility, when executing using translations controlled from an Exception level that is using AArch32, the ASID size remains at 8 bits. If the implementation supports 16-bit ASIDS, the 8-bit ASID used is zero-extended to 16 bits.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### TLB matching

A TLB is a hardware caching structure for translation table information. Like other hardware caching structures, it is mostly invisible to software. However, there are some situations where it can become visible. These are associated with coherency problems caused by an update to the translation table that has not been reflected in the TLB. Use of the TLB maintenance instructions described in *TLB maintenance requirements and the TLB maintenance instructions* on page D4-1837 can prevent any TLB incoherency becoming a problem.

A particular case where the presence of the TLB can become visible is if the translation table entries that are in use under a particular ASID and VMID are changed without suitable invalidation of the TLB. This is an issue regardless of whether the translation table entries are global. In some cases, the TLB can hold two mappings for the same address, and this:

- Might generate a Data abort reported using the TLB Conflict fault code, see *TLB conflict aborts* on page D4-1836.
- Might lead to UNPREDICTABLE behavior. In this case, behavior will be consistent with one of the mappings held in the TLB, or with some amalgamation of the values held in the TLB, but cannot give access to regions of memory with permissions or attributes that could not be assigned by valid translation table entries in the translation regime being used for the access. For more information see Appendix K1 *Architectural Constraints on UNPREDICTABLE behaviors*.

### TLB behavior at reset

The ARM architecture does not require a reset to invalidate the TLBs. The architecture recognizes that an implementation might require caches, including TLBs, to maintain their contents over a system reset. Possible reasons for doing so include power management and debug requirements.

Therefore, for ARMv8:

- All TLBs reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
- All TLBs are disabled from reset. All stages of address translation are disabled from reset, and the contents of the TLBs have no effect on address translation. For more information see *Controlling address translation stages* on page D4-1749.
- An implementation can require the use of a specific TLB invalidation routine, to invalidate the TLB arrays before they are enabled after a reset. The exact form of this routine is IMPLEMENTATION DEFINED, but if an invalidation routine is required it must be documented clearly as part of the documentation of the device.

  ARM recommends that if an invalidation routine is required for this purpose, the routine is based on the TLB maintenance instructions described in *TLB maintenance instructions* on page D4-1839.

Similar rules apply to cache behavior, see *Behavior of caches at reset* on page D3-1709.

### TLB lockdown

The ARM architecture recognizes that any TLB lockdown scheme is heavily dependent on the microarchitecture, making it inappropriate to define a common mechanism across all implementations. This means that:

- VMSAv8-64 does not require TLB lockdown support.
- If TLB lockdown support is implemented, the lockdown mechanism is IMPLEMENTATION DEFINED. However, key properties of the interaction of lockdown with the architecture must be documented as part of the implementation documentation.

This means that a region of the system instruction encoding space is reserved for IMPLEMENTATION DEFINED functions, see *Reserved encodings for IMPLEMENTATION DEFINED registers* on page C5-288. An implementation might use some of these encodings to implement TLB lockdown functions. These functions might include:

- Unlock all locked TLB entries.
- Preload into a specific level of TLB. This is beyond the scope of the `PLI` and `PLD` hint instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In an implementation that includes EL2, exceptions generated by problems related to TLB lockdown in a Non-secure EL1 mode, can be routed to either:

- Non-secure EL1, as a Data Abort exception.
- Non-secure EL2, as a Hyp Trap exception.

For more information, see *Traps to EL2 of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page D1-1589.

### TLB conflict aborts

If an address matches multiple entries in the TLB, it is IMPLEMENTATION DEFINED whether a TLB conflict abort is generated.

**Note**

An address can hit multiple entries in the TLB if the TLB has been invalidated inappropriately, for example if TLB invalidation required by the architecture has not been performed.

An implementation can generate TLB conflict aborts on either or both instruction fetches and data accesses. A TLB conflict abort:

- On an instruction fetch is reported as an Instruction abort, see *ISS encoding for an exception from an Instruction abort* on page D7-1972.
- On a data access is reported as a Data abort, see *ISS encoding for an exception from a Data abort* on page D7-1974.

ARMv8 defines the fault status encoding of `0b110000` for TLB conflict aborts. On a TLB conflict abort, the returned syndrome includes the address that generated the fault. That is, it includes the address that was being looked up in the TLB.

It is IMPLEMENTATION DEFINED whether a TLB conflict abort is a stage 1 abort or a stage 2 abort.

**Note**

A stage 2 abort cannot be generated if stage 2 of the Non-secure EL1&0 translation regime is disabled.

The priority of the TLB conflict abort is IMPLEMENTATION DEFINED, because it depends on the form of a TLB that can generate the abort. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.

If an address matches multiple entries in the TLB and no TLB conflict abort is generated, the resulting behavior is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5502. The CONSTRAINED UNPREDICTABLE behavior must not permit access to regions of memory with permissions or attributes that mean they cannot be accessed in the current Security state at the current Exception level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D4.7 TLB maintenance requirements and the TLB maintenance instructions

*Translation Lookaside Buffers* (TLBs) are an implementation mechanism that caches translations or translation table entries. The ARM architecture does not specify the form of any TLB structures, but defines the mechanisms by which TLBs can be maintained.The following sections describe the VMSA TLB maintenance instructions:

- *General TLB maintenance requirements*.
- *TLB maintenance instructions* on page D4-1839.

See also:

- *Maintenance requirements on changing System register values* on page D4-1849.
- *Atomicity of register changes on changing virtual machine* on page D4-1754.

### D4.7.1 General TLB maintenance requirements

TLB maintenance instructions provide a mechanism for invalidating entries from TLB caching structures to ensure that changes to the translation tables are reflected correctly in those TLB caching structures.

The architecture permits the caching of any translation table entry that has been returned from memory without a fault, provided that the entry does not, itself, cause a Translation fault, an Address size fault, or an Access Flag fault. This means that the entries that can be cached include:

- Entries in translation tables that point to subsequent tables to be used in that stage of translation.
- Stage 2 translation table entries used as part of a stage 1 translation table walk
- Stage 2 translation table entries used to translate the output address of the stage 1 translation.

Such entries might be held in intermediate TLB caching structures that are used during a translation table walk and that are distinct from the data caches in that they are not required to be invalidated as the result of writes of the data. The architecture makes no restriction of the form of these intermediate TLB caching structures.

The architecture does not intend to restrict the form of TLB caching structures used for holding translation table entries, and in particular for translation regimes that involve two stages of translation, it is recognized that such caching structures might contain:

- Entries containing information from stage 1 translation table entries, at any level of the translation table walk.
- Entries containing information from stage 2 translation table entries, at any level of the translation table walk.
- Entries that combine information from stage 1 and stage 2 translation table entries, at any level of the translation table walk.

**Note**

For the purpose of TLB maintenance, the term *TLB entry* denotes any structure, including temporary working registers in translation table walk hardware, that holds a translation table entry.

Where a TLB maintenance instruction is:

- Required to apply to stage 1 entries, then it must apply to any cached entries in caching structures that include any stage 1 information that are used to translate the address being invalidated.

  **Note**

  — Where stage 1 information has been cached in multiple TLB entries, as could occur from splintering a page when caching in the TLB, then the invalidation must apply to each cached entry containing stage 1 information from the page that is used to translate the address being invalidated, regardless of whether or not that cached entry would be used to translate the address being invalidated.

  — As stated in *Global and process-specific translation table entries* on page D4-1834, translation table entries from levels of translation other than the final level are treated as being non-global. ARM expects that, in at least some implementations, cached copies of levels of the translation table walk other than the last level are tagged with their ASID, regardless of whether the final level is global. This means that TLB invalidations that involve the ASID require the ASID to match such entries to perform the required invalidation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Required to apply to stage 2 entries only, then:
  - It is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.
  - It must apply to caching structures that contain information only from stage 2 translation table entries.
- Required to apply to both stage 1 and stage 2 entries, then it must apply to any entry in the caching structures that includes information from either a stage 1 translation table entry or a stage 2 translation table entry, including any entry that combines information from both stage 1 and stage 2 translation table entries.

Whenever translation tables entries associated with a particular VMID or ASID are changed, the corresponding entries must be invalidated from the TLB to ensure that these changes are visible to subsequent execution, including speculative execution, that uses the changed translation table entries.

Some System register bit descriptions state that the effect of the bit is permitted to be cached in a TLB. This means that all TLB entries that might be affected by a change in one of these bits must be invalidated whenever that bit is changed, to ensure that the effect of the change of that control bit is visible to subsequent execution including speculative execution, that uses those control bits. This invalidation is required in addition to, and after, the normal synchronization of the System registers described in *Synchronization requirements for System registers* on page D7-1908, and applies to any stage of address translation that is implemented for the translation regime, and VMID if appropriate, that is affected by that control bit. A control bit that is permitted to be cached in a TLB requires this maintenance even when the TLB is not enabled.

In addition to any TLB maintenance requirement, when changing the cacheability attributes of an area of memory, software must ensure that any cached copies of affected locations are removed from the caches. For more information see *Cache maintenance requirement created by changing translation table attributes* on page D4-1852.

Because a TLB never holds any translation table entry that generates a Translation fault, an Address size fault, or an Access Flag fault, a change from a translation table entry that causes a Translation, Address size, or Access flag fault to one that does not fault, does not require any TLB invalidation.

Special considerations can apply to translation table updates that change the memory type, cacheability, or output address of an entry, see *Using break-before-make when updating translation table entries*.

### Using break-before-make when updating translation table entries

To avoid the effects of TLB caching possibly breaking coherency, ordering guarantees or uniprocessor semantics, or possibly failing to clear the exclusive monitors, ARM strongly recommends the use of a break-before-make when changing translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation entries involves any of:

- A change of the memory type.
- A change of the cacheability attributes.
- A change of the output address (OA), if the OA of at least one of the old translation table entry and the new translation table entry is writable.

A break-before-make approach on changing from an old translation table entry to a new translation table entry requires the following steps:

1. Replace the old translation table entry with an invalid entry, and execute a `DSB` instruction.
2. Invalidate the translation table entry with a broadcast TLB invalidation instruction, and execute a `DSB` instruction to ensure the completion of that invalidation.
3. Write the new translation table entry, and execute a `DSB` instruction to ensure that the new entry is visible.

This sequence ensures that at no time are both the old and new entries simultaneously visible to different threads of execution. This means the problems described at the start of this subsection cannot arise.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D4.7.2 TLB maintenance instructions

The architecture defines TLB maintenance instructions, that provide the following:

- Invalidate all entries in the TLB.
- Invalidate a single TLB entry by ASID for a non-global entry.
- Invalidate all TLB entries that match a specified ASID.
- Invalidate all TLB entries that match a specified VA, regardless of the ASID.

Each instruction can be specified as applying only to the PE that executes the instruction, or as applying to all PEs in the same Inner Shareable shareability domain as the PE that executes the instruction.

The following subsubsections describe these instructions:

- *TLB maintenance instruction syntax*.
- *Operation of the TLB maintenance instructions* on page D4-1841.
- *Scope of the A64 TLB maintenance instructions* on page D4-1842.
- *Invalidation of TLB entries from stage 2 translations* on page D4-1845.
- *Broadcast TLB maintenance between AArch32 and AArch64* on page D4-1845.
- *Broadcast TLB maintenance with different translation granule sizes* on page D4-1847.
- *Ordering and completion of TLB maintenance instructions* on page D4-1847.
- *TLB maintenance in the event of TLB conflict* on page D4-1848.
- *The interaction of TLB lockdown with TLB maintenance instructions* on page D4-1848.

*TLB maintenance instructions* on page C5-274 describes the encoding of the TLB maintenance instructions.

#### TLB maintenance instruction syntax

The A64 syntax for TLB maintenance instructions is:

```
TLBI <operation>{, <Xt>}
```

Where:

`<operation>` Is one of `ALLE1`, `ALLE2`, `ALLE3`, `ALLE1IS`, `ALLE2IS`, `ALLE3IS`, `VMALLE1`, `VMALLE1IS`, `VMALLS12E1`, `VMALLS12E1IS`, `ASIDE1`, `ASIDE1IS`, `VA{L}E1`, `VA{L}E2`, `VA{L}E3`, `VA{L}E1IS`, `VA{L}E2IS`, `VA{L}E3IS`, `VAA{L}E1`, `VAA{L}E1IS`, `IPAS2{L}E1`, or `IPAS2{L}E1IS`.

`<operation>` has a structure of `<type><level>{IS}` where:

`<type>` Is one of:

`ALL` All translations used at `<level>`.
For the scope of `ALL` instructions see *ALL* on page D4-1842.
The `ALL` instructions are valid for all values of `<level>`.

`VMALL` All stage 1 translations used at <level> with the current VMID, if appropriate.
For the scope of the `VMALL` instructions see *VMALL* on page D4-1842.
The `VMALL` instructions are valid only when `level` == `E1`.

`VMALLS12` All stage 1 and stage 2 translations used at EL1 with the current VMID, if appropriate.
For the scope of the `VMALLS12` instructions see *VMALLS12* on page D4-1842.
The `VMALLS12` instructions are valid only when `level` == `E1`.

`ASID` All translations used at EL1 with the supplied ASID.
For the scope of the `ASID` instructions see *ASID* on page D4-1843.
The `ASID` instructions are valid only when `level` == `E1`.

`VA{L}` Translations used at `<level>` for the specified address and, if appropriate, the specified ASID.
For the scope of the `VA` instructions see *VA* on page D4-1843. For the scope of the `VAL` instructions see *VAL* on page D4-1843.
The `VA{L}` instructions are valid for all values of `<level>`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`VAA{L}` Translations used at `<level>` for the specified address, for all ASID values, if appropriate.

For the scope of the `VAA` instructions see *VAA* on page D4-1843. For the scope of the `VAAL` instructions see *VAAL* on page D4-1844.

The `VAA{L}` instructions are valid only when `level` == `E1`.

`IPAS2{L}` Translations used at `<level>` for the specified IPA that are held in stage 2 only caching structures.

For the scope of the `IPAS2` instructions see *IPAS2* on page D4-1844. For the scope of the `IPAS2L` instructions see *IPAS2L* on page D4-1844

The `IPAS2{L}` instructions are valid only when `level` == `E1`.

In the `VA{L}`, `VAA{L}`, and `IPAS2{L}` types:

`L` An optional parameter that indicates that the invalidation only applies to caching of entries returned from the final lookup level of the translation table walk.

`<level>` Defines the Exception level of the translation regime that the invalidation applies to. Is one of:

`E1` EL1.

`E2` EL2.

`E3` EL3.

An instruction that applies to the translation regime of an Exception level higher than the Exception level at which the instruction is executed is UNDEFINED.

`TLBI ALLE1{IS}`, `TLBI IPAS2{L}E1{IS}` and `TLBI VMALLS12E1{IS}` are UNDEFINED at EL1.

**Note**

All TLB maintenance instructions are UNDEFINED at EL0.

`IS` When present, it indicates that the function applies to all TLBs in the Inner Shareable shareability domain.

`<Xt>` Passes one or both of an address and an ASID as an argument, where required. `<Xt>` is required for the `TLB ASID`, `TLB VA{L}`, `TLB VAA{L}`, and `TLB IPAS2{L}` instructions.

If EL2 is not implemented, the `TLBI VA{L}E2`, `TLBI VA{L}E2IS`, `TLBI ALLE2`, and `TLBI ALLE2IS` instructions are UNDEFINED.

VMSAv8-64 TLB maintenance instructions that take a register argument that holds a virtual address, an ASID, or both, use the following register argument format:

**Bits[63:48]** ASID. These bits are RES0 if the instruction does not require an ASID argument.

**Bits[47:44]** RES0.

**Bits[43:0]** VA[55:12]. For an instruction that requires a VA argument, the treatment of the low-order bits of this field depends on the translation granule size, as follows:

**4KB granule size** All bits are valid and used for the invalidation.

**16KB granule size** Bits[1:0] RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.

**64KB granule size** Bits[3:0] are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

These bits are RES0 if the instruction does not require a VA argument.

For TLB maintenance instructions that take an address argument, hardware interprets VA[63:56] as each having the same value as VA[55].

If a TLB maintenance instruction targets a translation regime that is using AArch32, meaning the VA is only 32-bit, then software must treat VA[55:32] as RES0, and these bits are ignored when the instruction is executed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the implementation supports 16 bits of ASID then the upper 8 bits of the ASID are RES0 when the context being invalidated only uses 8 bits.

VMSAv8-64 TLB maintenance instructions that take a register argument that holds an IPA, use the following register argument format:

**Bits[63:36]** RES0.

**Bits[35:0]** IPA[47:12]. For an instruction that requires a VA argument, the treatment of the low-order bits of this field depends on the translation granule size, as follows:

- **4KB granule size** All bits are valid and used for the invalidation.
- **16KB granule size** Bits[1:0] RES0 and ignored when the instruction is executed, because IPA[13:12] have no effect on the operation of the instruction.
- **64KB granule size** Bits[3:0] are RES0 and ignored when the instruction is executed, because IPA[15:12] have no effect on the operation of the instruction.

### Operation of the TLB maintenance instructions

Any TLB maintenance instruction can affect any TLB entries that are not locked down.

The TLB maintenance instructions specify the Exception level of the translation regime to which they apply.

**Note**

Because there is no guarantee that an unlocked TLB entry remains in the cache, architecturally it is not possible to tell whether a TLB maintenance instruction has affected any TLB entries that were not specified by the instruction.

If a TLB maintenance instruction specifies a VA, and a data or instruction access to that VA would generate an MMU abort, the TLB maintenance instruction does not generate an abort. VAs for which a TLB maintenance instruction does not generate an abort include VAs that are not in the range of VAs that can be translated.

When EL3 is implemented:

- The TLB maintenance instructions that apply to the EL1&0 translation regime take account of the current Security state, as part of the address translation required for the TLB operation.
- SCR_EL3.NS modifies the effect of the TLB maintenance instructions as follows:
  - For instructions that apply to the EL1&0 translation regime, the SCR_EL3.NS bit identifies whether the maintenance instructions apply to the Secure or Non-secure EL1&0 translation regime.

    **Note**

    If EL3 is not implemented, then there is only a single EL1&0 translation regime.

  - For instructions that apply to the EL2 translation regime, the SCR_EL3.NS bit must be 1 or the instruction is UNDEFINED.
  - For instructions that apply to the EL3 translation regime, the SCR_EL3.NS bit has no effect.

**Note**

- An address-based TLB maintenance instruction that applies to the Inner Shareable domain does so regardless of the Shareability attributes of the address supplied as an argument to the instruction.
- Previous versions of the ARM architecture included TLB maintenance instructions that operated only on instruction TLBs, or only on data TLBs. From the introduction of ARMv7, ARM deprecated any use of these instructions. In ARMv8:
  - AArch64 state does not include any of these instructions.
  - AArch32 state includes some of these instructions, but ARM deprecates their use.

The ARM architecture does not dictate the form in which the TLB stores translation table entries. However, when a TLB maintenance instruction is executed, the minimum size of the table entry that is invalidated from the TLB must be at least the size that appears in the translation table entry.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The Contiguous bit does not affect the minimum size of entry that must be invalidated from the TLB

### Scope of the A64 TLB maintenance instructions

The TLB invalidation instruction <type> affects the different possible cached entries in the TLB as follows:

`ALL` The invalidation applies to all cached copies of the stage 1 and stage 2 translation table entries from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with the state specified by SCR_EL3.NS.

For entries from the Non-secure EL1&0 translation regime, `ALL` applies to entries with any VMID.

For entries from the EL1&0 translation regimes, the invalidation applies to:

- All entries above the final level of lookup.
- All entries at the final level of lookup.

**Note**

This means the invalidation applies to both:

— Global entries.

— Non-global entries with any ASID.

`VMALL` The invalidation applies to all cached copies of the stage 1 translation table entries, from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- For the Non-secure EL1&0 translation regime, the current VMID.

For entries from the EL1&0 translation regimes that meet the other specified conditions, the invalidation applies to:

- All entries above the final level of lookup.
- All entries at the final level of lookup.

**Note**

This means the invalidation applies to both:

— Global entries.

— Non-global entries with any ASID.

`VMALL` is valid only for EL1.

`VMALLS12` The invalidation applies to all cached copies of the stage 1 and stage 2 translation table entries from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- For the Non-secure EL1&0 translation regime, the current VMID.

For entries from the EL1&0 translation regimes that meet the other specified conditions, the invalidation applies to:

- All entries above the final level of lookup.
- All entries at the final level of lookup.

**Note**

This means the invalidation applies to both:

— Global entries.

— Non-global entries with any ASID.

`VMALLS12` is valid only for EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If EL2 is not implemented, or if the TLBI VMALLS12 instruction is executed when the value of SCR_EL3.NS is 0, the instruction is not UNDEFINED but it has the same effect as TLBI VMALL. This is because there are no stage 2 translations to invalidate.

ASID

The invalidation applies to all cached copies of the stage 1 translation table entries from any level of the translation table walk required to translate any address at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- For the Non-secure EL1&0 translation regime, the current VMID.

For entries from the EL1&0 translation regimes that meet the other specified conditions, the invalidation applies only if either:

- The entry is from a level of lookup above the final level and matches the specified ASID.
- The entry is a non-global entry from the final level of lookup and matches the specified ASID.

ASID is valid only for EL1.

VA

The invalidation applies to all cached copies of the stage 1 translation table entries from any level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- For the Non-secure EL1&0 translation regime, the current VMID.

For entries from the EL1&0 translation regimes that meet the other specified conditions, the invalidation applies only if one of the following applies:

- The entry is from a level of lookup above the final level and matches the specified ASID.
- The entry is a global entry from the final level of lookup.
- The entry is a non-global entry from the final level of lookup that matches the specified ASID.

VAL

The invalidation applies to all cached copies of the stage 1 translation table entry from the final level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level, that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- For the Non-secure EL1&0 translation regime, the current VMID.

For entries from the EL1&0 translation regimes that meet the other specified conditions, the invalidation applies only if either:

- The entry is a global entry from the final level of lookup.
- The entry is a non-global entry from the final level of lookup that matches the specified ASID.

VAA

The invalidation applies to all cached copies of the stage 1 translation table entries from any level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- For the Non-secure EL1&0 translation regime, the current VMID.

For entries from the EL1&0 translation regimes that meet the other specified conditions, the invalidation applies to all of:

- All entries above the final level of lookup.
- All entries at the final level of lookup.

**Note**

This means the invalidation applies to both:

— Global entries.

— Non-global entries with any ASID.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`VAAL` The invalidation applies to all cached copies of the stage 1 translation table entry from the final level of the translation table walk required to translate the address specified in the invalidation instruction at the specified Exception level that would be used with all of:

- The Security state specified by SCR_EL3.NS.
- For the Non-secure EL1&0 translation regime, the current VMID.

For entries from the EL1&0 translation regimes that meet the other specified conditions, the invalidation applies to all entries at the final level of lookup.

**Note**

This means the invalidation apples to both:

- Global entries.
- Non-global entries with any ASID.

`IPAS2` The invalidation applies to all cached copies of the stage 2 translation table entries from any level of the translation table walk required to translate the specified IPA, that both:

- Are held in TLB caching structures holding stage 2 only entries.
- Would be used with the current VMID.

It is not required that this instruction invalidates TLB caching structures holding entries that combine stage 1 and stage 2 of the translation.

The only translation regime to which this instruction can apply is the Non-secure EL1&0 translation regime.

When executed with the SCR_EL3.NS==0, or in an implementation that does not implement EL2, this instruction is a `NOP`.

For more information about the architectural requirements for the `IPAS2` instruction see *Invalidation of TLB entries from stage 2 translations* on page D4-1845.

`IPAS2L` The invalidation applies to cached copies of the stage 2 translation table entry from the final level of the stage 2 translation table walk required to translate the specified IPA, that both:

- Are held in TLB caching structures holding stage 2 only entries.
- Would be used with the current VMID.

It is not required that this instruction invalidates TLB caching structures holding entries that combine stage 1 and stage 2 of the translation.

The only translation regime to which this instruction can apply is the Non-secure EL1&0 translation regime.

When executed with the SCR_EL3.NS==0, or in an implementation that does not implement EL2, this instruction is a `NOP`.

For more information about the architectural requirements for the `IPAS2L` instruction see *Invalidation of TLB entries from stage 2 translations* on page D4-1845.

The entries that the invalidations apply to are not affected by the state of any other control bits involved in the translation process. Therefore, the following is a non-exhaustive list of control bits that do not affect how a TLB maintenance instruction updates the TLB entries:

**In AArch64** SCTLR_EL1.M, SCTLR_EL2.M, SCTLR_EL3.{M, RW], HCR_EL2.{VM, RW}, TCR_EL1.{TG1, EPD1, T1SZ, TG0, EPD0, T0SZ, AS, A1}, TCR_EL2.{TG0, T0SZ}, TCR_EL3.{TG0, T0SZ}, VTCR_EL2.{SL0, T0SZ}, TTBR0_EL1.ASID, TTBR1_EL1.ASID.

**In AArch32** SCTLR.M, HCR.VM, TTBCR.{EAE, PD1, PD0, N, EPD1, T1SZ, EPD0, T0SZ, A1}, HTCR.T0SZ, VTCR.{SL0, T0SZ}, TTBR0.ASID, TTBR1.ASID, CONTEXTIDR.ASID.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- ARM expects most TLB maintenance performed by an operating system to occur to the last level entries of the stage 1 translation table walks, and the purpose of the address-based TLB invalidation instructions where the invalidation need only apply to caching of entries returned from the last level of translation table walk of stage 1 translation is to avoid unnecessary loss of the intermediate caching of the translation table entries. Similarly, for stage 2 translations ARM expects that most TLB maintenance performed by a hypervisor for a given Guest operation system will affect only the last level entries of the stage 2 translations. Therefore, similar capability is provided for instructions that invalidate single stage 2 entries.
- The architecture permits the invalidation of entries in TLB caching structures at any time, so for each of these instructions the definition is in terms of the minimum set of entries that must be invalidated from TLB caching structures, and an implementation might choose to invalidate more entries. In general, for best performance, ARM recommends not invalidating entries that are not required to be invalidated.
- Dependencies on the VMID for the Non-secure EL1&0 translation regime apply even when HCR_EL2.VM is set to 0. Because the architecture does not require the VTTBR_EL2.VMID field to be reset in hardware, the reset routine of each active PE must initialize VTTBR_EL2.VMID[7:0] to a common value such as 0, even if stage 2 translation is not in use.

### Invalidation of TLB entries from stage 2 translations

The architectural requirements of the `IPAS2` instruction are that:

1. The following code is sufficient to invalidate all cached copies of the stage 2 translation of the IPA held in `Xt` for the current VMID, with the corresponding requirement for the broadcast versions of the instructions:

   ```
   TLBI IPAS2E1, Xt
   DSB
   TLBI VMALLE1
   ```

2. The following code is sufficient to invalidate all cached copies of the stage 2 translations of the IPA held in `Xt` used to translate the virtual address VA (and the specified ASID when executing `TLBI VAE1`) held in `Xt2`, with the corresponding requirement for the broadcast versions of the instructions:

   ```
   TLBI IPAS2E1, Xt
   DSB
   TLBI VAE1, Xt2     ; or TLBI VAAE1, Xt2
   ```

3. The following code is sufficient to invalidate all cached copies of the stage 2 translations of the IPA held in `Xt` used to translate the IPA produced by the last level of stage 1 translation table lookup for the virtual address VA (and ASID when executing `TLBI VALE1`) held in `Xt2`, with the corresponding requirement for the broadcast versions of the instructions:

   ```
   TLBI IPAS2E1, Xt
   DSB
   TLBI VALE1, Xt2    ; or TLBI VAALE1, Xt2
   ```

**Note**

Sequences 1, 2, and 3 must use the `TLBI IPAS2E1` instruction even when Non-secure EL1&0 stage 1 translation is disabled.

Equivalent architectural requirements apply to the `IPAS2L` instruction, except that the only TLB entries that must be invalidated by an `IPAS2L` instruction are those that come from the final level of the translation table lookup.

### Broadcast TLB maintenance between AArch32 and AArch64

In most cases, a TLB maintenance instruction affecting the Inner Shareable shareability domain executed by a PE in an Exception level that is using AArch64 also affects any other PE in the same Inner Shareable domain that is executing at the same Exception level and is using AArch32, provided that the address, qualify the scope of the ASID and VMID matching requirements of the original instruction are met, as specified in *Scope of the A64 TLB maintenance instructions* on page D4-1842.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The requirement to match means that the invalidation only occurs on the PE that is using AArch32 if, for the PE that executed the TLB maintenance instruction at an Exception level that is using AArch64, both of the following apply:

- If VA matching is required, the VA is `0x0000 FFFF FFFF` or lower of the memory space.
- If ASID matching is required and the PE is using a 16-bit ASID, then the top 8 bits of the ASID are zero.

Except for the cases identified here, a TLB maintenance instructions affecting the Inner Shareable shareability domain executed by a PE in an Exception level that is using AArch32 also affects any other PE in the same Inner Shareable domain that is executing at the same Exception level and is using AArch64, provided that the address, ASID, and VMID matching requirements of the original instruction are met, as specified in *Scope of the A64 TLB maintenance instructions* on page D4-1842. In addition, for the instruction executed in AArch32 state:

- For a `TLBIMVAAIS`, `TLBIMVAALIS`, `TLBIMVAHIS`, `TLBIMVAIS`, `TLBIMVALHIS`, or `TLBIMVALIS` instruction, the VA supplied as an argument is zero-extended.
- For a `TLBIIPAS2IS` or `TLBIIPAS2LIS` instruction, the IPA supplied as an argument is zero-extended.
- For a `TLBIASIDIS`, `TLBIMVAIS`, or `TLBIMVALIS` instruction, the ASID supplied as an argument is zero-extended if the PE executing in AArch64 state is using a 16-bit ASID.

The virtual address from the instruction executed in AArch32 state is zero-extended, and the ASID is zero-extended if the PE executing in AArch64 state is using a 16-bit ASID.

The exceptions to these general rules are as follows:

1. An ARMv7 PE in the same Inner Shareable domain is treated in the same way as an ARMv8 PE for which EL3 is using AArch32, except that if an ARMv8 PE issues a broadcast instruction that does not exist in ARMv7, then that instruction is not required to have an effect on the TLBs of the ARMv7 PE. The instructions that do not exist in ARMv7 include the following TLB maintenance instructions that ARMv8 adds to the T32 and A32 instruction sets:
   - The following instructions that operate on TLB entries for the final level of translation table walk for stage 1 translations:

     `TLBIMVALIS`, `TLBIMVAALIS`, `TLBIMVALHIS`, `TLBIMVAL`, `TLBIMVAAL`, and `TLBIMVALH`.
   - The following instructions that operate by IPA on TLB entries for stage 2 translations:

     `TLBIIPAS2IS`, `TLBIIPAS2LIS`, `TLBIIPAS2`, and `TLBIIPAS2L`.
2. The number of Exception levels in Secure state depends on whether EL3 is using AArch32 or EL3 is using AArch64. This means that, within the Inner Shareable domain, there might be PEs with different numbers of Exception levels in Secure state. Therefore, the following exceptions are made to the general rules:
   - If a PE with EL3 using AArch32 issues a broadcast AArch32 TLB maintenance instruction affecting Secure entries, and the Inner Shareable domain also contains PEs with EL3 using AArch64, then the architecture does not require that the broadcast AArch32 TLB maintenance instruction has any effect on either:
     - The EL3 translation regime of the PEs with EL3 using AArch64.
     - The Secure EL1 translation regime of the PEs with EL3 using AArch64, regardless of whether the Secure EL1 translation regime is using AArch64 or AArch32.
   - If a PE with EL3 using AArch64 issues a broadcast AArch64 TLB maintenance instruction affecting EL3 entries, and the Inner Shareable domain also contains PEs with EL3 using AArch32, then the architecture does not require that the broadcast AArch64 TLB maintenance instruction has any effect on the EL3 translation regime of the PEs with EL3 using AArch32.
   - If a PE with EL3 using AArch64 issues a broadcast AArch64 TLB maintenance instruction affecting Secure EL1 entries, and the Inner Shareable domain also contains PEs with EL3 using AArch32 then the architecture does not require that the broadcast AArch64 TLB maintenance instruction has any effect on the EL3 translation regime of the PEs with EL3 using AArch32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

While the exceptions to the general rule mean the architecture does not require the specified TLB invalidations, the architecture also does not require that entries in the TLB remain in the TLB at any time, and so it is permissible that such broadcast instructions affect these translation regimes.

### Broadcast TLB maintenance with different translation granule sizes

In the following cases, a broadcast TLB maintenance instruction is not required to perform any invalidation on the recipient PE:

- The TLB maintenance instruction specifying a virtual address and affecting the EL2 translation regime or the EL3 translation regime is broadcast from a PE using one translation granule size for that translation regime to a PE using a different translation granule size for that same translation regime
- The TLB maintenance instruction specifying a virtual address and affecting the EL1 translation regime is broadcast from a PE using one stage 1 translation granule size for that translation regime for a particular ASID (if applicable), VMID (if applicable), and Security state, to a PE where EL1 for the same ASID (if applicable), VMID (if applicable), and Security state, is using a different stage 1 translation granule size.
- The TLB maintenance instruction specifying a virtual address and affecting the Non-secure EL1 translation regime is broadcast from a PE using one stage 2 translation granule size for a particular ASID (if applicable) and VMID, to a PE where EL1 for the same ASID (if applicable) and VMID is using a different stage 2 translation granule size.
- The TLB maintenance instruction specifying an intermediate physical address and affecting the Non-secure EL1 translation regime is broadcast from a PE using one stage 2 translation granule size for a particular VMID to a PE where EL1 for the same VMID is using a different stage 2 translation granule size.

### Ordering and completion of TLB maintenance instructions

For AArch64 execution, a TLB maintenance instruction can be executed in any order relative to:

- Any load or store instruction, unless a DSB is executed between the load or store and the TLB maintenance instruction.

  **Note**

  In the ARM architecture, a translation table walk is considered to be a separate observer, and a store to translation tables can be observed by that separate observer at any time after the instruction has been executed, but is only guaranteed to be observable after the execution of a `DSB` instruction by the PE that executed the store to the translation tables.

- Another TLB maintenance instruction, unless a `DSB` is executed between the instructions.
- A data or instruction cache maintenance instruction, unless a `DSB` is executed between the instructions.

For AArch64 execution, the completion rules are:

- A TLB invalidate instruction is complete when all memory accesses using the TLB entries that have been invalidated have been observed by all observers, to the extent that those accesses must be observed. The shareability and cacheability of the accessed memory locations determine the extent to which the accesses must be observed.

  **Note**

  For TLB maintenance instructions that affect other PEs, the memory accesses from those PEs that used the TLB entries that have been invalidated are included in the set of memory accesses that must have been observed when the TLB maintenance instruction is complete.

  After the TLB invalidate instruction is complete, no new memory accesses using the invalidated TLB entries will be observed by those observers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

This requirement does not mean that speculative memory accesses cannot be performed using those entries if it is impossible to tell that those memory accesses have been observed by the observers.

- A TLB maintenance instruction can complete at any time after it is issued, but is only guaranteed to be complete after the execution of `DSB` by the PE that executed the TLB maintenance instruction.
- The effects of a completed TLB maintenance instruction are only guaranteed to be visible on the PE that executed the instruction after the execution of an `ISB` instruction by the PE that executed the TLB maintenance instruction.

In all cases in this section, where a `DMB` or `DSB` is referred to, it refers to a `DMB` or `DSB` whose required access type is both loads and stores.

### TLB maintenance in the event of TLB conflict

In the event that multiple entries in the TLB are being used to translate a given address (which implies that an attempt to access the given address might give rise to a TLB Conflict abort), it is IMPLEMENTATION DEFINED as to the form of TLB maintenance operation that the software must perform in order to be guaranteed that all TLB entries associated with the given address and translation regime have been invalidated. In all cases, an `ALL` or `VMALL` form of TLB maintenance operation that targets the given translation regime is guaranteed to remove all entries within that regime, even if there are multiple, conflicting TLB entries for any given address within that regime.

### The interaction of TLB lockdown with TLB maintenance instructions

The precise interaction of TLB lockdown with the TLB maintenance instructions is IMPLEMENTATION DEFINED. However, the architecturally-defined TLB maintenance instructions must comply with these rules:

- The effect on a locked TLB entry of a TLB invalidate all operation that would invalidate that entry if the entry was not locked is IMPLEMENTATION DEFINED. However, the instruction operation must be implemented as one of the following options:
  - The operation has no effect on entries that are locked down.
  - The operation generates an IMPLEMENTATION DEFINED Data Abort exception if an entry is locked down, or might be locked down.

    Any such exceptions taken from Non-secure EL1 can be trapped to EL2, see *Traps to EL2 of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page D1-1589.

**Note**

These options permit a usage model for TLB invalidate routines, where the routine invalidates a large range of addresses, without considering whether any entries are locked in the TLB.

- The effect on a locked TLB entry of a TLB invalidate by VA or invalidate by ASID match operation that would invalidate that entry if the entry was not locked is IMPLEMENTATION DEFINED. However, the operation must implement one of the following options:
  - The locked entry is invalidated in the TLB.
  - The operation has no effect on any locked entry in the TLB. In the case of an invalidate single entry by VA, this means the PE treats the operation as a `NOP`.
  - The operation generates an IMPLEMENTATION DEFINED Data Abort exception if it operates on an entry that is locked down, or might be locked down.

The exception syndrome definitions include a fault code for cache and TLB lockdown faults, see *ESR_EL1, Exception Syndrome Register (EL1)* on page D7-1948.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Any implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other specified alternatives for the locked entries.

ARM recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use the architecturally-defined operations. This minimizes the number of customized operations required.

In addition, an implementation that uses an abort mechanism for handling the effect of TLB maintenance instructions on entries that can be locked down but are not actually locked down must provide an IMPLEMENTATION DEFINED mechanism that ensures that no TLB entries are locked.

Similar rules apply to cache lockdown, see *The interaction of cache lockdown with cache maintenance instructions* on page D3-1725.

The architecture does not guarantee that any unlocked entry in the TLB remains in the TLB. This means that, as a side-effect of any TLB maintenance instruction, any unlocked entry in the TLB might be invalidated.

### D4.7.3 Maintenance requirements on changing System register values

The TLB contents can be influenced by control bits in a number of System registers. This means the TLB must be invalidated after any changes to these bits, unless the changes are accompanied by a change to the VMID or ASID that defines the context to which the bits apply. The general form of the required invalidation sequence is as follows:

```
; Change control bits in System registers
ISB        ; Synchronize changes to the control bits
; Perform TLB invalidation of all entries that might be affected by the changed control bits
```

The System register changes that maintenance requirement applies to are:

- Any change to the MAIR_EL1, MAIR_EL2, or MAIR_EL3 registers.
- Any change to the AMAIR_EL1, AMAIR_EL2, or AMAIR_EL3 registers.
- Any change to SCTLR_EL1.EE, SCTLR_EL2.EE, or SCTLR_EL3.EE.
- Any change to SCTLR_EL1.WXN, SCTLR_EL2.WXN, or SCTLR_EL3.WXN.
- Any change to any of the SCR_EL3.{RW, SIF} bits.
- Any change to any of the HCR_EL2.{RW, DC, PTW, VM} bits. See also *Changing HCR_EL2.PTW*.
- Any changes to the registers that control address translation:
  - Any change to any of the TCR_EL1, TCR_EL2, TCR_EL3, or VTCR_EL2 registers.
  - Any change to the TTBR0_EL1, TTBR1_EL1, TTBR0_EL2, TTBR0_EL3, or VTTBR_EL2 registers.

#### Changing HCR_EL2.PTW

When the value of the Protected table walk bit, HCR_EL2.PTW, is 1, a stage 1 translation table access in the Non-secure EL1&0 translation regime, to an address that is mapped to any type of Device memory by its stage 2 translation, generates a stage 2 Permission fault. A TLB associated with a particular VMID might hold entries that depend on the effect of HCR_EL2.PTW. Therefore, if the value of HCR_EL2.PTW is changed without a change to the VMID value, all TLB entries associated with the current VMID must be invalidated before executing software at Non-secure EL1 or EL0. If this is not done, behavior is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5502.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D4.8 Caches in a VMSA implementation

The ARM architecture describes the required behavior of an implementation of the architecture. As far as possible it does not restrict the implemented microarchitecture, or the implementation techniques that might achieve the required behavior.

In particular, maintaining this level of abstraction is difficult when describing the relationship between memory address translation and caches, especially regarding the indexing and tagging policy of caches. This section:

- Summarizes the architectural requirements for the interaction between caches and address translation.
- Gives some information about the likely implementation impact of the required behavior.

The following sections give this information:

- *Data and unified caches*.
- *Instruction caches*.

In addition, *Cache maintenance requirement created by changing translation table attributes* on page D4-1852 describes the cache maintenance required after updating the translation tables to change the attributes of an area of memory.

For more information about cache maintenance see *Cache maintenance instructions* on page D3-1716, that describes the cache maintenance instructions in the A64 instruction set.

### D4.8.1 Data and unified caches

For data and unified caches, the use of address translation is entirely transparent to any data access that is not UNPREDICTABLE.

This means that the behavior of accesses from the same observer to different VAs, that are translated to the same PA with the same memory attributes, is fully coherent. This means these accesses behave as follows, regardless of which VA is accessed:

- Two writes to the same PA occur in program order.
- A read of a PA returns the value of the last successful write to that PA.
- A write to a PA that occurs, in program order, after a read of that PA, has no effect on the value returned by that read.

The memory system behaves in this way without any requirement to use barrier or cache maintenance instructions.

In addition, if cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

These properties are consistent with implementing all caches that can handle data accesses as *Physically-indexed, physically-tagged* (PIPT) caches.

### D4.8.2 Instruction caches

In the ARM architecture, an instruction cache is a cache that is accessed only as a result of an instruction fetch. Therefore, an instruction cache is never written to by any load or store instruction executed by the PE.

The ARM architecture supports three different behaviors for instruction caches. For ease of reference and description these are identified by descriptions of the associated expected implementation, as follows:

- PIPT instruction caches.
- *Virtually-indexed, physically-tagged* (VIPT) instruction caches.
- ASID and VMID tagged *Virtually-indexed, virtually-tagged* (VIVT) instruction caches.

The CTR_EL0.L1Ip field identifies the form of the instruction caches.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following subsections describe the behavior associated with these cache types, including any occasions where explicit cache maintenance is required to make the use of address translation transparent to the instruction cache:

- *PIPT instruction caches*.
- *VIPT instruction caches*.
- *ASID and VMID tagged VIVT instruction caches*.
- *The IVIPT Extension* on page D4-1852.

**Note**

For software to be portable between implementations that might use any of PIPT instruction caches, VIPT instruction caches, or ASID and VMID tagged VIVT instruction caches, the software must invalidate the instruction cache whenever any condition occurs that would require instruction cache maintenance for at least one of the instruction cache types.

### PIPT instruction caches

For PIPT instruction caches, the use of memory address translation is entirely transparent to all instruction fetches that are not UNPREDICTABLE.

If cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

An implementation that provides PIPT instruction caches implements the IVIPT Extension, see *The IVIPT Extension* on page D4-1852.

### VIPT instruction caches

For VIPT instruction caches, the use of memory address translation is transparent to all instruction fetches that are not UNPREDICTABLE, except for the effect of memory address translation on instruction cache invalidate by address operations.

**Note**

Cache invalidation is the only cache maintenance that can be performed on an instruction cache.

If instruction cache invalidation by address is performed on a memory location, the effect of that invalidation is visible only to the virtual address supplied with the operation. The effect of the invalidation might not be visible to any other aliases of that physical memory location.

The only architecturally-guaranteed way to invalidate all aliases of a physical address from a VIPT instruction cache is to invalidate the entire instruction cache.

An implementation that provides VIPT instruction caches implements the IVIPT Extension, see *The IVIPT Extension* on page D4-1852.

### ASID and VMID tagged VIVT instruction caches

For ASID and VMID tagged VIVT instruction caches, if the instructions at any virtual address change, for a given translation regime and a given ASID and VMID, as appropriate, then instruction cache maintenance is required to ensure that the change is visible to subsequent execution. This maintenance is required when writing new values to instruction locations. It can also be required as a result of any of the following situations that change the translation of a virtual address to a physical address, if, as a result of the change to the translation, the instructions at the virtual addresses change:

- For any translation regime other than the Non-secure EL1&0 translation regime, enabling or disabling stage 1 translations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For the Non-secure EL1&0 translation regime:
  - When stage 2 translations are enabled, enabling or disabling stage 1 translations unless accompanied by a change of VMID.
  - When stage 2 translations are disabled, enabling or disabling stage 1 translations.
  - Enabling or disabling stage 2 translations.
- Writing new mappings to the translation tables.
- Any change to the TCR or TTBR for the current translation regime, unless:
  - For a change to the Secure EL1&0 translation regime, the change is accompanied by a change to the ASID.
  - For a change to the stage 1 translations of the Non-secure EL1&0 translation regime, the change is accompanied by a change to the ASID or VMID.
  - For a change to the stage 2 translations of the Non-secure EL1&0 translation regime, the change is accompanied by a change to the VMID.

**Note**

For ASID and VMID tagged VIVT instruction caches only, for a given translation regime and a given ASID and VMID, as appropriate, invalidation is not required if a change to the translations is such that the instructions associated with the non-faulting translations of a virtual address remain unchanged through the change to the translations, even if the physical locations being mapped to by the changed translation have been written as part of changing the translation.

Examples of situations where this might occur include:

- Copy-on-Write.
- Demand Paging of memory locations to/from disk.

This does not apply for VIPT or PIPT instruction caches, because those caches hold copies of physical addresses, and therefore must be invalidated when the contents are written to, to avoid the use of stale entries.

If instruction cache invalidation by address is performed on a memory location, the effect of that invalidation is visible only to the virtual address supplied with the operation. The effect of the invalidation might not be visible to any other aliases of that physical memory location.

The only architecturally-guaranteed way to invalidate all aliases of a physical address from an ASID and VMID tagged VIVT instruction cache is to invalidate the entire instruction cache.

### The IVIPT Extension

An implementation in which the instruction cache exhibits the behaviors described in *PIPT instruction caches* on page D4-1851, or those described in *VIPT instruction caches* on page D4-1851, is said to implement the *IVIPT Extension* to the ARM architecture.

The formal definition of the IVIPT Extension to the ARM architecture is that it reduces the instruction cache maintenance requirement to the following condition:

- Instruction cache maintenance is required only after writing new data to a physical address that holds an instruction.

## D4.8.3 Cache maintenance requirement created by changing translation table attributes

Any change to the translation tables to change the attributes of an area of memory can require maintenance of the translation tables, as described in *General TLB maintenance requirements* on page D4-1837. If the change affects the cacheability attributes of the area of memory, including any change between Write-Through and Write-Back attributes, software must ensure that any cached copies of affected locations are removed from the caches, typically


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

by cleaning and invalidating the locations from the levels of cache that might hold copies of the locations affected by the attribute change. Any of the following changes to the inner cacheability or outer cacheability attribute creates this maintenance requirement:

- Write-Back to Write-Through
- Write-Back to Non-cacheable
- Write-Through to Non-cacheable
- Write-Through to Write-Back.

The cache clean and invalidate avoids any possible coherency errors caused by mismatched memory attributes.

Similarly, to avoid possible coherency errors caused by mismatched memory attributes, the following sequence must be followed when changing the shareability attributes of a cacheable memory location:

1. Make the memory location Non-cacheable, Outer Shareable.
2. Clean and invalidate the location from them cache.
3. Change the shareability attributes to the required new values.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter D5
# The Performance Monitors Extension

This chapter describes the ARMv8 implementation of the ARM Performance Monitors, that are an optional non-invasive debug component. It describes version 3 of the *Performance Monitor Unit* (PMU) architecture, PMUv3. It contains the following sections:


- *About the Performance Monitors* on page D5-1856.
- *Accuracy of the Performance Monitors* on page D5-1859.
- *Behavior on overflow* on page D5-1861.
- *Attributability* on page D5-1863.
- *Effect of EL3 and EL2* on page D5-1864.
- *Event filtering* on page D5-1866
- *Performance Monitors and Debug state* on page D5-1868.
- *Counter enables* on page D5-1869.
- *Counter access* on page D5-1870.
- *Events, Event numbers and mnemonics* on page D5-1872.
- *Performance Monitors Extension registers* on page D5-1893.


**Note**

Table K13-1 on page K13-5708 disambiguates the general register references used in this chapter.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D5.1 About the Performance Monitors

In ARMv8-A, the Performance Monitors Extension is an OPTIONAL feature of an implementation, but ARM strongly recommends that ARMv8-A implementations include version 3 of the Performance Monitors Extension, PMUv3.

**Note**

No previous versions of the Performance Monitors Extension can be implemented in ARMv8.

The basic form of the Performance Monitors is:

- A 64-bit cycle counter, see *Time as measured by the Performance Monitors cycle counter* on page D5-1858.
- A number of 32-bit event counters. The event counted by each counter is programmable. ARMv8 provides space for up to 31 counters. The actual number of counters is IMPLEMENTATION DEFINED, and the specification includes an identification mechanism.

  **Note**

  ARM recommends that at least two counters are implemented, and that hypervisors provide at least this many counters to guest operating systems.

- Controls for:
  - Enabling and resetting counters.
  - Flagging overflows.
  - Enabling interrupts on overflow.

  Monitoring software can enable the cycle counter independently of the event counters.

The events that can be monitored split into:

- Architectural and microarchitectural events that are likely to be consistent across many microarchitectures.
- Implementation-specific events.

The PMU architecture uses event numbers to identify an event. It:

- Defines event numbers for common events, for use across many architectures and microarchitectures.

  **Note**

  Implementations that include PMUv3 must, as a minimum requirement, implement a subset of the common events. See *Common event numbers* on page D5-1875.

- Reserves a large event number space for IMPLEMENTATION DEFINED events.

The full set of events for an implementation is IMPLEMENTATION DEFINED. ARM recommends that implementations include all of the events that are appropriate to the architecture profile and microarchitecture of the implementation.

The event numbers of the common events are reserved for the specified events. Each of these event numbers must either:

- Be used for its assigned event.
- Not be used.

When a implementation supports monitoring of an event that is assigned a common event number, ARM strongly recommends that it uses that number for the event. However, software might encounter implementations where an event assigned a number in this range is monitored using an event number from the IMPLEMENTATION DEFINED range.

**Note**

ARM might define other common event numbers. This is one reason why software must not assume that an event with an assigned common event number is never monitored using an event number from the IMPLEMENTATION DEFINED range.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When an implementation includes the Performance Monitors Extension, ARMv8 defines the following possible interfaces to the Performance Monitors Extension registers:

- A System register interface. This interface is mandatory.
- An external debug interface which optionally supports memory-mapped accesses. Implementation of this interface is OPTIONAL. See Chapter I2 *Recommended Memory-mapped Interfaces to the Performance Monitors*.

An operating system can use the System registers to access the counters. This supports a number of uses, including:

- Dynamic compilation techniques.
- Energy management.

Also, if required, the operating system can enable application software to access the counters. This enables an application to monitor its own performance with fine-grain control without requiring operating system support. For example, an application might implement per-function performance monitoring.

There are many situations where performance monitoring features integrated into the implementation are valuable for applications and for application development. When an operating system does not use the Performance Monitors itself, ARM recommends that the operating system enables application software to access the Performance Monitors.

A hypervisor running on the PE can limit the access of a Non-secure operating system to the Performance Monitors.

To enable interaction with external monitoring, an implementation might consider additional enhancements, such as providing:

- A set of events, from which a selection can be exported onto a bus for use as external events.
- The ability to count external events. This enhancement requires the implementation to include a set of external event input signals.

The Performance Monitors Extension is common to AArch64 operation and AArch32 operation. This means the ARMv8 architecture defines both AArch64 and AArch32 System registers to access the Performance Monitors. For example, the Performance Monitors Cycle Count Register is accessible as:

- When executing in AArch64 state, PMCCNTR_EL0, see *PMCCNTR_EL0, Performance Monitors Cycle Count Register* on page D7-2239.
- When executing in AArch32 state, PMCCNTR, see *PMCCNTR, Performance Monitors Cycle Count Register* on page G6-4741.

The ARMv8 definition of the PMDEVARCH.ARCHID field specifies the Performance Monitors architecture version as PMUv3. This indicates that the Performance Monitors Extension conforms to the memory-mapped programmers' model described in this Manual.

**Note**

The PMUv3 memory-mapped programmers' model can be used by devices other than ARMv8 processors. Software must determine whether the PMU is attached to an ARMv8 processor by using the PMDEVAFF0 and PMDEVAFF1 registers to discover the affinity of the PMU to any ARMv8 processors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D5.1.1 Time as measured by the Performance Monitors cycle counter

The Performance Monitors cycle counter, accessed through PMCCNTR_EL0 or PMCCNTR, increments from the hardware processor clock, not PE clock cycles.

The relationship between the count recorded by the Performance Monitors cycle counter and the passage of real time is IMPLEMENTATION DEFINED.

**Note**

- This means that, in an implementation where PEs are multi-threaded, the counter continues to increment across all PEs, rather than only counting cycles for which the current PE is active.
- Although the architecture requires that direct reads of PMCCNTR_EL0 or PMCCNTR occur in program order, there is no requirement that the count increments between two such reads. Even when the counter is incrementing on every clock cycle, software might need check that the difference between two reads of the counter is nonzero.

  The architecture requires that an indirect write to the PMCCNTR_EL0 or PMCCNTR is observable to direct reads of the register in finite time. The counter increments from the hardware processor clock are indirect writes to these registers.

### D5.1.2 Interaction with trace

It is IMPLEMENTATION DEFINED whether the implementation exports counter events to a Trace extension, or other external monitoring agent, to provide triggering information. The form of any exporting is also IMPLEMENTATION DEFINED. If implemented, this exporting might be enabled as part of the performance monitoring control functionality.

ARM recommends system designers include a mechanism for importing a set of external events to be counted, but such a feature is IMPLEMENTATION DEFINED. When implemented, this feature enables the Trace extension to pass in events to be counted.

### D5.1.3 Interaction with power saving operations

All counters are subject to any changes in clock frequency, including clock stopping caused by the `WFI` and `WFE` instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D5.2 Accuracy of the Performance Monitors

The Performance Monitors:

- Are a non-invasive debug component. See *Non-invasive behavior*.
- Must provide broadly accurate and statistically useful count information.

However, the Performance Monitors allow for:

- A reasonable degree of inaccuracy in the counts to keep the implementation and validation cost low. See *A reasonable degree of inaccuracy* on page D5-1860.
- IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to put the PE in an operating state that might do one or both of the following:
  - Change the level of non-invasiveness of the Performance Monitors so that enabling an event counter can impact the performance or behavior of the PE.
  - Allow inaccurate counts. This includes, but is not limited to, cycle counts.

### D5.2.1 Non-invasive behavior

The Performance Monitors are a non-invasive debug feature. A non-invasive debug feature permits the observation of data and program flow. Performance Monitors, Sample-based Profiling and Trace are non-invasive debug features.

Non-invasive debug components do not guarantee that they do not make any changes to the behavior or performance of the processor. Any changes that do occur must not be severe however, as this will reduce the usefulness of event counters for performance measurement and profiling. This does not include any change to program behavior that results from the same program being instrumented to use the Performance Monitors, or from some other performance monitoring process being run concurrently with the process being profiled in a multi-tasking operating system. As such, a reasonable variation in performance is permissible.

**Note**

Power consumption is one measure of performance. Therefore, a reasonable variation in power consumption is permissible.

ARM does not define a *reasonable variation in performance*, but recommends that such a variation is kept within 5% of normal operating performance, when averaged across a suite of code that is representative of the application workload.

**Note**

For profiles other than A-profile, there is the potential for stronger requirements. Ultimately, performance requirements are determined by end-users, and not set by the architecture.

For some common architectural events, this requirement to be non-invasive can conflict with the requirement to present an accurate value of the count under normal operating conditions. Should an implementation require more performance-invasive techniques to accurately count an event, there are the following options:

- If the event is optional, define an alternative IMPLEMENTATION DEFINED event that accurately counts the event and document the impact on performance of enabling the event.
- Provide an IMPLEMENTATION DEFINED control that disables accurate counting of the event to restore broadly accurate performance, and document the impact on performance of accurate counting.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D5.2.2 A reasonable degree of inaccuracy

The Performance Monitors provide broadly accurate and statistically useful count information. To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the counts is acceptable. ARM does not define a *reasonable degree of inaccuracy* but recommends the following guidelines:

- Under normal operating conditions, the counters must present an accurate value of the count.
- In exceptional circumstances, such as a change in Security state or other boundary condition, it is acceptable for the count to be inaccurate.
- Under very unusual, non-repeating pathological cases, the counts can be inaccurate. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in the count is very unlikely.

**Note**

An implementation must not introduce inaccuracies that can be triggered systematically by the execution of normal pieces of software. For example, it is not reasonable for the count of branch behavior to be inaccurate when caused by a systematic error generated by the loop structure producing a dropping in branch count.

However, dropping a single branch count as the result of a rare interaction with an interrupt is acceptable.

The permitted inaccuracy limits the possible uses of the Performance Monitors. In particular, the architecture does not define the point in a pipeline where the event counter is incremented, relative to the point where a read of the event counters is made. This means that pipelining effects can cause some imprecision.

A change of Security state can also affect the accuracy of the Performance Monitors, see *Interaction with EL3* on page D5-1864.

In addition to this, entry to and exit from Debug state can disturb the normal running of the PE, causing further inaccuracy in the Performance Monitors. Disabling the counters while in Debug state limits the extent of this inaccuracy. An implementation can employ methods to limit this inaccuracy, for example by promptly disabling the counters during the Debug state entry sequence.

An implementation must document any particular scenarios where significant inaccuracies are expected.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D5.3 Behavior on overflow

All events are counted in 32-bit wrapping counters, that overflow when they wrap. The cycle counter, PMCCNTR, is a 64-bit wrapping counter, that is configured by PMCR.LC to either:

- Signal an overflow when bit PMCCNTR[63] overflows.
- Signal an overflow when bit PMCCNTR[31] overflows into bit PMCCNTR[32].

On a Performance Monitors counter overflow:

- An overflow status bit is set to 1. See PMOVSCLR.
- An interrupt request is generated if the PE is configured to generate counter overflow interrupts. For more information, see *Generating overflow interrupt requests*.
- The counter continues counting events.

### D5.3.1 Generating overflow interrupt requests

Software can program the Performance Monitors so that an overflow interrupt request is generated when a counter overflows. See *PMINTENSET* on page K13-5710 and *PMINTENCLR* on page K13-5710.

The overflow interrupt request is a level-sensitive request.

**Note**

- The mechanism by which an interrupt request from the Performance Monitors generates an FIQ or IRQ exception is IMPLEMENTATION DEFINED.
- ARM recommends that the overflow interrupt requests:
  - Translate into a **PMUIRQ** signal, so that they are observable to external devices.
  - Connect to inputs on an IMPLEMENTATION DEFINED generic interrupt controller as a *Private Peripheral Interrupt* (PPI) for the originating processor. See the *ARM Generic Interrupt Controller Architecture Specification* for information about PPIs.
  - Connect to a *Cross Trigger Interface* (CTI), see Chapter H5 *The Embedded Cross-Trigger Interface*.
- ARM strongly discourages implementations from connecting overflow interrupt requests from multiple PEs to the same *System Peripheral Interrupt* (SPI) identifier.
- From GICv3, the *ARM® Generic Interrupt Controller Architecture Specification* recommends that the *Private Peripheral Interrupt* (PPI) with ID 23 is used for overflow interrupt requests.

---

Counters overflow when counting one or more events generates an unsigned carry out. Software can write to the counters to control the frequency at which interrupt requests occur. For counters other than the cycle counter, the counter is always a 32-bit unsigned wrapping value. For example, software might set a counter to `0xFFFF0000`, to generate another counter overflow after 65 536 increments, and reset it to this value every time an overflow interrupt occurs.

**Note**

If an event can occur multiple times in a single clock cycle then counter overflow can occur without the counter registering a value of zero.

---

For the cycle counter, software can program PMCR.LC to treat the counter as either a 64-bit or a 32-bit unsigned value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The overflow interrupt request is a level-sensitive request. The PE signals a request for:

- Any given PMN*x* counter, when the value of PMOVSSET[*x*] is 1, the value of PMINTENSET[*x*] is 1, and one of the following is true:
  - EL2 is not implemented and the value of PMCR.E is 1.
  - EL2 is implemented, x is less than the value of HDCR.HPMN, and the value of PMCR.E is 1.
  - EL2 is implemented, x is greater than or equal to the value of HDCR.HPMN, and the value of HDCR.HPME is 1.
- The cycle counter, when the values of PMOVSSET[31], PMINTENSET[31], and PMCR.E are all 1.

The overflow interrupt request is active in both Secure and Non-secure states. In particular, if EL3 and EL2 are both implemented, overflow events from PMN*x* where *x* is greater than or equal to the value of HDCR.HPMN can be signaled from all modes and states but only if the value of HDCR.HPME is 1.

The interrupt handler for the counter overflow request must cancel the interrupt request, by writing to PMOVSCLR[*x*] to clear the overflow bit to 0.

### Pseudocode description of overflow interrupt requests

See Chapter J1 *ARMv8 Pseudocode* for a pseudocode description of overflow interrupt requests. The `AArch64.CheckForPMUOverflow()` and `AArch32.CheckForPMUOverflow()` pseudocode functions signal PMU overflow interrupt requests to an interrupt controller and PMU overflow trigger events to the cross-trigger interface.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D5.4 Attributability

An event caused by the PE counting the event is Attributable. If an agent other than the PE that is counting the events causes an event, these events are Unattributable.

An event is defined as being either Attributable or Unattributable. If the event is Attributable, it is further defined whether it is Attributable to:

- The current Security state of the PE.
- The current Exception level of the processor.
- When the PE is in Debug state, operations issued to the PE by the debugger through the external debug interface.

In a multi-threaded implementation, an event might be Attributable either to the current Exception level alone, or to both the Exception level and the Security state of another processor with the same values for affinity level 1 and higher.

**Note**

A multi-threaded implementation is one where the value of MPIDR.MT is 1. In this section, when referring to a multi-threaded implementation, *thread* is used to mean processing elements with different affinity level 0 values and the same values for affinity level 1 and higher.

An event can be defined as the combination of multiple subevents, which can be either Attributable or Unattributable.

All architecturally defined events are Attributable.

Unattributable events might be counted when Attributable events are not counted. See:

- *Interaction with EL3* on page D5-1864.
- *Event filtering* on page D5-1866.
- *Performance Monitors and Debug state* on page D5-1868.

These sections are summarized by Table D5-1 for events Attributable to the processor, and Unattributable events.

**Table D5-1 Counting events**

| Counter and PMU enabled | State | Allowed or prohibited | Filtered | Event type | | |
|---|---|---|---|---|---|---|
| | | | | If Attributable to: | Then | Else |
| Yes | Non-debug | Allowed | Not filtered | X | Count | Count |
| | | | Filtered | Current Exception level | Do not count | IMPLEMENTATION DEFINED |
| | | Prohibited | X | Current Security state | Do not count | IMPLEMENTATION DEFINED |
| | Debug | X | X | Debugger operations or raw cycles | Do not count | IMPLEMENTATION DEFINED |
| No | X | X | X | X | Do not count | Do not count |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D5.5 Effect of EL3 and EL2

This section describes the effects of implementing EL3 and EL2 on the Performance Monitors. It contains the following subsections:

- *Interaction with EL3*.
- *Interaction with EL2* on page D5-1865.

### D5.5.1 Interaction with EL3

While counting events is never prohibited in Non-secure state, there are some restrictions on counting events in Secure state. From reset, counting events Attributable to Secure state is prohibited in Secure state. When executing in AArch32 state, software can set SDCR.SPME to 1 to permit event counting in Secure state. In AArch64 state, software can set MDCR_EL3.SPME to 1 to permit event counting in Secure state.

**Note**

This enables a Secure Monitor to permit profiling within Secure state without having to configure an IMPLEMENTATION DEFINED debug authentication interface.

The system can use the external authentication interface to override SPME. For example, if **SPNIDEN** and **NIDEN** are HIGH then this permits event counting in Secure state, irrespective of the value in SDCR.SPME or MDCR_EL3.SPME.

If EL3 is not implemented, the behavior is as if the value of SDCR.SPME or MDCR_EL3.SPME is 1, as appropriate.

In summary, counting Attributable events in Secure state is prohibited unless any one of the following is true:

- EL3 is not implemented.
- EL3 is implemented, is using AArch64, and the value of MDCR_EL3.SPME is 1.
- EL3 is implemented, is using AArch32, and the value of SDCR.SPME is 1.
- EL3 is implemented, EL3 or EL1 is using AArch32, executing at EL0, and the value of SDER32_EL3 is 1.
- EL3 is implemented, and counting is permitted by an IMPLEMENTATION DEFINED authentication interface, `ExternalSecureNoninvasiveDebugEnabled()` == TRUE.

  **Note**

  Software can read the Authentication Status register, DBGAUTHSTATUS, to determine the state of an IMPLEMENTATION DEFINED authentication interface.

The cycle counter, PMCCNTR, counts even when event counting is prohibited, unless PMCR.DP is set to 1 or the PE is in Debug state.

For each Unattributable event it is IMPLEMENTATION DEFINED whether it is counted when counting Attributable events is prohibited.

**Note**

- Additional controls in PMCR, HDCR, PMCNTENSET, and PMCNTENCLR can also disable the event counters and the cycle counter.
- Controls in PMEVTYPER<n> and PMCCFILTR can filter out events based on Exception level and Security state.

This disabling of counters or filtering of events takes precedence over the authentication controls.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See AArch64.ProfilingProhibited(), AArch64.CountEvents(), AArch32.ProfilingProhibited(), and AArch32.CountEvents() in Chapter J1 *ARMv8 Pseudocode* for more information. The CountEvents() function returns TRUE if PMN*x* counts events in the current mode and state. However, this function does not completely describe the behavior for Unattributable events.

If an implementation is multi-threaded, the value of PMEVTYPER<n>.MT ==1, and an event is Attributable to Secure state on another thread, then the event is not counted when counting events Attributable to Secure state in Secure state is prohibited on the counting thread.

**Example D5-1 The effect of having PMEVTYPER<n>.MT==1**

If the value of MDCR_EL3.SPME is 0 on one thread, then it does not count events Attributable to Secure state on another thread, even if one or both of the following applies:

- This thread is in Non-secure state.
- MDCR_EL3.SPME==1 on the other thread.

Otherwise:

- When the current configuration prohibits counting of events Attributable to Secure state in Secure state, it is IMPLEMENTATION DEFINED whether:
  - Counting events Attributable to Secure state on this PE in Non-secure state is permitted.
  - Counting Unattributable events related to other secure operations in the system is permitted.
- Otherwise, counting events in Non-secure state is permitted.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether counting the event is permitted when counting of Attributable events is prohibited.

In AArch32 state, the Performance Monitors registers are Common registers, see *Classification of System registers* on page G4-4122.

The Performance Monitors registers are always accessible regardless of the values of the authentication signals and the SDER.SUNIDEN bit. Authentication controls whether the counters count events, it does not control access to the Performance Monitors registers.

The Performance Monitors are intended to be broadly accurate and statistically useful, see *Accuracy of the Performance Monitors* on page D5-1859. Some inaccuracy is permitted at the point of changing Security state, however. To avoid the leaking of information from the Secure state, the permitted inaccuracy is that transactions that are not prohibited can be uncounted. Where possible, prohibited transactions must not be counted, but if they are counted, then that counting must not degrade security.

### D5.5.2 Interaction with EL2

In an implementation that includes EL2, Non-secure software executing at EL2 can:

- Trap any attempt by the Guest OS to access the PMU. This means the hypervisor can identify which Guest OSs are using the PMU and intelligently employ switching of the PMU state.
- Trap accesses to the PMCR, so that it can fully virtualize the PMU identity registers, PMCR.IMP and PMCR.IDCODE.
- Reserve the highest-numbered counters for its own use by overriding the value of PMCR.N seen by the Guest OS. The implementation must not permit a Guest OS to access the reserved counters.

HDCR controls Performance Monitors virtualization.

For more information see:

- *Counter enables* on page D5-1869.
- *Counter access* on page D5-1870.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D5.6 Event filtering

The PMU can filter events by various combinations of Exception level and Security state. This gives software the flexibility to count events across multiple processes.

### D5.6.1 Filtering by Exception level and state

For each event counter PMEVTYPER<n>_EL0 specifies the Exception levels in which the counter counts events Attributable to Exception levels.

PMCCFILTR_EL0 specifies the Exception levels in which the cycle counter counts.

If the value of PMEVTYPER<n>_EL0 is 1 and an event is Attributable to another thread, then the implementation is multi-threaded and this filtering only applies using the current Exception level of the other thread, regardless of the Exception level of the counting thread. For example, if the value of PMEVTYPER<n>_EL0 is 0 on one thread, then it does not count events Attributable to Secure state on another thread, even if the first thread is in Non-secure state and the value of MDCR_EL3.SPME is 1 on another thread.

Otherwise, for each Unattributable event, it is IMPLEMENTATION DEFINED whether the filtering applies.

For more information, see the individual register descriptions.

### D5.6.2 Accuracy of event filtering

The PMU architecture does not require event filtering to be accurate.

For most events, it is acceptable that, during a transition between states, events generated by instructions executed in one state are counted in the other state. The following sections describe the cases where event counts must not be counted in the wrong state:

- *Exception-related events*.
- *Software increment events*.

#### Exception-related events

The PMU must filter events related to exceptions and exception handling according to the Exception level from which the exception was taken. These events are:

- Exception taken.
- Instruction architecturally executed, condition code check pass, exception return.
- Instruction architecturally executed, condition code check pass, write to CONTEXTIDR.
- Instruction architecturally executed, condition code check pass, write to translation table base.

The PMU must not count an exception after it has been taken because this could systematically report a result of zero exceptions at EL0. Similarly, it is not acceptable for the PMU to count exception returns or writes to CONTEXTIDR after the return from the exception.

**Note**

Unprivileged software cannot write to CONTEXTIDR.

#### Software increment events

The PMU must filter software increment events according to the Exception level in which the software increment occurred. Software increment counting must also be precise, meaning the PMU must count every architecturally executed software increment event, and must not count any speculatively executed software increment.

Software increment events must also be counted without the need for explicit synchronization. For example, two software increments executed without an intervening context synchronization operation must increment the event counter twice.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Pseudocode description of event filtering

See `AArch64.CountEvents()` and `AArch32.CountEvents()` in Chapter J1 *ARMv8 Pseudocode* for a pseudocode description of event filtering. However, this function does not completely describe the behavior for Unattributable events.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D5.7 Performance Monitors and Debug state

Events that count cycles are not counted in Debug state.

Events Attributable to the operations issued by the debugger through the external debug interface are not counted in Debug state.

If the implementation is multi-threaded and the value of PMEVTYPER<n>_EL0 is 1, and the event is Attributable to an operation issued by the debugger through the external debug interface to another thread that is in Debug state, then the event is not counted, and it is IMPLEMENTATION DEFINED whether the event is counted when the counting thread is in Debug state.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether it is counted when the counting processor is in Debug state. If the event might be counted, then the rules in *Filtering by Exception level and state* on page D5-1866 apply for the current Security state in Debug state.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D5.8 Counter enables

Table D5-2 shows an implementation that does not include EL2, and where the PMCR.E bit is a global counter enable bit, and PMCNTENSET provides an enable bit for each counter.

**Table D5-2 Event counter enables when an implementation does not include EL2**

| PMCR.E | PMCNTENSET[x] == 0 | PMCNTENSET[x] == 1 |
|---|---|---|
| 0 | PMN*x* disabled | PMN*x* disabled |
| 1 | PMN*x* disabled | PMN*x* enabled |

If the implementation includes EL2, then in addition to the PMCR.E and PMCNTENSET enable bits:

- HDCR.HPME overrides the value of PMCR.E for counters configured for access in Hyp mode.
- HDCR.HPMN specifies the number of performance counters that the Guest OS can access. The minimum permitted value of HDCR.HPMN is 1, meaning there must be at least one counter that the Guest OS can access.

Table D5-3 shows the combined effect of all the counter enable controls.

**Table D5-3 Event counter enables when an implementation includes EL2**

| HDCR.HPME | PMCR.E | PMCNTENSET[x] == 0 | PMCNTENSET[x] == 1 | |
|---|---|---|---|---|
| | | | x < HDCR.HPMN | x ≥ HDCR.HPMN |
| 0 | 0 | PMN*x* disabled | PMN*x* disabled | PMN*x* disabled |
| 0 | 1 | PMN*x* disabled | PMN*x* enabled | PMN*x* disabled |
| 1 | 0 | PMN*x* disabled | PMN*x* disabled | PMN*x* enabled |
| 1 | 1 | PMN*x* disabled | PMN*x* enabled | PMN*x* enabled |

**Note**

The effect of HDCR.{HPME, HPMN} on the counter enables applies in both Security states. However, in Secure state the value returned for PMCR.N is not affected by HDCR.HPMN.

EL2 does not affect the enabling of PMCCNTR. Table D5-4 shows the PMCCNTR enables, for all implementations.

**Table D5-4 Cycle counter enables**

| PMCR.E | PMCNTENSET[31] == 0 | PMCNTENSET[31] == 1 |
|---|---|---|
| 0 | PMCCNTR disabled | PMCCNTR disabled |
| 1 | PMCCNTR disabled | PMCCNTR enabled |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D5.9 Counter access

All counters are accessible in EL3, Secure EL1 and EL2. If EL2 is implemented the hypervisor uses HDCR.HPMN to reserve an event counter, with the effect that software cannot access that counter and its associated state from Non-secure EL1 modes or from Non-secure EL0.

**Note**

This section describes a counter as being accessible from a particular Exception level and state. However, access to the registers are subject to the access permissions described in *Access permissions* on page D5-1893. In particular, accesses from EL0 might be UNDEFINED and accesses from Non-secure EL1 and EL0 might be trapped to EL2.

### D5.9.1 Access at EL0

To allow self-profiling code executing at EL0 to make use of the Performance Monitors, three controls are provided in PMUSERENR_EL0.

For more information, see *Traps to EL1 of EL0 accesses to Performance Monitors registers* on page D1-1582.

### D5.9.2 PMN*x* event counters

For an implementation that includes EL2 and EL3, Table D5-5 shows how the values of the HDCR.HPMN field control the behavior of accesses to the PMN*x* event counter registers.

**Note**

Access to the Performance Monitors registers is also subject to the access permissions described in *Access permissions* on page D5-1893. In particular, accesses might be trapped to EL1 or EL2.

**Table D5-5 Result of PMN*x* event counter accesses**

| Condition | Secure state | | | Non-secure state | | |
|---|---|---|---|---|---|---|
| | EL3 | EL1 | EL0 | EL2 | EL1 | EL0 |
| x < HDCR.HPMN | Succeeds | Succeeds | Succeeds | Succeeds | Succeeds | Succeeds |
| x ≥ HDCR.HPMN | Succeeds | Succeeds | Succeeds | Succeeds | No access | No access |

Where Table D5-5 shows no access:

- If PMSELR.SEL is *x* then:
  - A direct read of PMXEVTYPER or PMXEVCNTR is CONSTRAINED UNPREDICTABLE.
  - A direct write to PMXEVTYPER or PMXEVCNTR is CONSTRAINED UNPREDICTABLE.
- A direct read of PMEVTYPER<n> or PMEVCNTR<n> is CONSTRAINED UNPREDICTABLE.
- A direct write of PMEVTYPER<n> or PMEVCNTR<n> is CONSTRAINED UNPREDICTABLE.
- For direct reads and direct writes, PMOVSCLR[*x*], PMOVSSET[*x*], PMCNTENSET[*x*], PMCNTENCLR[*x*], PMINTENSET[*x*], and PMINTENCLR[*x*] are RAZ/WI
- Direct writes to PMSWINC[*x*] are ignored.
- A direct write of 1 to PMCR.P does not reset PMN*x*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

In Secure state, and in the Non-secure EL2 mode, the value of HDCR.HPMN does not affect the value returned for PMCR.N.

### D5.9.3 CCNT cycle counter

The PMU does not provide any control that a hypervisor can use to reserve the cycle counter for its own use. The only control over the cycle counter is an access permission control for EL0. See *Access permissions* on page D5-1893.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D5.10 Events, Event numbers and mnemonics

The following sections describe the events that can be counted and their associated event numbers, and the mnemonics for the events:

- *Definitions*.
- *Common event numbers* on page D5-1875.
- *Common architectural event numbers* on page D5-1877.
- *Common microarchitectural event numbers* on page D5-1881.
- *Relationship between REFILL events and associated access events* on page D5-1890.
- *Required events* on page D5-1891.
- *IMPLEMENTATION DEFINED event numbers* on page D5-1891.

### D5.10.1 Definitions

The following subsections give more information about terms used in the event definitions:

- *Definition of terms*.
- *Levels of caches and TLBs* on page D5-1875.

#### Definition of terms

**Speculatively executed**

Many events relate to speculatively executed operations. Here, speculatively executed means the PE did some work associated with one or more instructions but the instructions were not necessarily architecturally executed.

An instruction might create one or more *microarchitectural operations* (µ-ops) at any point in the execution pipeline. For the purpose of event counting, the µ-ops are counted. The definition of a µ-op is implementation specific. An architecture instruction might create more than one µ-op for each instruction. µ-ops might also be removed or merged in the execution stream, so an architecture instruction might create no µ-ops for an instruction. Any arbitrary translation of instructions to an equivalent sequence of µ-ops is permitted.

This means there is no architecturally guaranteed relationship between a speculatively executed µ-op and an architecturally executed instruction. The results of such an operation can also be discarded, if it transpires that the operation was not required, such as a mispredicted branch. Therefore, ARMv8-A defines these events as *operation speculatively executed*, where appropriate.

**Note**

The definition of *speculatively executed* does not mean only those operations that are executed speculatively and later abandoned, for example due to a branch misprediction or fault. That is, speculatively executed operations must count operations on both false and correct execution paths.

The counting of operations can indicate the workload on the PE. However, there is no requirement for operations to represent similar amounts of work, and direct comparisons between different microarchitectures are not meaningful.

For example, an implementation might split an A32 or T32 `LDM` instruction of six registers into six µ-ops, one for each load, and a seventh address-generation operation to determine the base address or writeback address. Also, for doubleword alignment, the six load µ-ops might combine into four operations, that is, a word load, two doubleword loads, and a second word load. This single instruction can then be counted as five, or possibly six, events:

- Four (Operation speculatively executed - Load) events.
- One (Operation speculatively executed - Integer data processing) event.
- One (Operation speculatively executed - Software change of the PC) event, if the PC was one of the six registers in the `LDM` instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Different groups of events can have different IMPLEMENTATION DEFINED definitions of speculatively executed. Such groups share a common base type, which the event name denotes. Each of the events in the previous example are of the base type, operation speculatively executed.

For groups of events with a common base type, speculatively executed operations are all counted on the same basis, which normally means at the same point in the pipeline. It is possible to compare the counts and make meaningful observations about the program being profiled.

Within these groups, events are commonly defined with reference to a particular architecture instruction or group of instructions. In the case of speculatively executed operations this means operations with semantics that map to that type of instruction.

**Instruction memory access**

A PE acquires instructions for execution through instruction fetches. Instruction fetches might be due to:

- Fetching instructions that are architecturally executed.
- The result of the execution of an instruction preload instruction, `PLI`.
- Speculation that a particular instruction might be executed in the future.

The relationship between the fetch of an individual instruction and an instruction memory access is IMPLEMENTATION DEFINED. For example, an implementation might fetch many instructions including a non-integer number of instructions in a single instruction memory access.

**Memory-read operations**

A PE accesses memory through memory-read and memory-write operations. A memory-read operation might be due to:

- The result of an architecturally executed memory-reading instructions.
- The result of a speculatively executed memory-reading instructions.
- A translation table walk.

For levels of cache hierarchy beyond the Level 1 caches, memory-read operations also include accesses made as part of a refill of another cache closer to the PE. Such refills might be due to:

- Memory-read operations or memory-write operations that miss in the cache
- The execution of a data preload instruction.
- The execution of an instruction preload instruction on a unified cache.
- The execution of a cache maintenance instruction.

  **Note**

  A preload instruction or cache maintenance instruction is not, in itself, an access to that cache. However, it might generate cache refills which are then treated as memory-read operations beyond that cache.

- Speculation that a future instruction might access the memory location.

This list is not exhaustive.

The relationship between memory-read instructions and memory-read operations is IMPLEMENTATION DEFINED. For example, for some implementations an `LDP` instruction that reads two 64-bit registers might generate one memory-read operation if the address is quadword-aligned, but for other addresses it generates two or more memory-read operations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Memory-write operations**

Memory-write operations might be due to:

- The result of an architecturally executed memory-writing instructions.
- The result of a speculatively executed memory-writing instructions.

**Note**

Speculatively executed memory-writing instructions that do not become architecturally executed must not alter the architecturally defined view of memory. They can, however, generate a memory-write operation that is later undone in some implementation specific way.

For levels of cache hierarchy beyond the Level 1 caches, memory-write operations also include accesses made as part of a write-back from another cache closer to the PE. Such write-backs might be due to:

- Evicting a dirty line from the cache, to allocate a cache line for a cache refill, see memory-read operations.
- The execution of a cache maintenance instruction.

  **Note**

  A cache maintenance instruction is not in itself an access to that cache. However, it might generate write-backs which are then treated as memory-write operations beyond that cache.

- The result of a coherency request from another PE.

This list is not exhaustive.

The relationship between memory-writing instructions and memory-write operations is IMPLEMENTATION DEFINED. For example, for some implementations an `STP` instruction that writes two 64-bit registers might generate one memory-write operation if the address is quadword-aligned, but for other addresses it generates two or more memory-write operations. In some implementations, the result of two `STR` instructions that write to adjacent memory might be merged into a single memory-write operation.

**Note**

The data written back from a cache that is shared with other PEs might not be data that was written by the PE that performs the operation that leads to the write-back. Nevertheless, the event is counted as a write-back event for that PE.

**Instruction architecturally executed**

*Instruction architecturally executed* is a class of event that counts for each instruction of the specified type. Architecturally executed means that the program flow is such that the counted instruction would be executed in a *Simple sequential execution* of the program. Therefore an instruction that has been executed and retired is defined to be *architecturally executed*. When a PE can perform speculative execution, an instruction is not architecturally executed if the PE discards the results of the speculative execution.

If an instruction that would be executed in a *Simple sequential execution* of the program generates a synchronous exception, it is IMPLEMENTATION DEFINED whether the instruction is counted.

Each architecturally executed instruction is counted once, even if the implementation splits the instruction into multiple operations.Instructions that have no visible effect on the architectural state of the PE are architecturally executed if they form part of the architecturally executed program flow. The point where such instructions are retired is IMPLEMENTATION DEFINED.

Examples of instructions that have no visible effect are:

- A `NOP`.
- A conditional instruction that fails its condition code check.
- A Compare and Branch on Zero, `CBZ`, instruction that does not branch.
- A Compare and Branch on Nonzero, `CBNZ`, instruction that does not branch.

The point at which an event causes an event counter to be updated is not defined.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Unless otherwise stated, all instructions of the specified type are counted even if they have no visible effect on the architectural state of the PE. This includes a conditional instruction that fails its condition code check.

For events that count only the execution of instructions that update context state, such as writes to the CONTEXTIDR, if such an instruction is executed twice without an intervening context synchronization operation, it is CONSTRAINED UNPREDICTABLE whether the first instruction is counted.

**Note**

See *Context synchronization operation* for the definition of this term.

**Instruction architecturally executed, condition code check pass**

*Instruction architecturally executed, condition code check pass* is a class of events that explicitly do not occur for:

- A conditional instruction that fails its condition code check.
- A Compare and Branch on Zero, `CBZ`, instruction that does not branch.
- A Compare and Branch on Nonzero, `CBNZ`, instruction that does not branch.
- A Test and Branch on Zero, `TBZ`, instruction that does not branch.
- A Test and Branch on Nonzero, `TBNZ`, instruction that does not branch.
- A Store-Exclusive instruction that does not write to memory.

Otherwise, the definition of architecturally executed is the same as for *Instruction architecturally executed*.

### Levels of caches and TLBs

The mapping of different levels of cache or TLB to the PMU events is determined by the implementation. Although the CLIDR_EL1, or the AArch32 CLIDR, defines the implemented levels of cache, the architecture does provide any way of determining implemented levels of TLB. Also, many implementations include structures that provide some caching at a higher level than the level 1 caches or TLBs. Typically, these structures, that might be called Level 0 caches, or mini caches, or microcaches, are invisible to software. The implementation-specific nature of cache and TLB implementations mean that, in general, PMU event counts cannot be used reliably to make direct comparisons between different implementations.

## D5.10.2 Common event numbers

Table D5-6 lists the PMU architectural and microarchitectural event numbers in event number order. The entries in the *Event mnemonic* column link to the event description in *Common architectural event numbers* on page D5-1877 or *Common microarchitectural event numbers* on page D5-1881.

**Table D5-6 PMU event numbers**

| Event number | Event type | Event mnemonic | Description |
|---|---|---|---|
| `0x000` | Architectural | SW_INCR | Instruction architecturally executed, condition code check pass, software increment |
| `0x001` | Microarchitectural | L1I_CACHE_REFILL[a] | Level 1 instruction cache refill |
| `0x002` | Microarchitectural | L1I_TLB_REFILL[a] | Level 1 instruction TLB refill |
| `0x003` | Microarchitectural | L1D_CACHE_REFILL[a] | Level 1 data cache refill |
| `0x004` | Microarchitectural | L1D_CACHE | Level 1 data cache access |
| `0x005` | Microarchitectural | L1D_TLB_REFILL[a] | Level 1 data TLB refill |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D5-6 PMU event numbers (continued)**

| Event number | Event type | Event mnemonic | Description |
|---|---|---|---|
| 0x006 | Architectural | LD_RETIRED | Instruction architecturally executed, condition code check pass, load |
| 0x007 | Architectural | ST_RETIRED | Instruction architecturally executed, condition code check pass, store |
| 0x008 | Architectural | INST_RETIRED | Instruction architecturally executed |
| 0x009 | Architectural | EXC_TAKEN | Exception taken |
| 0x00A | Architectural | EXC_RETURN | Instruction architecturally executed, condition code check pass, exception return |
| 0x00B | Architectural | CID_WRITE_RETIRED | Instruction architecturally executed, condition code check pass, write to CONTEXTIDR |
| 0x00C | Architectural | PC_WRITE_RETIRED | Instruction architecturally executed, condition code check pass, software change of the PC |
| 0x00D | Architectural | BR_IMMED_RETIRED | Instruction architecturally executed, immediate branch |
| 0x00E | Architectural | BR_RETURN_RETIRED | Instruction architecturally executed, condition code check pass, procedure return |
| 0x00F | Architectural | UNALIGNED_LDST_RETIRED | Instruction architecturally executed, condition code check pass, unaligned load or store |
| 0x010 | Microarchitectural | BR_MIS_PRED | Mispredicted or not predicted branch speculatively executed |
| 0x011 | Microarchitectural | CPU_CYCLES | Cycle |
| 0x012 | Microarchitectural | BR_PRED | Predictable branch speculatively executed |
| 0x013 | Microarchitectural | MEM_ACCESS | Data memory access |
| 0x014 | Microarchitectural | L1I_CACHE | Level 1 instruction cache access |
| 0x015 | Microarchitectural | L1D_CACHE_WB | Level 1 data cache write-back |
| 0x016 | Microarchitectural | L2D_CACHE | Level 2 data cache access |
| 0x017 | Microarchitectural | L2D_CACHE_REFILL[a] | Level 2 data cache refill |
| 0x018 | Microarchitectural | L2D_CACHE_WB | Level 2 data cache write-back |
| 0x019 | Microarchitectural | BUS_ACCESS | Bus access |
| 0x01A | Microarchitectural | MEMORY_ERROR | Local memory error |
| 0x01B | Microarchitectural | INST_SPEC | Operation speculatively executed |
| 0x01C | Architectural | TTBR_WRITE_RETIRED | Instruction architecturally executed, condition code check pass, write to TTBR |
| 0x01D | Microarchitectural | BUS_CYCLES | Bus cycle |
| 0x01E | Architectural | CHAIN | For odd-numbered counters, increments the count by one for each overflow of the preceding even-numbered counter. For even-numbered counters there is no increment. |
| 0x01F | Microarchitectural | L1D_CACHE_ALLOCATE | Level 1 data cache allocation without refill |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D5-6 PMU event numbers (continued)**

| Event number | Event type | Event mnemonic | Description |
|---|---|---|---|
| 0x020 | Microarchitectural | L2D_CACHE_ALLOCATE | Level 2 data cache allocation without refill |
| 0x021 | Architectural | BR_RETIRED | Instruction architecturally executed, branch |
| 0x022 | Microarchitectural | BR_MIS_PRED_RETIRED | Instruction architecturally executed, mispredicted branch |
| 0x023 | Microarchitectural | STALL_FRONTEND | No operation issued due to the frontend |
| 0x024 | Microarchitectural | STALL_BACKEND | No operation issued due to backend |
| 0x025 | Microarchitectural | L1D_TLB | Level 1 data or unified TLB access |
| 0x026 | Microarchitectural | L1I_TLB | Level 1 instruction TLB access |
| 0x027 | Microarchitectural | L2I_CACHE | Attributable Level 2 instruction cache access |
| 0x028 | Microarchitectural | L2I_CACHE_REFILL[a] | Attributable Level 2 instruction cache refill |
| 0x029 | Microarchitectural | L3D_CACHE_ALLOCATE | Attributable Level 3 data or unified cache allocation without refill |
| 0x02A | Microarchitectural | L3D_CACHE_REFILL[a] | Attributable Level 3 data or unified cache refill |
| 0x02B | Microarchitectural | L3D_CACHE | Attributable Level 3 data or unified cache access |
| 0x02C | Microarchitectural | L3D_CACHE_WB | Attributable Level 3 data or unified cache write-back |
| 0x02D | Microarchitectural | L2D_TLB_REFILL[a] | Attributable Level 2 data or unified TLB refill |
| 0x02E | Microarchitectural | L2I_TLB_REFILL[a] | Attributable Level 2 instruction TLB refill |
| 0x02F | Microarchitectural | L2D_TLB | Attributable Level 2 data or unified TLB access |
| 0x030 | Microarchitectural | L2I_TLB | Attributable Level 2 instruction TLB access |

a. For more information, see *Relationship between REFILL events and associated access events* on page D5-1890.

The PMCEID0_EL0 and PMCEID1_EL0 registers identify the events that an implementation supports.

Future revisions of this manual, or of the architecture, might define additional common event numbers. Events that do not require additional features in the PMU can be implemented retrospectively, meaning such events can be supported as part of a PMUv3 implementation.

**Note**

This means that an ARMv7 PMUv2 implementation can include support for any of the event numbers defined in Table D5-6 on page D5-1875.

### D5.10.3 Common architectural event numbers

This section describes the defined common architectural event numbers.

For the common features, normally the counters must increment only once for each event. The event descriptions include any exceptions to this rule.

In these definitions, the term *architecturally executed* means that the instruction flow is such that the counted instruction would have been executed in a *Simple sequential execution* model.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The common architectural event numbers are:

**0x000, SW_INCR, Instruction architecturally executed, condition code check pass, software increment**

The counter increments on writes to the PMSWINC register.

If the PE performs two architecturally executed writes to the PMSWINC register without an intervening context synchronization operation, then the counter is incremented twice.

If PMEVTYPER<n>_EL0.evtCount is set to 0x000, then in AArch64 state, counts MSR writes [*n*] to PMSWINC_EL0 with bit [n] set to 1.

If the value of PMEVTYPER<n>_EL0.MT is 1 then, in a multi-threaded implementation, this counts writes by all PEs that have the same affinity at level 1 and above.

**0x006, LD_RETIRED, Instruction architecturally executed, condition code check pass, load**

The counter increments for every executed memory-reading instruction.

**Note**

Event 0x006 does not count the return status value of a Store-Exclusive instruction.

Whether the preload instructions PRFM, PLD, PLDW, PLI, count as memory-reading instructions is IMPLEMENTATION DEFINED. ARM recommends that if the instruction is not implemented as a NOP then it is counted as a memory-reading instruction.

**0x007, ST_RETIRED, Instruction architecturally executed, condition code check pass, store**

The counter increments for every executed memory-writing instruction.

DC ZVA is counted as a store.

The counter does not increment for a Store-Exclusive instruction that fails.

**0x008, INST_RETIRED, Instruction architecturally executed**

The counter increments for every architecturally executed instruction.

**0x009, EXC_TAKEN, Exception taken**

The counter increments for each exception taken. See *Exception-related events* on page D5-1866.

**Note**

The counter counts the PE exceptions described in:

- For exceptions taken to an Exception level using AArch64, *Exception entry* on page D1-1537.
- For exceptions taken to an Exception level using AArch32, *AArch32 state exception descriptions* on page G1-3811.

**0x00A, EXC_RETURN, Instruction architecturally executed, condition code check pass, exception return**

The counter increments for each executed exception return instruction. See also *Exception-related events* on page D5-1866.The following sections define the counted instructions:

- For an exception return to an Exception level using AArch64, *Exception return* on page D1-1552.
- For an exception return to an Exception level using AArch32, *Exception return to an Exception level using AArch32* on page G1-3796.

**0x00B, CID_WRITE_RETIRED, Instruction architecturally executed, condition code check pass, write to CONTEXTIDR**

The counter increments for every write to CONTEXTIDR. See *Exception-related events* on page D5-1866.

If the PE performs two architecturally-executed writes to CONTEXTIDR without an intervening context synchronization operation, it is CONSTRAINED UNPREDICTABLE whether the first write is counted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**0x00C, PC_WRITE_RETIRED, Instruction architecturally executed, condition code check pass, software change of the PC**

The counter increments for every software change of the PC. This includes all:

- Branch instructions.
- Memory-reading instructions that explicitly write to the PC.
- Data processing instructions that explicitly write to the PC.
- Exception return instructions, ERET and RET.

It is IMPLEMENTATION DEFINED whether the counter increments for any or all of:

- BRK and BKPT instructions.
- Undefined Instruction exceptions.
- The exception-generating instructions, SVC, HVC and SMC.

It is IMPLEMENTATION DEFINED whether an ISB is counted as a software change of the PC.

The counter does not increment for exceptions other than those explicitly identified in these lists.

**Note**

Conditional branches are only counted if the branch is taken.

**0x00D, BR_IMMED_RETIRED, Instruction architecturally executed, immediate branch**

The counter counts all immediate branch instructions that are architecturally executed.

In AArch32 state, the counter increments each time the PE executes one of the following instructions:

- B <label>.
- BL <label>.
- BLX <label>.
- CBZ <Rn>, <label>.
- CBNZ <label>.

In AArch64 state, the counter increments each time the PE executes an immediate branch instructions:

- B <label>.
- B.cond <label>.
- BL <label>.
- CBZ <Rn>, <label>.
- CBNZ <Rn>, <label>.
- TBZ <Rn>, <label>.
- TBNZ <Rn>, <label>.

**Note**

Conditional branches are always counted, regardless of whether the branch is taken.

If an ISB is counted as a software change of the PC instruction then it is IMPLEMENTATION DEFINED whether an ISB is counted as an immediate branch instruction.

**0x00E, BR_RETURN_RETIRED, Instruction architecturally executed, condition code check pass, procedure return**

In AArch32 state, the counter counts the following procedure return instructions:

- BX R14.
- MOV PC, LR.
- POP {…, PC}.
- LDR PC, [SP], #offset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The counter counts only the listed instructions as procedure returns. For example, it does not count the following as procedure return instructions:

- `BX R0`, because Rm != R14.
- `MOV PC, R0`, because Rm != R14.
- `LDM SP, {…, PC}`, because writeback is not specified.
- `LDR PC, [SP, #offset]`, because this specifies the wrong addressing mode.

In AArch64 state, the counter counts all architecturally executed `RET` instructions.

**`0x00F`, UNALIGNED_LDST_RETIRED, Instruction architecturally executed, condition code check pass, unaligned load or store**

The counter counts each memory-reading instruction or memory-writing instruction that accesses an unaligned address. It is IMPLEMENTATION DEFINED whether the counter also counts each Alignment fault Data Abort exception.

See *Unaligned data access* on page E2-2363 for more information.

**`0x01C`, TTBR_WRITE_RETIRED, Instruction architecturally executed, condition code check pass, write to TTBR**

The counter counts writes to TTBR0_EL1 and TTBR1_EL1 in AArch64 state and TTBR0 and TTBR1 in AArch32 state. When EL3 is implemented and using AArch32, this includes counting writes to both banked copies of TTBR0 and TTBR1. See *Exception-related events* on page D5-1866.

If the PE executes two writes to the same TTBR, without an intervening context synchronization operation, it is CONSTRAINED UNPREDICTABLE whether the first write to the TTBR, is counted.

If EL3 is implemented and using AArch64, the counter does not count writes to TTBR0_EL3.

If EL2 is implemented and using AArch64, the counter does not count writes to TTBR0_EL2 and to VTTBR_EL2.

If EL2 is implemented and using AArch32, the counter does not count writes to HTTBR and to VTTBR.

**`0x01E`, CHAIN**

For an odd-numbered counter, increments when an overflow occurs on the preceding even-numbered counter on the same PE. Even-numbered counters never increment as a result of this event. This means the CHAIN event links the odd-numbered counter with the preceding even-numbered counter to provide a 64-bit counter.

**Note**

- The CHAIN event means a system can provide *N* 32-bit counters, *N*/2 64-bit counters, or a mixture of 32-bit counters and 64-bit counters.
- The CHAIN event only counts overflows from the preceding even-numbered counter on the same PE. This means it is unaffected by the value of PMEVTYPER<n>_EL0.MT.

There is no atomic access to a pair of counters, so if software reads a counter-pair that is enabled, it must use a high-low-high read sequence, or employ reasonable heuristics, to avoid tearing. Similarly, if using CHAIN events, when disabling the counters software must take care that the result is not torn by the low counter overflowing at the same time as the counters are disabled Example D5-2 shows suitable sequences for disabling and enabling CHAIN counters.

**Example D5-2 Usage examples for 64-bit counters**

An example high-low-high read sequence for a 64-bit counter is:

```
    MRS W2,PMEVCNTR1_EL0   ;; read high counter, must be odd-numbered
retry:
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ISB                   ;; force ordering
MRS W0,PMEVCNTR0_EL0  ;; read low counter
                      ;; must return the previous counter to PMEVCNTR1_EL0
ISB                   ;; force ordering
MRS W1,PMEVCNTR1_EL0  ;; read high counter
CMP W1,W2
BNE retry             ;; if the high counter has changed, then retry
```

When disabling a pair of counters that are paired by a CHAIN event, software must:

1. Disable the low counter, by setting PMCNTENCLR_EL0[n] to 1.
   Typically, software uses a read-modify-write sequence to update PMCNTENCLR_EL0.
2. Execute an ISB instruction, or perform another context synchronization operation.
3. Disable the high counter, by setting PMCNTENCLR_EL0[n+1] to 1, or setting PMCR_EL0.E to 0.

When enabling a pair of counters that are paired by a CHAIN event, software must:

1. Enable the high counter, by setting PMCNTENCLR_EL0[n+1] to 0 and, if necessary, setting PMCR_EL0.E to 1.
2. Execute an ISB instruction, or perform another context synchronization operation.
3. Enable the low counter by setting PMCNTENCLR_EL0[n] to 0.

---

When using 64-bit counters, the architecture does not define the latency between the first counter overflowing and the second counter incrementing the CHAIN event. There is no requirement for updates to occur synchronously, but software reading or enabling the counter pair using a *low*-ISB-*high* sequence, as shown in Example D5-2 on page D5-1880, must not observe the low counter incrementing and overflowing for the event and the high counter not incrementing for the resulting CHAIN event. This means that the ISB executed after reading the low counter must ensure the completion of the update of the high counter by the CHAIN event.

**Note**

The way the CHAIN event operates means that, to filter the Exception levels and Security states in which the event is counted, software must:

- Program PMEVTYPER<n>_EL0 to count the event in the required conditions.
- Program PMEVTYPER<n+1>_EL0 to count the CHAIN event in all Exception levels and states.

---

**0x021, BR_RETIRED, Instruction architecturally executed, branch.**

The counter counts all branches on the architecturally executed path that would incur cost if mispredicted.

- Counts all branch instructions, memory-reading and data-processing instructions that explicitly write to the PC, at retirement.
- Counts both taken and not-taken branches.
- It is IMPLEMENTATION DEFINED whether this includes each of:
  - Unconditional direct branch instructions.
  - Exception-generating instructions.
  - Exception return instructions.
  - Context synchronization instructions.

### D5.10.4 Common microarchitectural event numbers

This section describes the defined common microarchitectural event numbers.

The common microarchitectural events are features that are likely to be implemented across a wide range of implementations. Unlike the common architectural events, there can be some IMPLEMENTATION DEFINED variation between definitions on different implementations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Unless otherwise stated, the common microarchitectural features relate only to events resulting from the operation of the PE counting the events. Events resulting from the operation of other PEs that might share a resource must not be counted. Where a resource can be subject to events that do not result from the operation of any of the PEs that share it, ARM recommends that the resource implements its own event counters. An example of a resource that might require its own event counters is a shared Level 2 cache that is subject to accesses from a system coherency port on that cache.

The event definitions relating to Level 2 caches generally assume the Level 2 cache is shared. The event definitions relating to Level 1 caches generally assume the Level 1 cache is not shared.

The common microarchitectural event numbers are:

**`0x001`, L1I_CACHE_REFILL, Level 1 instruction cache refill**

The counter counts instruction memory accesses that cause a refill of at least the Level 1 instruction or unified cache. This includes each instruction memory access that causes a refill from outside the cache. It excludes accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss.

A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

The counter does not count cache maintenance instructions.

See also *Relationship between REFILL events and associated access events* on page D5-1890.

**`0x002`, L1I_TLB_REFILL, Level 1 instruction TLB refill**

The counter counts instruction memory accesses that cause a TLB refill of at least the Level 1 instruction TLB. This includes each instruction memory access that causes an access to a level of memory system due to a translation table walk or an access to another level of TLB caching. It is IMPLEMENTATION DEFINED whether the count increments when:

- A refill results in a Translation fault.
- A refill is not allocated in the TLB.

The counter does not count:

- A TLB miss that does not cause a refill but does generate a translation table walk.
- TLB maintenance instructions.

See also *Relationship between REFILL events and associated access events* on page D5-1890.

**`0x003`, L1D_CACHE_REFILL, Level 1 data cache refill**

The counter counts each memory-read operation or memory-write operation that causes a refill of at least the Level 1 data or unified cache from outside the Level 1 cache. Each access to a cache line that causes a new linefill is counted, including those from instructions that generate multiple accesses, such as load or store multiples, and `PUSH` and `POP` instructions. In particular, the counter counts accesses to the Level 1 cache that cause a refill that is satisfied by another Level 1 data or unified cache, or a Level 2 cache, or memory.

A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

The counter does not count:

- Accesses that do not cause a new Level 1 cache refill but are satisfied from refilling data of a previous miss.
- Accesses to a cache line that generate a memory access but not a new linefill, such as write-through writes that hit in the cache.
- Cache maintenance instructions.
- A write that writes an entire line to the cache and does not fetch any data from outside the Level 1 cache, for example:
  - A write of a full cache line from a coalescing buffer.
  - A DC ZVA operation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A write that misses in the cache, and writes through the cache without allocating a line.

See also *Relationship between REFILL events and associated access events* on page D5-1890

**0x004, L1D_CACHE, Level 1 data cache access**

The counter counts each memory-read operation or memory-write operation that causes a cache access to at least the Level 1 data or unified cache. Each access to a cache line is counted including the multiple accesses of instructions, such as LDM or STM. Each access to other Level 1 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

The counter does not count cache maintenance instructions.

**0x005, L1D_TLB_REFILL, Level 1 data TLB refill**

The counter counts each memory-read operation or memory-write operation that causes a TLB refill of at least the Level 1 data or unified TLB. It counts each read or write that causes a refill, in the form of a translation table walk or an access to another level of TLB caching. It is IMPLEMENTATION DEFINED whether the count increments when:

- A refill results in a Translation fault.
- A refill is not allocated in the TLB.

The counter does not count:

- A TLB miss that does not cause a refill but does generate a translation table walk.
- TLB maintenance instructions.

See also *Relationship between REFILL events and associated access events* on page D5-1890

**0x010, BR_MIS_PRED, Mispredicted or not predicted branch speculatively executed**

The counter counts each correction to the predicted program flow that occurs because of a misprediction from, or no prediction from, the branch prediction resources and that relates to instructions that the branch prediction resources are capable of predicting.

If no program-flow prediction resources are implemented, ARM recommends that the counter counts all branches that are not taken.

**0x011, CPU_CYCLES, Cycle**

The counter increments on every cycle.

All counters are subject to changes in clock frequency, including when a WFI or WFE instruction stops the clock. This means that it is CONSTRAINED UNPREDICTABLE whether or not CPU_CYCLES continues to increment when the clocks are stopped by WFI and WFE instructions.

**Note**

Unlike PMCCNTR, this count is not affected by PMCR.DP, PMCR.D, or PMCR.C:

- The counter is not incremented in prohibited regions, so is not affected by PMCR.DP.
- The counter increments on every cycle, regardless of the setting of PMCR.D.
- The counter is reset when event counters are reset by PMCR.P, never by PMCR.C.

In a multi-threaded implementation, CPU_CYCLES counts each cycle for the multi-threaded processor, not only cycles for which this processor thread is active.

**0x012, BR_PRED, Predictable branch speculatively executed**

The counter counts every branch or other change in the program flow that the branch prediction resources are capable of predicting.

If all branches are subject to prediction, for example a BTB or BTAC, then all branches are predictable branches.

If branches are decoded before the predictor, so that the branch prediction logic dynamically predicts only some branches, for example conditional and indirect branches, then it is IMPLEMENTATION DEFINED whether other branches are counted as predictable branches. ARM recommends that all branches are counted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An implementation might include other structures that predict branches, such as a loop buffer that predicts short backwards direct branches as taken. Each execution of such a branch is a predictable branch. Terminating the loop might generate a misprediction event that is counted by BR_MIS_PRED.

If no program-flow prediction resources are implemented, this event is optional, but ARM recommends that BR_PRED counts all branches.

**`0x013`, MEM_ACCESS, Data memory access**

The counter counts memory-read or memory-write operations that the PE made. The counter increments whether the access results in an access to a Level 1 data or unified cache, a Level 2 data or unified cache, or neither of these.

The counter does not increment as a result of:

- Instruction memory accesses, see *Definition of terms* on page D5-1872.
- Translation table walks.
- Cache maintenance instructions.
- Write-back from any cache.
- Refilling of any cache.

**`0x014`, L1I_CACHE, Level 1 instruction cache access**

The counter counts instruction memory accesses that access at least the Level 1 instruction or unified cache. Each access to other Level 1 instruction memory structures, such as refill buffers, is also counted.

**`0x015`, L1D_CACHE_WB, Level 1 data cache write-back**

The counter counts every write-back of data from the Level 1 data or unified cache. The counter counts each write-back that causes data to be written from the Level 1 cache to outside of the Level 1 cache. For example, the counter counts the following cases:

- A write-back that causes data to be written to a Level 2 cache or memory.
- A write-back of a recently-fetched cache line that has not been allocated to the Level 1 cache.
- Transfer of data from the Level 1 cache to outside of this cache made as a result of a coherency request. The conditions determining which of these are counted for transfers to other Level 1 caches within the same multiprocessor cluster are IMPLEMENTATION DEFINED.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

Whether write-backs made as a result of cache maintenance instructions is IMPLEMENTATION DEFINED.

The counter does not count:

- The invalidation of a cache line without any write-back to a Level 2 cache or memory.
- Writes from the PE that write through the Level 1 cache to outside of the Level 1 cache.

An Unattributable write-back event occurs when a requestor outside the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted. See *Attributability* on page D5-1863.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

**`0x016`, L2D_CACHE, Level 2 data cache access**

The counter counts memory-read or memory-write operations, that the PE made, that access at least the Level 2 data or unified cache. Each access to a cache line is counted including refills of and write-backs from the Level 1 data, instruction, or unified caches. Each access to other Level 2 data or unified memory structures, such as refill buffers, write buffers, and write-back buffers, is also counted.

The counter does not count:

- Operations made by other PEs that share this cache.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Cache maintenance instructions.

**0x017, L2D_CACHE_REFILL, Level 2 data cache refill**

The counter counts memory-read or memory-write operations, that the PE made, that access at least the Level 2 data or unified cache and cause a refill of a Level 1 data, instruction, or unified cache or of the Level 2 data or unified cache. Each read from or write to the cache that causes a refill from outside the Level 1 and Level 2 caches is counted.

A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 2 cache that cause a refill that is satisfied by another Level 2 cache, a Level 3 cache, or memory.
- Refills of and write-backs from any Level 1 data, instruction or unified cache that cause a refill from outside the Level 1 and Level 2 caches.
- Accesses to the Level 2 cache that cause a refill of a Level 1 cache from outside of the Level 1 and Level 2 caches, even if there is no refill of the Level 2 cache.

The counter does not count, as events on this PE:

- Accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss.
- Accesses to the Level 2 cache that generate a memory access but not a new linefill, such as write-through writes that hit in the Level 2 cache.
- Accesses to the Level 2 cache that are part of a Level 1 cache refill or write-back that hit in the Level 2 cache so do not cause a refill from outside of the Level 1 and Level 2 caches.
- Operations made by other PEs that share this cache.
- Cache maintenance instructions.
- A write that writes an entire line to the cache and does not fetch any data from outside the Level 1 and Level 2 caches, for example:
  - A write-back from a Level 1 cache to a Level 2 cache.
  - A write from a coalescing buffer of a full cache line.
  - A `DC ZVA` operation.
- A write that misses in the cache, and writes through the cache without allocating a line.

See also *Relationship between REFILL events and associated access events* on page D5-1890.

**0x018, L2D_CACHE_WB, Level 2 data cache write-back**

The counter counts every write-back of data from the Level 2 data or unified cache that occurs as a result of an operation by this PE. It counts each write-back that causes data to be written from the Level 2 cache to outside the Level 1 and Level 2 caches. For example, the counter counts:

- A write-back that causes data to be written to a Level 3 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 2 cache.

Each write-back is counted once, even if it requires multiple accesses to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of data from the Level 2 cache to outside the Level 1 and Level 2 cache made as a result of a coherency request.
- Write-backs made as a result of Cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to a Level 3 cache or memory.
- Writes from the PE or Level 1 data or unified cache that write through the Level 2 cache to outside the Level 1 and Level 2 caches.
- Transfers of data from the Level 2 cache to a Level 1 cache, to satisfy a Level 1 cache refill.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An Unattributable write-back event occurs when a requestor outside the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted. See *Attributability* on page D5-1863.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

**`0x019`, BUS_ACCESS, Bus access**

The counter counts memory-read or memory-write operations that access outside of the boundary of the PE and its closely-coupled caches. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED. It must count accesses beyond the cache furthest from the PE for which accesses can be counted.

This means that:

- If Level 2 cache access events are implemented and no IMPLEMENTATION DEFINED events can count accesses for any caches outside a Level 2 cache, this counter increments for an access beyond the Level 2 cache.
- If Level 2 cache access events are not implemented and Level 1 cache access events are implemented, this counter increments for an access beyond the Level 1 cache.
- If neither Level 1 or Level 2 cache access events are implemented, this counter increments for all data accesses that the PE made.

The definition of a bus access is IMPLEMENTATION DEFINED but physically is a single beat rather than a burst. That is, for each bus cycle for which the bus is active.

Bus accesses include refills of and write-backs from Level 1 and Level 2 data, instruction, and unified caches. Whether bus accesses include operations that do use the bus but not explicitly transfer data, such as barrier operations, is IMPLEMENTATION DEFINED.

Where an implementation has multiple external buses, this event counts the sum of accesses across all buses.

If a bus supports multiple accesses per cycle, for example through multiple channels, the counter increments once for each channel that is active on a cycle, and so it might increment by more than one in any given cycle.

**`0x01A`, MEMORY_ERROR, Local memory error**

The counter counts every occurrence of a memory error signaled by a memory closely coupled to this PE. The definition of local memories is IMPLEMENTATION DEFINED but includes caches, tightly-coupled memories, and TLB arrays.

Memory error refers to a physical error detected by the hardware, such as a parity or ECC error. It includes errors that are correctable and those that are not. It does not include errors as defined in the architecture, such as MMU faults.

**`0x01B`, INST_SPEC, Operation speculatively executed**

The counter counts instructions that are speculatively executed by the PE. This includes instructions that are subsequently not architecturally executed. As a result, this event counts a larger number of instructions than the number of instructions architecturally executed. The definition of speculatively executed is IMPLEMENTATION DEFINED.

**`0x01D`, BUS_CYCLES, Bus cycle**

The counter increments on every cycle of the external memory interface of the PE.

**Note**

If the implementation clocks the external memory interface at the same rate as the processor hardware, the counter counts every cycle.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**0x01F, L1D_CACHE_ALLOCATE, Level 1 data cache allocation without refill**

The counter increments on every write that writes an entire line into the Level 1 cache without fetching from outside the Level 1 cache, for example:

- A write from a coalescing buffer of a full cache line.
- A DC ZVA operation.

**0x020, L2D_CACHE_ALLOCATE, Level 2 data cache allocation without refill**

The counter increments on every write that writes an entire line into the Level 2 cache without fetching from outside the Level 1 or Level 2 caches, for example:

- A write-back from a Level 1 to Level 2 cache.
- A write from a coalescing buffer of a full cache line.
- A DC ZVA operation.

**0x022, BR_MIS_PRED_RETIRED, Instruction architecturally executed, mispredicted branch**

The counter counts all instructions counted by BR_RETIRED that were not correctly predicted.

If no program-flow prediction resources are implemented, this event counts all retired not-taken branches.

**0x023, STALL_FRONTEND, No operation issued due to the frontend**

The counter counts every cycle on which no operation was issued because there are no operations available to issue from the frontend.

The division between frontend and backend is IMPLEMENTATION DEFINED. Frontend and backend events must count at the same point in the pipeline.

**Note**

- For a simplified pipeline model of Fetch → Decode → Issue → Execute → Retire, ARM recommends that the events are counted when instructions are dispatched from Decode to Issue.
- On a given cycle, both events might be counted if the backend is unable to accept any operations and there are no operations available to issue from the frontend.

**0x024, STALL_BACKEND, No operation issued due to the backend**

The counter counts every cycle on which no operation was issued because either:

- The backend is unable to accept any of the operations available for issue.
- The backend is unable to accept any operations.

For example, the back end might be unable to accept operations because of a resource conflict or non-availability.

The division between frontend and backend is IMPLEMENTATION DEFINED. Frontend and backend events must count at the same point in the pipeline. See *STALL_FRONTEND* for more information.

**0x025, L1D_TLB, Level 1 data or unified TLB access**

The counter counts each memory-read operation or memory-write operation that causes a TLB access to at least the Level 1 data or unified TLB. Each access to a TLB record is counted including the multiple accesses of instructions, such as LDM or STM.

The counter does not count TLB maintenance instructions.

**0x026, L1I_TLB, Level 1 instruction TLB access**

The counter counts each instruction memory access that causes a TLB access to at least the Level 1 instruction or unified TLB.

The counter does not count TLB maintenance instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**0x027, L2I_CACHE, Attributable Level 2 instruction cache access**

The counter counts attributable instruction memory accesses that access at least the Level 2 instruction or unified cache. Each Attributable access to other Level 2 instruction memory structures, such as refill buffers, is also counted.

See also *Attributability* on page D5-1863.

**0x028, L2I_CACHE_REFILL, Attributable Level 2 instruction cache refill**

The counter counts Attributable instruction memory accesses that cause a refill of at least the Level 2 instruction or unified cache. This includes each Attributable memory access that causes a refill from outside the cache. It excludes accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss.

A refill includes any Attributable access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

The counter does not count cache maintenance instructions.

See also:

- *Attributability* on page D5-1863.
- *Relationship between REFILL events and associated access events* on page D5-1890.

**0x029, L3D_CACHE_ALLOCATE, Attributable Level 3 data cache allocation without refill**

The counter increments on every attributable write that writes an entire line into the Level 3 cache without fetching from outside the Level 1, Level 2, or Level 3 cache, for example:

- A write-back from a Level 2 to Level 3 cache.
- An write from a coalescing buffer of a full cache line.
- An `DC ZVA` operation.

**0x02A, L3D_CACHE_REFILL, Attributable Level 3 data cache refill**

The counter counts each memory-read operation or memory-write operation that the PE makes that accesses at least the Level 3 data or unified cache and causes a refill of a Level 1 or Level 2 instruction, data, or unified cache or of the Level 3 data or unified cache. Each read from or write to the cache that causes a refill from outside the Level 1, Level 2, and Level 3 caches is counted.

A refill includes any access that causes data to be fetched from outside the cache, even if the data is ultimately not allocated into the cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

The counter does not count as events on this PE:

- Accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss.
- Accesses to the Level 3 cache that generate a memory access but not a new linefill, such as write-through writes that hit in the Level 3 cache.
- Accesses to the Level 3 cache that are part of a Level 1 or Level 2 cache refill or write-back that hit in the Level 3 cache so do not cause a refill from outside of the Level 1, Level 2, and Level 3 caches.
- Operations made by other PEs that share this cache.
- Cache maintenance instructions.
- A write that writes an entire line to the cache and does not fetch any data from outside the Level 1, Level2, and Level 3 cache, for example:
  - A write-back from a Level 2 to Level 3 cache.
  - A write of a full cache line from a coalescing buffer.
  - A DC ZVA operation.
- A write that misses in the cache, and writes through the cache without allocating a line.

See also *Relationship between REFILL events and associated access events* on page D5-1890.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**0x02B, L3D_CACHE, Attributable Level 3 data cache access**

The counter counts memory-read or memory-write operations, that the PE made, that access at least the Level 3 data or unified cache. Each access to a cache line is counted including refills of and write-backs from the Level 1 or Level 2 data, instruction, or unified caches. Each access to other Level 3 data or unified memory structures, such as refill buffers, write buffers, and write-back buffers, is also counted.

The counter does not count:

- Operations made by other PEs that share the Level 3 data or unified cache.
- Cache maintenance instructions.

**0x02C, L3D_CACHE_WB, Attributable Level 3 data cache write-back**

The counter counts every write-back of data from the Level 3 data or unified cache that occurs as a result of an operation by this PE. It counts each write-back that causes data to be written from the Level 3 cache to outside of the Level1, Level 2, and Level 3 caches. For example, the counter counts the following cases:

- A write-back that causes data to be written to a Level 4 cache, or to memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 3 cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED.whether the counter counts:

- A transfer of data from the Level 3 cache to outside the Level 1, Level 2, and Level 3 caches made as a result of a coherency request.
- A write-back made as a result of a Cache maintenance instruction.

The counter does not count:

- The invalidation of a cache line without any write-back to a Level  cache or memory.
- Writes from the PE, Level 1, or Level 2 data or unified cache, that write through the Level 3 cache to outside of the Level 3 cache.
- Transfers of data from the Level 3 cache to a Level 1 or Level 2 cache, to satisfy a Level 1 or Level 2 cache refill.

An Unattributable write-back event occurs when a requestor outside the PE makes a coherency request that results in write-back. If the cache is shared then Unattributable write-back events are not counted. If the cache is not shared then Unattributable write-back events are counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

**0x02D, L2D_TLB_REFILL, Attributable Level 2 data TLB refill**

The counter counts each Attributable memory-read operation or Attributable memory-write operation that causes a TLB refill of at least the Level 2 data or unified TLB. It counts each Attributable read or Attributable write that causes a refill, in the form of a translation table walk or an access to another level of TLB caching. It is IMPLEMENTATION DEFINED whether the count increments when:

- A refill results in a Translation fault.
- A refill is not allocated in the TLB.

The counter does not count:

- A TLB miss that does not cause a refill but does generate a translation table walk.
- TLB maintenance instructions.

See also:

- *Attributability* on page D5-1863.
- *Relationship between REFILL events and associated access events* on page D5-1890.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**0x02E, L2I_TLB_REFILL, Attributable Level 2 instruction TLB refill**

The counter counts Attributable instruction memory accesses that cause a TLB refill of at least the Level 2 instruction TLB. This includes each Attributable instruction memory access that causes an access to a level of memory system due to a translation table walk or an access to another level of TLB caching. It is IMPLEMENTATION DEFINED whether the count increments when:

- A refill results in a Translation fault.
- A refill is not allocated in the TLB.

The counter does not count:

- A TLB miss that does not cause a refill but does generate a translation table walk.
- TLB maintenance instructions.

See also:

- *Attributability* on page D5-1863.
- *Relationship between REFILL events and associated access events*.

**0x02F, L2D_TLB, Attributable Level 2 data or unified TLB access**

The counter counts each Attributable memory read operation or Attributable memory write operation that causes a TLB access to at least the Level 2 data or unified TLB. Each access to a TLB record is counted, including the multiple accesses of instructions such as LDM or STM.

The counter does not count TLB maintenance instructions.

See also *Attributability* on page D5-1863.

**0x030, L2I_TLB, Attributable Level 2 instruction TLB access**

The counter counts each Attributable memory read operation or Attributable memory write operation that causes a TLB access to at least the Level 2 instruction or unified TLB.

The counter does not count TLB maintenance instructions.

See also *Attributability* on page D5-1863.

### D5.10.5 Relationship between REFILL events and associated access events

CACHE_REFILL and TLB_REFILL events count the refills for accesses that are counted by the corresponding CACHE or TLB event. Table D5-7 shows this correspondence.

**Table D5-7 REFILL events and associated access events**

| REFILL event | Access event | Ratio REFILL/Access |
|---|---|---|
| 0x001 L1I_CACHE_REFILL | 0x014 L1I_CACHE | Level 1 instruction cache refill rate |
| 0x002 L1I_TLB_REFILL | 0x026 L1I_TLB | Level 1 instruction TLB refill rate |
| 0x003 L1D_CACHE_REFILL | 0x004 L1D_CACHE | Level 1 data or unified cache refill rate |
| 0x005 L1D_TLB_REFILL | 0x025 L1D_TLB | Level 1 data or unified TLB refill rate |
| 0x017 L2D_CACHE_REFILL | 0x016 L2D_CACHE | Level 2 data or unified cache refill rate |
| 0x028 L2I_CACHE_REFILL | 0x027 L2I_CACHE | Level 2 instruction cache refill rate |
| 0x02A L3D_CACHE_REFILL | 0x02B L3D_CACHE | Level 3 data or unified cache refill rate |
| 0x02D L2D_TLB_REFILL | 0x02F L2D_TLB | Attributable Level 2 data or unified TLB refill rate |
| 0x02E L2I_TLB_REFILL | 0x030 L2I_TLB | Attributable Level 2 instruction TLB refill rate |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D5.10.6 Required events

PMUv3 requires that an implementation includes the following common events:

- `0x000`, SW_INCR, Instruction architecturally executed, condition code check pass, software increment.
- `0x003`, L1D_CACHE_REFILL, Level 1 data cache refill.

  **Note**

  Event `0x003` is only required if the implementation includes a Level 1 data or unified cache.

- `0x004`, L1D_CACHE, Level 1 data cache access.

  **Note**

  Event `0x004` is only required if the implementation includes a Level 1 data or unified cache.

- `0x010`, BR_MIS_PRED, Mispredicted or not predicted branch speculatively executed.

  **Note**

  Event `0x010` is only required if the implementation includes program-flow prediction. However, ARM recommends that the event is implemented as described in *Common microarchitectural event numbers* on page D5-1881.

- `0x011`, CPU_CYCLES, Cycle.
- `0x012`, BR_PRED, Predictable branch speculatively executed.

  **Note**

  Event `0x012` is only required if the implementation includes program-flow prediction. However, ARM recommends that the event is implemented as described in *Common microarchitectural event numbers* on page D5-1881.

- At least one of:
  - `0x008`, INST_RETIRED, Instruction architecturally executed.
  - `0x01B`, INST_SPEC, Operation speculatively executed.

  **Note**

  ARM strongly recommends that event `0x008` is implemented.

### D5.10.7 IMPLEMENTATION DEFINED event numbers

For IMPLEMENTATION DEFINED event numbers, each counter is defined, independently, to either:

- Increment only once for each event.
- Count the duration for which an event occurs.

ARM recommends that implementers establish a standardized numbering scheme for their IMPLEMENTATION DEFINED events, with common definitions, and common count numbers, applied to all of their implementations. In general, the recommended approach is for standardization across implementations with common features. However, ARM recognizes that attempting to standardize the encoding of microarchitectural features across too wide a range of implementations is not productive.

ARM strongly recommends that at least the following classes of event are identified in the IMPLEMENTATION DEFINED events:

- Cumulative duration of stalls resulting from the holes in the instruction availability, separating out counts for key buffering points that might exist.
- Cumulative duration of data-dependent stalls, separating out counts for key dependency classes that might exist.
- Cumulative duration of stalls due to unavailability of execution resources, including, for example, write buffers, separating out counts for key resources that might exist.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Missed superscalar issue opportunities, if relevant, separating out counts for key classes of issue that might exist.
- Miss rates for different levels of caches and TLBs.
- Any external events passed to the PE through an IMPLEMENTATION DEFINED mechanism.
- Cumulative duration of a PSTATE.{A, I, F} interrupt mask set to 1.
- Any other microarchitectural features that the implementer considers are valuable to count.

The IMPLEMENTATION DEFINED event numbers are `0x040` to `0x3FF`. Appendix K3 *Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events* lists the ARM recommended standardized numbering scheme for these events.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D5.11 Performance Monitors Extension registers

The following section describes the Performance Monitors Extension registers.

The following subsections give general information about the Performance Monitors Extension registers, that apply for both Execution states:

- *Relationship between AArch32 and AArch64 Performance Monitors registers*.
- *Access permissions*.

*Performance Monitors Extension registers, functional group* on page G4-4173 summarizes the Performance Monitors Extension registers in AArch32 state.

*Instructions for accessing non-debug System registers* on page C5-278 summarizes the Performance Monitors Extension registers in AArch64 state.

### D5.11.1 Relationship between AArch32 and AArch64 Performance Monitors registers

Table K13-2 on page K13-5710 lists the Performance Monitors register names for AArch32 and AArch64 states.

### D5.11.2 Access permissions

Each Exception level is able to control Performance Monitors System register accesses at lower Exception levels. The access control flow is:

1. At EL0:
   - Writes to PMUSERENR are UNDEFINED.
   - Reads and writes of PMINTENSET and PMINTENCLR are UNDEFINED.
   - If PMUSERENR.EN == 0:
     - If PMUSERENR.SW == 0 then writes to PMSWINC are trapped to EL1.
     - If PMUSERENR.CR == 0 then reads of PMCCNTR are trapped to EL1.
     - If PMUSERENR.ER == 0 then reads of PMEVCNTR<n> and PMXEVCNTR, and reads and writes of PMSELR, are trapped to EL1.
     - Otherwise, for all other Performance Monitors registers, other than reads of PMUSERENR, reads and writes are trapped to EL1.

   **Note**

   If HCR.TGE==1, then all exceptions that would be taken to EL1 are instead taken to EL2.

2. Otherwise, at EL1 and EL0 in Non-secure state, if EL2 is implemented:
   - If HDCR.TPMCR == 1 then accesses to PMCR are trapped to EL2.
   - If HDCR.TPM == 1 then accesses to all Performance Monitors registers, including PMCR, are trapped to EL2.
3. Otherwise, at EL2, EL1 and EL0, if EL3 is implemented and using AArch64, and if MDCR_EL3.TPM == 1 then accesses to all Performance Monitors registers are trapped to EL3.

   **Note**

   These traps are not possible if EL3 is using AArch32.

4. Otherwise, the access is permitted.

**Note**

These traps and enables only apply to System register accesses using System register access instructions. For accesses through the optional memory-mapped or external debug interfaces, see *Access permissions for memory-mapped views of the Performance Monitors* on page I2-5113.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For details of the headings used in Table D5-8, see *Configurable instruction enables and disables, and trap controls* on page D1-1575. In addition, the following terms are used:

**Instruction** This shows the access instruction, read (`MRS`), write (`MSR`), or both (-). In AArch32 state, the equivalent instructions are `MRC` and `MCR`.

**Default access**

If the *Default access* is - then the access is trapped from EL0 to EL1 unless the PMUSERENR *enables* are set to 1.

**Resultant access permission**

This indicates the resulting access permission provided the enables at EL0 are enabled and the traps to EL2 or EL3 are disabled.

Table D5-8 shows the access permissions for System register accesses to the Performance Monitors registers.

**Table D5-8 Access permissions for the Performance Monitors System registers**

| Register | Instruction | At EL0: | | Traps from below to: | | Resultant access permission |
|---|---|---|---|---|---|---|
| | | Default access | PMUSERENR enables | EL2 | EL3[a] | |
| PMCR | - | - | EN | TPMCR or TPM | TPM | RW |
| PMCNTENSET | - | - | EN | TPM | TPM | RW |
| PMCNTENCLR | - | - | EN | TPM | TPM | RW |
| PMOVSCLR | - | - | EN | TPM | TPM | RW |
| PMSWINC | - | - | EN or SW | TPM | TPM | WO |
| PMSELR | - | - | EN or ER | TPM | TPM | RW |
| PMCEID0 | - | - | EN | TPM | TPM | RO |
| PMCEID1 | - | - | EN | TPM | TPM | RO |
| PMCCNTR | Read | - | EN or CR | TPM | TPM | RW |
| | Write | - | EN | | | |
| PMXEVTYPER | - | - | EN | TPM | TPM | RW |
| PMXEVCNTR | Read | - | EN or ER | TPM | TPM | RW |
| | Write | - | EN | | | |
| PMUSERENR | Read | RO | - | TPM | TPM | RW |
| | Write | UND | | | | |
| PMINTENSET | - | UND | - | TPM | TPM | RW |
| PMINTENCLR | - | UND | - | TPM | TPM | RW |
| PMOVSSET | - | - | EN | TPM | TPM | RW |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D5-8 Access permissions for the Performance Monitors System registers (continued)**

| Register | Instruction | At EL0: Default access | At EL0: PMUSERENR enables | Traps from below to: EL2 | Traps from below to: EL3[a] | Resultant access permission |
|---|---|---|---|---|---|---|
| PMEVCNTR<n> | Read | - | EN or ER | TPM | TPM | RW |
| | Write | - | EN | | | |
| PMEVTYPER<n> | - | - | EN | TPM | TPM | RW |
| PMCCFILTR | - | - | EN | TPM | TPM | RW |

a. Only if EL3 is using AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter D6
# The Generic Timer in AArch64 state

This chapter describes the implementation of the ARM Generic Timer. It includes an overview of the AArch64 System register interface to an ARM Generic Timer.

It contains the following section:

- *About the Generic Timer in AArch64 state* on page D6-1898.

Chapter G5 *The Generic Timer in AArch32 state* describes the AArch32 view of the Generic Timer, and Chapter I1 *System Level Implementation of the Generic Timer* describes the system level implementation of the Generic Timer.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D6.1 About the Generic Timer in AArch64 state

Figure D6-1 shows an example system-on-chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The *ARM Generic Interrupt Controller Architecture Specification* defines a possible architecture for the interrupt controllers.
- Generic Timer functionality is distributed across multiple components.

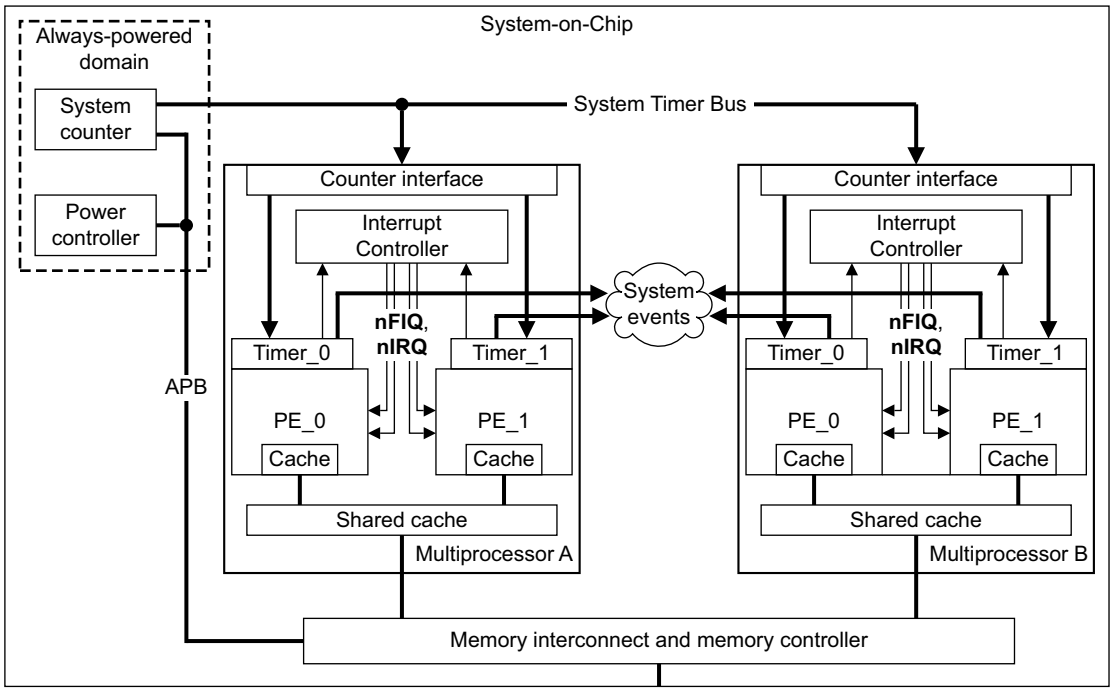



**Figure D6-1 Generic Timer example**

The Generic Timer:

- Provides a system counter, that measures the passing of time in real-time.

  **Note**

  The Generic Timer can also provide other components at a system level, but Figure D6-1 does not show any such components.

- Supports *virtual counters* that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.
- Timers, that can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

This chapter describes an instance of the Generic Timer component that Figure D6-1 shows as Timer_0 or Timer_1 within the Multiprocessor A or Multiprocessor B block. This component can be accessed from AArch64 state or AArch32 state, and this chapter describes access from AArch64 state. Chapter G5 *The Generic Timer in AArch32 state* describes access to this component from AArch32 state.

A Generic Timer implementation must also include a memory-mapped system component. This component:

- Must provide the System counter shown in Figure D6-1
- Optionally, can provide timer components for use at a system level.

Chapter I1 *System Level Implementation of the Generic Timer* describes this memory-mapped component.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D6.1.1 System counter

The Generic Timer provides a system counter with the following specification:

**Width** At least 56 bits wide.
The value returned by any 64-bit read of the counter is zero-extended to 64 bits.

**Frequency** Increments at a fixed frequency, typically in the range 1-50MHz.
Can support one or more alternative operating modes in which it increments by larger amounts at a lower frequency, typically for power-saving.

**Roll-over** Roll-over time of not less than 40 years.

**Accuracy** ARM does not specify a required accuracy, but recommends that the counter does not gain or lose more than ten seconds in a 24-hour period.
Use of lower-frequency modes must not affect the implemented accuracy.

**Start-up** Starts operating from zero.

The system counter must provide a uniform view of system time. More precisely, it must be impossible for the following sequence of events to show system time going backwards:

1. Device A reads the time from the system counter.
2. Device A communicates with another agent in the system, Device B.
3. After recognizing the communication from Device A, Device B reads the time from the system counter.

The system counter must be implemented in an always-on power domain.

To support lower-power operating modes, the counter can increment by larger amounts at a lower frequency. For example, a 10MHz system counter might either increment:

- By 1 at 10MHz.
- By 500 at 20kHz, when the system lowers the clock frequency, to reduce power consumption.

In this case, the counter must support transitions between high-frequency, high-precision operation, and lower-frequency, lower-precision operation, without any impact on the required accuracy of the counter.

The CNTFRQ_EL0 register is intended to hold a copy of the current clock frequency to allow fast reference to this frequency by software running on the PE. For more information see *Initializing and reading the system counter frequency*.

The mechanism by which the count from the system counter is distributed to system components is IMPLEMENTATION DEFINED, but each PE with a System register interface to the system counter must have a counter input that can capture each increment of the counter.

**Note**

So that the system counter can be clocked independently from the PE hardware, the count value might be distributed using a Gray code sequence. *Gray-count scheme for timer distribution scheme* on page I1-5110 gives more information about this possibility.

#### Initializing and reading the system counter frequency

The CNTFRQ_EL0 register must be programmed to the clock frequency of the system counter. Typically, this is done only during the system boot process, by using the System register interface to write the system counter frequency to the CNTFRQ_EL0 register. Only software executing at the highest implemented Exception level can write to CNTFRQ_EL0.

**Note**

The CNTFRQ_EL0 register is UNKNOWN at reset, and therefore the counter frequency must be set as part of the system boot process.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Software can read the CNTFRQ_EL0 register, to determine the current system counter frequency, in the following states:

- EL2.
- Secure and Non-secure EL1.
- When CNTKCTL_EL1.EL0PCTEN is set to 1, Secure and Non-secure EL0.

### Memory-mapped controls of the system counter

Some system counter controls are accessible only through the memory-mapped interface to the system counter. These controls are:

- Enabling and disabling the counter.
- Setting the counter value.
- Changing the operating mode, to change the update frequency and increment value.
- Enabling Halt-on-debug, that a debugger can then use to suspend counting.

For descriptions of these controls, see Chapter I1 *System Level Implementation of the Generic Timer*.

## D6.1.2 The physical counter

The PE includes a physical counter that contains the count value of the system counter. The CNTPCT_EL0 register holds the current physical counter value.

### Accessing the physical counter

Software with sufficient privilege can read CNTPCT_EL0 using a 64-bit System register read.

CNTPCT_EL0:

- Is always accessible from EL3, Non-secure EL2 and Secure EL1 states.
- Is accessible from Non-secure EL1 only when CNTHCTL_EL2.EL1PCTEN is set to 1. When the value of CNTHCTL_EL2.EL1PCTEN is 0, any attempt to access CNTPCT_EL0 from Non-secure EL1 is trapped to EL2.
- Is accessible from Secure EL0 state when the value of CNTKCTL_EL1.EL0PCTEN is 1. When the value of CNTKCTL_EL1.EL0PCTEN is 0, any attempt to access CNTPCT_EL0 generates an UNDEFINED exception.
- Is accessible from Non-secure EL0 when the value of CNTHCTL_EL2.EL1PCTEN is 1 and the value of CNTKCTL_EL1.EL0PCTEN is 1. Otherwise:
  - When the value of CNTKCTL_EL1.EL0PCTEN is 0, any attempt to access CNTPCT_EL0 from Non-secure EL0 generates an UNDEFINED exception.
  - When the value of CNTKCTL_EL1.EL0PCTEN is 1 and the value of CNTHCTL_EL2.EL1PCTEN is 0, any attempt to access CNTPCT_EL0 from Non-secure EL0 is trapped to EL2.

Reads of CNTPCT_EL0 can occur speculatively and out of order relative to other instructions executed on the same PE.

For example, if a read from memory is used to obtain a signal from another agent that indicates that CNTPCT_EL0 must be read, an ISB must be used to ensure that the read of CNTPCT_EL0 occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                ; polling for some communication to indicate a requirement to read the timer
    LDR R1, [R2]
    CMP R1, #1
    BNE loop
    ISB             ; without this, the CNTPCT could be read before the memory location in [R2]
                    ; has had the value 1 written to it
    MRS R1, CNTPCT
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D6.1.3 The virtual counter

An implementation of the Generic Timer always includes a virtual counter, that indicates virtual time.

The virtual counter contains the value of the physical counter minus a 64-bit virtual offset. When executing at Non-secure EL1 or EL0, the virtual offset value relates to the current virtual machine.

The CNTVOFF_EL2 register contains the virtual offset. CNTVOFF_EL2 is only accessible from EL2 and EL3.

For more information see *Status of the CNTVOFF register*.

The CNTVCT_EL0 register holds the current virtual counter value.

#### Accessing the virtual counter

Software with sufficient privilege can read CNTVCT_EL0 using a 64-bit System register read.

CNTVCT_EL0 is always accessible from Secure EL3, from Secure EL1 when EL3 is using AArch64, and from Non-secure EL1 and EL2.

In addition, when CNTKCTL_EL1.EL0VCTEN is set to 1, CNTVCT_EL0 is accessible from EL0.

When CNTKCTL_EL1.EL0VCTEN is set to 0, any attempt to access CNTVCT_EL0 from EL0 is UNDEFINED.

Reads of CNTVCT_EL0 can occur speculatively and out of order relative to other instructions executed on the same PE.

For example, if a read from memory is used to obtain a signal from another agent that indicates that CNTVCT_EL0 must be read, an ISB must be used to ensure that the read of CNTVCT_EL0 occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                ; polling for some communication to indicate a requirement to read the timer
    LDR R1, [R2]
    CMP R1, #1
    BNE loop
    ISB             ; without this, the CNTVCT could be read before the memory location in [R2]
                    ; has had the value 1 written to it
    MRS R1, CNTVCT
```

### D6.1.4 Status of the CNTVOFF register

All implementations of the Generic Timer include the virtual counter. Therefore, conceptually, all implementations include the CNTVOFF_EL2 register that defines the *virtual offset* between the physical count and the virtual count. CNTVOFF_EL2 is only accessible at EL2 or above. If EL2 is not implemented, the virtual counter uses a fixed virtual offset of zero.

### D6.1.5 Event streams

An implementation that includes the Generic Timer can use the system counter to generate one or more *event streams*, to generate periodic wake-up events as part of the mechanism described in *Wait for Event mechanism and Send event* on page D1-1611.

**Note**

An event stream might be used:

- To impose a time-out on a Wait For Event polling loop.
- To safeguard against any programming error that means an expected event is not generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An event stream is configured by:

- Selecting which bit, from the bottom 16 bits of a counter, triggers the event. This determines the frequency of the events in the stream.
- Selecting whether the event is generated on each 0 to 1 transition, or each 1 to 0 transition, of the selected counter bit.

The CNTKCTL_EL1.{EVNTEN, EVNTDIR, EVNTI} fields define an event stream that is generated from the virtual counter.

In all implementations the CNTHCTL_EL2.{EVNTEN, EVNTDIR, EVNTI} fields define an event stream that is generated from the physical counter.

The operation of an event stream is as follows:

- The pseudocode variables PreviousCNTVCT and PreviousCNTPCT are initialized as:

```
// Variables used for generation of the timer event stream.
bits(64) PreviousCNTVCT = bits(64) UNKNOWN;
bits(64) PreviousCNTPCT = bits(64) UNKNOWN;
```

- The pseudocode functions TestEventCNTV() and TestEventCNTP() are called on each cycle of the PE clock.
- The TestEventCNTx() pseudocode template defines the functions TestEventCNTV() and TestEventCNTP():

```
// TestEventCNTx()
// ===============

// Template for the TestEventCNTV() and TestEventCNTP() functions
// Describes operation when all Exception Levels are using AArch64:
//   CNTxCT_EL0         is  CNTVCT_EL0          or  CNTPCT_EL0          64-bit count value
//   CNTx_CTL_EL0       is  CNTV_CTL_EL0        or  CNTP_CTL_EL0        Control register
//   PreviousCNTxCT_EL0 is  PreviousCNTVCT_EL0  or  PreviousCNTPCT_EL0

TestEventCNTx()
    if CNTx_CTL_EL0.EVNTEN == '1' then
        n = UInt(CNTx_CTL_EL0.EVNTI);
        SampleBit   = CNTxCT_EL0<n>;
        PreviousBit = PreviousCNTxCT_EL0<n>;

        if CNTx_CTL_EL0.EVNTDIR == '0' then
            if PreviousBit == '0' && SampleBit == '1' then EventRegisterSet();
        else
            if PreviousBit == '1' && SampleBit == '0' then EventRegisterSet();

    PreviousCNTxCT_EL0 = CNTxCT_EL0;

    return;
```

### D6.1.6 Timers

In an implementation of the Generic Timer that includes EL3, if EL3 can use AArch64, the following timers are implemented:

- An EL1 physical timer, that:
  - In Secure state, can be accessed from EL1.
  - In Non-secure state, can be accessed from EL1 unless those accesses are trapped to EL2.

  When this timer can be accessed from EL1, an EL1 control determines whether it can be accessed from EL0.
- A Non-secure EL2 physical timer.
- A Secure EL3 physical timer. An EL3 control determines whether this register is accessible from Secure EL1.
- A virtual timer.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The output of each implemented timer:

- Provides an output signal to the system.
- If the PE interfaces to a *Generic Interrupt Controller* (GIC), signals a *Private Peripheral Interrupt* (PPI) to that GIC. In a multiprocessor implementation, each PE must use the same interrupt number for each timer.

Each timer is implemented as three registers:

- A 64-bit CompareValue register, that provides a 64-bit unsigned upcounter.
- A 32-bit TimerValue register, that provides a 32-bit signed downcounter.
- A 32-bit Control register.

**Table D6-1 Timer registers summary for the Generic Timer**

| Timer register | EL1 physical timer | EL2 physical timer | EL3 physical timer | Virtual timer |
|---|---|---|---|---|
| CompareValue register | CNTP_CVAL_EL0 | CNTHP_CVAL_EL2 | CNTPS_CVAL_EL1 | CNTV_CVAL_EL0 |
| TimerValue register | CNTP_TVAL_EL0 | CNTHP_TVAL_EL2 | CNTPS_TVAL_EL1 | CNTV_TVAL_EL0 |
| Control register | CNTP_CTL_EL0 | CNTHP_CTL_EL2 | CNTPS_CTL_EL1 | CNTV_CTL_EL0 |

The following sections describe:

- *Accessing the timer registers*
- *Operation of the CompareValue views of the timers* on page D6-1904
- *Operation of the TimerValue views of the timers* on page D6-1904.

## Accessing the timer registers

For each timer, all timer registers have the same access permissions, as follows:

**EL1 physical timer** Accessible from EL1, except that Non-secure software executing at EL2 controls access from Non-secure EL1.

When access from EL1 is permitted, CNTKCTL_EL1.EL0PTEN determines whether the registers are accessible from EL0. If an access is not permitted because CNTKCTL_EL1.EL0PTEN is set to 0, an attempted access from EL0 is UNDEFINED.

In all implementations:

- The registers are accessible from EL3.
- In Non-secure state, the registers are accessible from EL2.
- CNTHCTL_EL2.EL1PCEN determines whether the Non-secure registers are accessible from Non-secure EL1. If this bit is set to 1, to enable access from Non-secure EL1, CNTKCTL_EL1.EL0PTEN determines whether the registers are accessible from Non-secure EL0.

  If an access is not permitted because CNTHCTL_EL2.EL1PCEN is set to 0, an attempted access from a Non-secure EL1 or EL0 is trapped to EL2. However, if CNTKCTL_EL1.EL0PTEN is set to 0, this control takes priority, and an attempted access from EL0 is UNDEFINED, and generates an exception that is taken to EL1.

**EL2 physical timer** Accessible from EL2, and from EL3 when SCR_EL3.NS is set to 1.

**EL3 physical timer** Accessible from EL3.

SCR_EL3.ST determines whether the timer is also accessible from Secure EL1. The effect of the ST bit is:

`0` The EL3 physical timer is only accessible from EL3. Attempts to access this timer from Secure EL1 generate an exception that is taken to EL3.

`1` The EL3 physical timer is accessible from EL3 and from Secure EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The EL3 physical timer registers are always UNDEFINED in Non-secure EL1, and any attempt to access them from Non-secure EL1 generates an exception that is taken to EL1.

**Virtual timer** Accessible from Secure and Non-secure EL1, from EL2, and from EL3.

CNTKCTL_EL1.EL0VTEN determines whether the registers are accessible from EL0. If an access is not permitted because CNTKCTL_EL1.EL0VTEN is set to 0, an attempted access from an EL0 is UNDEFINED.

### Operation of the CompareValue views of the timers

The CompareValue view of a timer operates as a 64-bit upcounter. The timer triggers when the appropriate counter reaches the value programmed into a CompareValue register. When the timer triggers, it generates an interrupt if the interrupt is enabled in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTPS_CTL_EL1, or CNTV_CTL_EL0. For CNTP_CTL_EL0, the asserted interrupt is the same as the interrupt asserted by the Non-secure instance of the AArch32 register CNTP_CTL.

The operation of this view of a timer is:

```
EventTriggered = (((Counter[63:0] - Offset[63:0])[63:0] - CompareValue[63:0]) >= 0)
```

Where:

`EventTriggered` Is TRUE if the event for this timer must be triggered, and FALSE otherwise.

`Counter` The physical counter value, that can be read from the CNTPCT_EL0 register.

**Note**

The virtual counter value, that can be read from the CNTVCT_EL0 register, is the value:

```
(Counter - Offset)
```

`Offset` For a physical timer it is zero, and for the virtual timer it is the virtual offset, held in the CNTVOFF_EL2 register.

`CompareValue` The value of the appropriate CompareValue register, CNTP_CVAL_EL0, CNTHP_CVAL_EL2, CNTPS_CVAL_EL1, or CNTV_CVAL_EL0.

In this view of a timer, `Counter`, `Offset`, and `CompareValue` are all 64-bit unsigned values.

**Note**

This means that a timer with a `CompareValue` of, or close to, `0xFFFF_FFFF_FFFF_FFFF` might never trigger. However, there is no practical requirement to use values close to the counter wrap value.

### Operation of the TimerValue views of the timers

The TimerValue view of a timer operates as a signed 32-bit downcounter. A TimerValue register is programmed with a count value. This value decrements on each increment of the appropriate counter, and the timer triggers when the value reaches zero. When the timer triggers, it generates an interrupt if the interrupt is enabled in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTPS_CTL_EL1, or CNTV_CTL_EL0.

This view of a timer depends on the following behavior of accesses to TimerValue registers:

**Reads** `TimerValue = (CompareValue - (Counter - Offset))[31:0]`

**Writes** `CompareValue = ((Counter - Offset)[63:0] + SignExtend(TimerValue))[63:0]`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Where the arguments have the definitions used in *Operation of the CompareValue views of the timers* on page D6-1904, and in addition:

`TimerValue` The value of a TimerValue register, CNTP_TVAL_EL0, CNTHP_TVAL_EL2, CNTPS_TVAL_EL1, or CNTV_TVAL_EL0.

In this view of a timer, all values are signed, in standard two's complement form.

After an event has triggered, a read of a TimerValue register indicates the time since the event triggered.

**Note**

- *Operation of the CompareValue views of the timers* on page D6-1904 gives a strict definition of EventTriggered. However, provided that the TimerValue is not expected to wrap as a 32-bit signed value when decremented from `0x80000000`, the TimerValue view can be used as giving an effect equivalent to:

  ```
  EventTriggered = (TimerValue ≤ 0)
  ```

- Programming TimerValue to a negative number with magnitude greater than (Counter−Offset) can lead to an arithmetic overflow that causes the CompareValue to be an extremely large positive value. This potentially delays the resultant interrupt for an extremely long period of time.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter D7
# AArch64 System Register Descriptions

This chapter defines the AArch64 System registers. It contains the following sections:


- *About the AArch64 System registers* on page D7-1908.
- *General system control registers* on page D7-1913.
- *Debug registers* on page D7-2165.
- *Performance Monitors registers* on page D7-2236.
- *Generic Timer registers* on page D7-2276.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D7.1 About the AArch64 System registers

The following sections describe common features of the AArch64 registers:

- *Fixed values in the System register descriptions*.
- *General behavior of accesses to the System registers*.

### D7.1.1 Fixed values in the System register descriptions

See *Fixed values in AArch64 instruction and System register descriptions* on page C2-135. This section defines how the glossary terms RAZ, RES0, RAO, and RES1 can be represented in the System register descriptions.

#### System register width

The register descriptions in this chapter describe each register as either a *32-bit register* or a *64-bit register*. For 32-bit registers, the upper bits, bit[63:32], are RES0.

### D7.1.2 General behavior of accesses to the System registers

This section gives general information about the behavior of accesses to the System registers.

#### Synchronization requirements for System registers

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that any data dependencies between the instructions are respected.

**Note**

In particular, System registers that hold self-incrementing counts such as the performance counters or the Generic Timer counter or timers, can be read early. For example, where a memory access is used to communicate a read of such counters, an `ISB` must be inserted between the read of the memory location and the read of the Generic Timer counter, where it is necessary that the Generic Timer counter returns a count value after the memory communication.

Direct writes using the instructions in Table C5-6 on page C5-279 require synchronization before software can rely on the effects of changes to the System registers to affect instructions appearing in program order after the direct write to the System register. Direct writes to these registers are not allowed to affect any instructions appearing in program order before the direct write. The only exceptions are:

- All direct writes to the same register, that use the same encoding for that register, are guaranteed to occur in program order relative to each other
- All direct writes to a register occur in program order with respect to all direct reads to the same register using the same encoding.
- Any System register access that an ARM *Architecture Specification* or equivalent specification defines as not requiring synchronization.

Explicit synchronization occurs as a result of a *Context synchronization operation*, which is one of the following events:

- Execution of an `ISB` instruction.
- Exception entry.
- Exception return.
- Execution of a `DCPS` instruction in Debug state.
- Execution of a `DRPS` instruction in Debug state.
- Exit from Debug state.

**Note**

The `ISB` and exception entry events are applicable both in Debug state and in Non-debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Conceptually, explicit synchronization occurs as the first step of each of these events, so that if the event uses state that has previously been changed but was not synchronized by the time of the event, the event is guaranteed to use the state as if it had been synchronized.

**Note**

This explicit synchronization applies as the first step of the execution of the events, and does not apply to any effect of System registers that apply to the fetch and decode of the instructions that cause these events, such as breakpoints or changes to the translation table.

In addition, any system instructions that cause a write to a System register must be synchronized before the result is guaranteed to be visible to subsequent direct reads of that System register.

Direct reads to any one of the following registers, using the same encoding, occur in program order relative to each other:

- ISR_EL1.
- The Generic Timer registers, that is, CNTPCT_EL0 and CNTVCT_EL0, and the Counter registers CNTP_TVAL_EL0, CNTV_TVAL_EL0, CNTHP_TVAL_EL2, and CNTPS_TVAL_EL1.
- DBGCLAIMCLR_EL1.
- The PMU Counters, that is, PMCCNTR_EL0, PMEVCNTR<n>_EL0, PMXEVCNTR_EL0, PMOVSCLR_EL0, and PMOVSSET_EL0.
- The Debug Communications Channel registers, that is, DBGDTRTX_EL0, DBGDTRRX_EL0, DBGDTR_EL0, and MDCCSR_EL0.

All other direct reads of System registers can occur in any order if synchronization has not been performed.

Table D7-1 describes the synchronization requirements between two successive read or write accesses to the same register, where the ordering of the read or write accesses is:

1. Program order, in the event that both the reads or writes are caused by an instruction executed on this PE, other than one caused by a memory access by this PE.
2. The order of arrival of asynchronous reads and writes at the PE relative to the execution of instructions that cause reads or writes.
3. The order of arrival of asynchronous reads and writes at the PE relative to each other.

**Table D7-1 Synchronization requirements**

| First read-write | Second read-write | Synchronization requirement |
|---|---|---|
| Direct read | Direct read | None |
| | Direct write | None |
| | Indirect read | None |
| | Indirect write | None, see *Notes* on page D7-1910 |
| Direct write | Direct read | None |
| | Direct write | None |
| | Indirect read | Required |
| | Indirect write | None, see *Notes* on page D7-1910 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table D7-1 Synchronization requirements (continued)**

| First read-write | Second read-write | Synchronization requirement |
|---|---|---|
| Indirect read | Direct read | None |
| | Direct write | None |
| | Indirect read | None |
| | Indirect write | None |
| Indirect write | Direct read | Required, see *Notes* |
| | Direct write | None, see *Notes* |
| | Indirect read | Required, see *Notes* |
| | Indirect write | None, see *Notes* |

### *Notes*

The terms Direct read, Direct write, Indirect read, and Indirect write, as used in Table D7-1 on page D7-1909, are defined as follows:

**Direct read** Where software uses a System register access instruction to read the register, see both:

- *Instructions for accessing non-debug System registers* on page C5-278.
- *Instructions for accessing debug System registers* on page C5-276.

Where a direct read of a register has a side-effect that changes the contents of a register, the effect of a direct read on that register is defined to be an indirect write. In this case, the indirect write is only guaranteed to have occurred, and be visible to subsequent direct or indirect reads or writes, if synchronization is performed after the direct read.

**Direct write** Where software uses a System register access instruction to write to the register, see both:

- *Instructions for accessing non-debug System registers* on page C5-278.
- *Instructions for accessing debug System registers* on page C5-276.

Where a direct write to a register has an effect on the register that means that the value in the register is not always the last value that is written (as is the case with set and clear registers), the effect of a direct write on that register is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct write and before the subsequent direct or indirect reads or writes.

**Indirect read** Where an instruction uses a System register to establish operating conditions, for example, TTBR addresses or whether the cache is enabled, for the instruction. This includes situations where the contents of one System register selects what value is read using a different register. Indirect reads also include reads of the System register by external agents such as debuggers. Where an indirect read of a register has a side-effect that changes the contents of that register, that is defined to be an indirect write.

**Indirect write** Where a System register is written as the consequence of some other instruction, exception, operation, or by the asynchronous operation of an external agent, including the passage of time as seen in counters, timers, or performance counters, the assertion of interrupts, or writes from an external debugger.

**Note**

Since an exception is context synchronizing, registers such as the Exception Syndrome registers that are indirectly written as part of exception entry do not require additional synchronization.

Where a direct read or write to a register is followed by an indirect write caused by an external agent, autonomous asynchronous event, or as a result of memory mapped write, synchronization is required to guarantee the order of those two accesses.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Where an indirect write caused by a direct write is followed by an indirect write caused by an external agent, autonomous asynchronous event, or as a result of memory mapped write, synchronization is required to guarantee the order of those two indirect accesses.

Where a direct read to one register causes a bit or field in a different register (or the same register using a different encoding) to be updated, the change to the different register (or same register using a different encoding) is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct read and before the subsequent direct or indirect reads or writes.

Where a direct write to one register causes a bit or field in a different register (or the same register using a different encoding) to be updated as a side-effect of that direct write (as opposed to simply being a direct write to the different encoding), the change to the different register (or same register using a different encoding) is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct write and before the subsequent direct or indirect reads or writes.

Where indirect writes are caused by the actions of external agents such as debuggers, or by memory-mapped reads or writes by the PE, then an indirect write by that agent and mechanism to a register, followed by an indirect read by that agent and mechanism to the same register using the same address, does not require synchronization.

Indirect writes caused by external agents, autonomous asynchronous events, or as a result of memory-mapped writes, to the registers shown in Table D7-2, are required to be observable to:

- Direct reads in finite time without explicit synchronization.
- Subsequent indirect reads without explicit synchronization.

**Table D7-2 Registers with a guarantee of observability, VMSAv8-64**

| Registers | Notes |
|---|---|
| ISR_EL1 | Interrupt Status Register |
| DBGCLAIMCLR_EL1, DBGCLAIMSET_EL1 | Debug CLAIM registers |
| CNTPCT_EL0, CNTVCT_EL0, CNTP_TVAL_EL0, CNTV_TVAL_EL0, CNTHP_TVAL_EL2, CNTPS_TVAL_EL1 | Generic Timer registers |
| PMCCNTR_EL0, PMEVCNTR<n>_EL0, PMXEVCNTR_EL0, PMOVSCLR_EL0, PMOVSSET_EL0 | PMU Counters |
| DBGDTRTX_EL0, DBGDTRRX_EL0, DBGDTR_EL0, and the DCC flags in MDCCSR_EL0 and EDSCR | Debug Communication Channel registers |
| EDSCR.PipeAdv | External Debug Status and Control Register PipeAdv field |

In addition to the requirements shown in Table D7-2, when the PE is in Debug state, there are synchronization requirements for:

- The EDITR and ITR flags in EDSCR.
- DBGDTRTX_EL0, DBGDTRRX_EL0, and the DCC flags in EDSCR.

See *DCC and ITR access in Debug state* on page H4-4901.

**Note**

- The provision of explicit synchronization requirements to System registers is provided to allow the direct access to these registers to be implemented in a small number of cycles, and that updates to multiple registers can be performed quickly with the synchronization penalty being paid only when the updates have occurred.
- Since toolkits might use registers such as the thread-local storage registers within compiled code, it is recommended that access to these registers is implemented to take a small number of cycles.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

- While no synchronization is required between a direct write and a direct read, or between a direct read and an indirect write, this does not imply that a direct read causes synchronization of a previous direct write. That is, the sequence direct write → direct read → indirect read, with no intervening context synchronization, does not guarantee that the indirect read observes the result of the direct write.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D7.2 General system control registers

This section lists the System registers in AArch64 that are not part of one of the other listed groups.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.1 ACTLR_EL1, Auxiliary Control Register (EL1)

The ACTLR_EL1 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TACR==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ACTLR_EL1[31:0] is architecturally mapped to AArch32 System register ACTLR.

AArch64 System register ACTLR_EL1[63:32] is architecturally mapped to AArch32 System register ACTLR2.

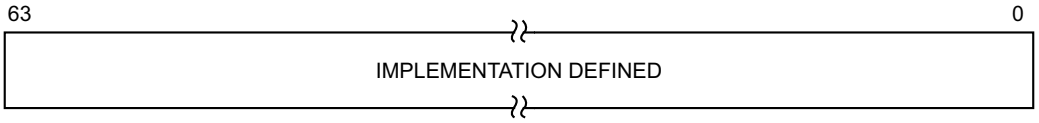
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

ACTLR_EL1 is a 64-bit register.

#### Field descriptions

The ACTLR_EL1 bit assignments are:




**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

#### Accessing the ACTLR_EL1:

To access the ACTLR_EL1:

```
MRS <Xt>, ACTLR_EL1 ; Read ACTLR_EL1 into Xt
MSR ACTLR_EL1, <Xt> ; Write Xt to ACTLR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0001 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.2 ACTLR_EL2, Auxiliary Control Register (EL2)

The ACTLR_EL2 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED configuration and control options for EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register ACTLR_EL2[31:0] is architecturally mapped to AArch32 System register HACTLR.

AArch64 System register ACTLR_EL2[63:32] is architecturally mapped to AArch32 System register HACTLR2.

If EL2 is not implemented, this register is RES0 from EL3.

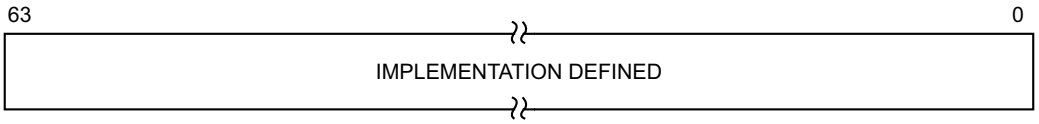
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

ACTLR_EL2 is a 64-bit register.

#### Field descriptions

The ACTLR_EL2 bit assignments are:




**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

#### Accessing the ACTLR_EL2:

To access the ACTLR_EL2:

```
MRS <Xt>, ACTLR_EL2 ; Read ACTLR_EL2 into Xt
MSR ACTLR_EL2, <Xt> ; Write Xt to ACTLR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0001 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.3 ACTLR_EL3, Auxiliary Control Register (EL3)

The ACTLR_EL3 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED configuration and control options for EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

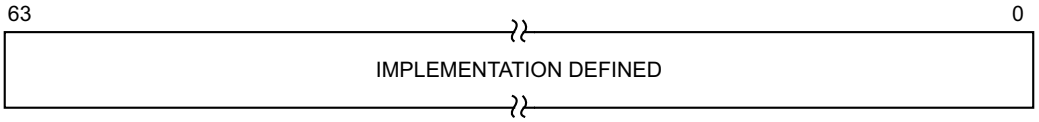
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

ACTLR_EL3 is a 64-bit register.

#### Field descriptions

The ACTLR_EL3 bit assignments are:




**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

#### Accessing the ACTLR_EL3:

To access the ACTLR_EL3:

```
MRS <Xt>, ACTLR_EL3 ; Read ACTLR_EL3 into Xt
MSR ACTLR_EL3, <Xt> ; Write Xt to ACTLR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0001 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.4 AFSR0_EL1, Auxiliary Fault Status Register 0 (EL1)

The AFSR0_EL1 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register AFSR0_EL1 is architecturally mapped to AArch32 System register ADFSR.

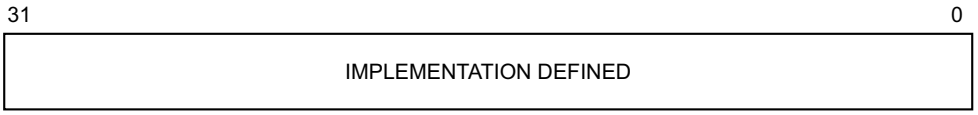
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AFSR0_EL1 is a 32-bit register.

#### Field descriptions

The AFSR0_EL1 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the AFSR0_EL1:

To access the AFSR0_EL1:

```
MRS <Xt>, AFSR0_EL1 ; Read AFSR0_EL1 into Xt
MSR AFSR0_EL1, <Xt> ; Write Xt to AFSR0_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0101 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.5 AFSR0_EL2, Auxiliary Fault Status Register 0 (EL2)

The AFSR0_EL2 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register AFSR0_EL2 is architecturally mapped to AArch32 System register HADFSR.

If EL2 is not implemented, this register is RES0 from EL3.

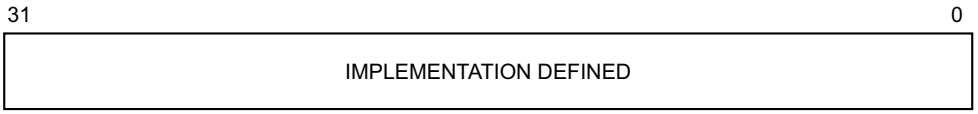
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AFSR0_EL2 is a 32-bit register.

#### Field descriptions

The AFSR0_EL2 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the AFSR0_EL2:

To access the AFSR0_EL2:

```
MRS <Xt>, AFSR0_EL2 ; Read AFSR0_EL2 into Xt
MSR AFSR0_EL2, <Xt> ; Write Xt to AFSR0_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0101 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.6 AFSR0_EL3, Auxiliary Fault Status Register 0 (EL3)

The AFSR0_EL3 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AFSR0_EL3 is a 32-bit register.

#### Field descriptions

The AFSR0_EL3 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the AFSR0_EL3:

To access the AFSR0_EL3:

```
MRS <Xt>, AFSR0_EL3 ; Read AFSR0_EL3 into Xt
MSR AFSR0_EL3, <Xt> ; Write Xt to AFSR0_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0101 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.7 AFSR1_EL1, Auxiliary Fault Status Register 1 (EL1)

The AFSR1_EL1 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register AFSR1_EL1 is architecturally mapped to AArch32 System register AIFSR.

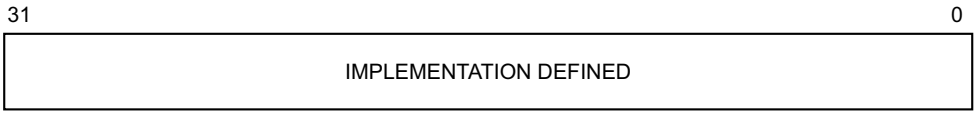
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AFSR1_EL1 is a 32-bit register.

#### Field descriptions

The AFSR1_EL1 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the AFSR1_EL1:

To access the AFSR1_EL1:

```
MRS <Xt>, AFSR1_EL1 ; Read AFSR1_EL1 into Xt
MSR AFSR1_EL1, <Xt> ; Write Xt to AFSR1_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0101 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.8 AFSR1_EL2, Auxiliary Fault Status Register 1 (EL2)

The AFSR1_EL2 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register AFSR1_EL2 is architecturally mapped to AArch32 System register HAIFSR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AFSR1_EL2 is a 32-bit register.

#### Field descriptions

The AFSR1_EL2 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the AFSR1_EL2:

To access the AFSR1_EL2:

```
MRS <Xt>, AFSR1_EL2 ; Read AFSR1_EL2 into Xt
MSR AFSR1_EL2, <Xt> ; Write Xt to AFSR1_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0101 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.9 AFSR1_EL3, Auxiliary Fault Status Register 1 (EL3)

The AFSR1_EL3 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AFSR1_EL3 is a 32-bit register.

#### Field descriptions

The AFSR1_EL3 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the AFSR1_EL3:

To access the AFSR1_EL3:

```
MRS <Xt>, AFSR1_EL3 ; Read AFSR1_EL3 into Xt
MSR AFSR1_EL3, <Xt> ; Write Xt to AFSR1_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0101 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.10 AIDR_EL1, Auxiliary ID Register

The AIDR_EL1 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED identification information.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

The value of this register must be interpreted in conjunction with the value of MIDR_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID1==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register AIDR_EL1 is architecturally mapped to AArch32 System register AIDR.

**Attributes**

AIDR_EL1 is a 32-bit register.

#### Field descriptions

The AIDR_EL1 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the AIDR_EL1:

To access the AIDR_EL1:

```
MRS <Xt>, AIDR_EL1 ; Read AIDR_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 001 | 0000 | 0000 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.11 AMAIR_EL1, Auxiliary Memory Attribute Indirection Register (EL1)

The AMAIR_EL1 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

AMAIR_EL1 is permitted to be cached in a TLB.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register AMAIR_EL1[31:0] is architecturally mapped to AArch32 System register AMAIR0.

AArch64 System register AMAIR_EL1[63:32] is architecturally mapped to AArch32 System register AMAIR1.

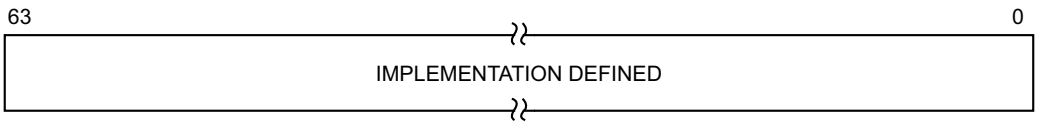
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AMAIR_EL1 is a 64-bit register.

#### Field descriptions

The AMAIR_EL1 bit assignments are:




**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

#### Accessing the AMAIR_EL1:

To access the AMAIR_EL1:

```
MRS <Xt>, AMAIR_EL1 ; Read AMAIR_EL1 into Xt
MSR AMAIR_EL1, <Xt> ; Write Xt to AMAIR_EL1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1010 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.12 AMAIR_EL2, Auxiliary Memory Attribute Indirection Register (EL2)

The AMAIR_EL2 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

AMAIR_EL2 is permitted to be cached in a TLB.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register AMAIR_EL2[31:0] is architecturally mapped to AArch32 System register HAMAIR0.

AArch64 System register AMAIR_EL2[63:32] is architecturally mapped to AArch32 System register HAMAIR1.

If EL2 is not implemented, this register is RES0 from EL3.

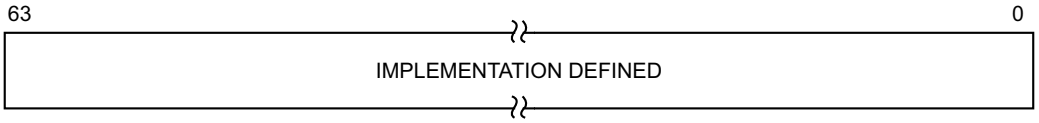
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AMAIR_EL2 is a 64-bit register.

#### Field descriptions

The AMAIR_EL2 bit assignments are:




**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

#### Accessing the AMAIR_EL2:

To access the AMAIR_EL2:

```
MRS <Xt>, AMAIR_EL2 ; Read AMAIR_EL2 into Xt
MSR AMAIR_EL2, <Xt> ; Write Xt to AMAIR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1010 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.13 AMAIR_EL3, Auxiliary Memory Attribute Indirection Register (EL3)

The AMAIR_EL3 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

AMAIR_EL3 is permitted to be cached in a TLB.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

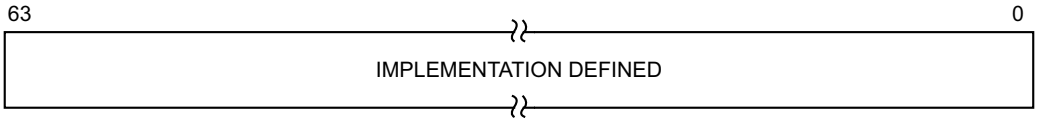
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AMAIR_EL3 is a 64-bit register.

#### Field descriptions

The AMAIR_EL3 bit assignments are:




**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

#### Accessing the AMAIR_EL3:

To access the AMAIR_EL3:

```
MRS <Xt>, AMAIR_EL3 ; Read AMAIR_EL3 into Xt
MSR AMAIR_EL3, <Xt> ; Write Xt to AMAIR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 1010 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.14 CCSIDR_EL1, Current Cache Size ID Register

The CCSIDR_EL1 characteristics are:

**Purpose**

Provides information about the architecture of the currently selected cache.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

If CSSELR_EL1.Level indicates a cache that is not implemented, then on a read of the CCSIDR_EL1 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR_EL1 read is treated as NOP.
- The CCSIDR_EL1 read is UNDEFINED.
- The CCSIDR_EL1 read returns an UNKNOWN value.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID2==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register CCSIDR_EL1 is architecturally mapped to AArch32 System register CCSIDR.

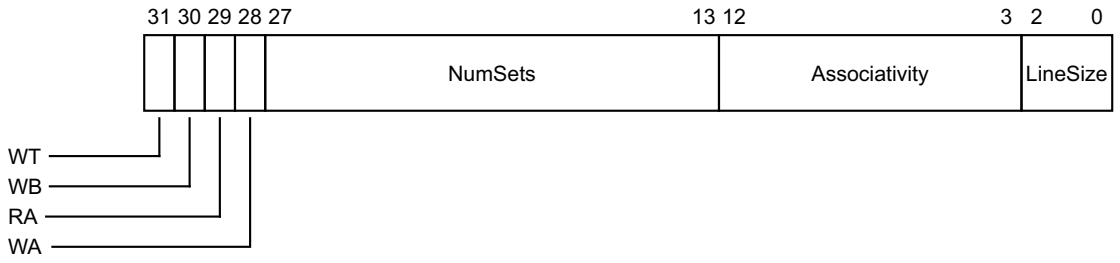
The implementation includes one CCSIDR_EL1 for each cache that it can access. CSSELR_EL1 selects which Cache Size ID Register is accessible.

**Attributes**

CCSIDR_EL1 is a 32-bit register.

## Field descriptions

The CCSIDR_EL1 bit assignments are:




**WT, bit [31]**

Indicates whether the selected cache level supports write-through. Permitted values are:

`0` Write-through not supported.

`1` Write-through supported.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**WB, bit [30]**

Indicates whether the selected cache level supports write-back. Permitted values are:

0 Write-back not supported.

1 Write-back supported.

**RA, bit [29]**

Indicates whether the selected cache level supports read-allocation. Permitted values are:

0 Read-allocation not supported.

1 Read-allocation supported.

**WA, bit [28]**

Indicates whether the selected cache level supports write-allocation. Permitted values are:

0 Write-allocation not supported.

1 Write-allocation supported.

**NumSets, bits [27:13]**

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

**Associativity, bits [12:3]**

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

**LineSize, bits [2:0]**

($Log_2$(Number of bytes in cache line)) - 4. For example:

For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.

For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

**Note**

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

## Accessing the CCSIDR_EL1:

To access the CCSIDR_EL1:

```
MRS <Xt>, CCSIDR_EL1 ; Read CCSIDR_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 001 | 0000 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.15 CLIDR_EL1, Cache Level ID Register

The CLIDR_EL1 characteristics are:

**Purpose**

Identifies the type of cache, or caches, implemented at each level, up to a maximum of seven levels. Also identifies the Level of Coherence (LoC) and Level of Unification (LoU) for the cache hierarchy.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID2==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

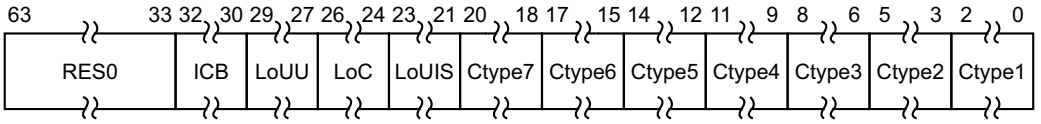
AArch64 System register CLIDR_EL1 is architecturally mapped to AArch32 System register CLIDR.

**Attributes**

CLIDR_EL1 is a 64-bit register.

#### Field descriptions

The CLIDR_EL1 bit assignments are:




**Bits [63:33]**

Reserved, RES0.

**ICB, bits [32:30]**

Inner cache boundary. This field indicates the boundary between the inner and the outer domain.

The possible values are:

000 Not disclosed in this mechanism.

001 L1 cache is the highest inner level.

010 L2 cache is the highest inner level.

011 L3 cache is the highest inner level.

100 L4 cache is the highest inner level.

101 L5 cache is the highest inner level.

110 L6 cache is the highest inner level.

111 L7 cache is the highest inner level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**LoUU, bits [29:27]**

Level of Unification Uniprocessor for the cache hierarchy.

**LoC, bits [26:24]**

Level of Coherence for the cache hierarchy.

**LoUIS, bits [23:21]**

Level of Unification Inner Shareable for the cache hierarchy.

**Ctype<n>, bits [3(n-1)+2:3(n-1)], for n = 1 to 7**

Cache Type fields. Indicate the type of cache implemented at each level, from Level 1 up to a maximum of seven levels of cache hierarchy. Possible values of each field are:

`000` No cache.

`001` Instruction cache only.

`010` Data cache only.

`011` Separate instruction and data caches.

`100` Unified cache.

All other values are reserved.

If software reads the Cache Type fields from Ctype1 upwards, once it has seen a value of `000`, no caches exist at further-out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of `000`, the values of Ctype4 to Ctype7 must be ignored.

### Accessing the CLIDR_EL1:

To access the CLIDR_EL1:

```
MRS <Xt>, CLIDR_EL1 ; Read CLIDR_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 001 | 0000 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.16 CONTEXTIDR_EL1, Context ID Register

The CONTEXTIDR_EL1 characteristics are:

**Purpose**

Identifies the current Process Identifier.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register CONTEXTIDR_EL1 is architecturally mapped to AArch32 System register CONTEXTIDR.

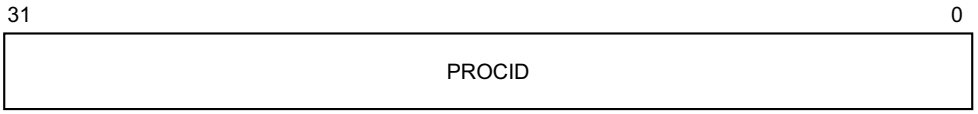
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CONTEXTIDR_EL1 is a 32-bit register.

#### Field descriptions

The CONTEXTIDR_EL1 bit assignments are:




**PROCID, bits [31:0]**

Process Identifier. This field must be programmed with a unique value that identifies the current process.

**Note**

In AArch32 state, CONTEXTIDR[7:0] holds the ASID. In AArch64 state, CONTEXTIDR_EL1[31:0] is independent of the ASID, and for the EL1&0 translation regime either TTBR0_EL1 or TTBR1_EL1 holds the ASID.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CONTEXTIDR_EL1:

To access the CONTEXTIDR_EL1:

```
MRS <Xt>, CONTEXTIDR_EL1 ; Read CONTEXTIDR_EL1 into Xt
MSR CONTEXTIDR_EL1, <Xt> ; Write Xt to CONTEXTIDR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1101 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.17 CPACR_EL1, Architectural Feature Access Control Register

The CPACR_EL1 characteristics are:

**Purpose**

Controls access to Trace, floating-point, and Advanced SIMD functionality.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CPTR_EL2.TCPAC==1, Non-secure EL1 accesses to this register are trapped to EL2.

If CPTR_EL3.TCPAC==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register CPACR_EL1 is architecturally mapped to AArch32 System register CPACR.

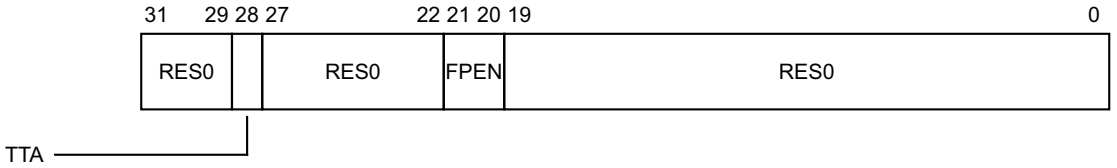
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CPACR_EL1 is a 32-bit register.

#### Field descriptions

The CPACR_EL1 bit assignments are:




**Bits [31:29]**

Reserved, RES0.

**TTA, bit [28]**

Traps EL0 and EL1 System register accesses to all implemented trace registers to EL1, from both Execution states.

`0` EL0 and EL1 System register accesses to all implemented trace registers are not trapped to EL1.

`1` EL0 and EL1 System register accesses to all implemented trace registers are trapped to EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED. A resulting Undefined Instruction exception is higher priority than a CPACR_EL1.TTA exception.
- The ARMv8-A architecture does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not implemented, this bit is RES0.

**Bits [27:22]**

Reserved, RES0.

**FPEN, bits [21:20]**

Traps EL0 and EL1 accesses to the SIMD and floating-point registers to EL1, from both Execution states.

`00` Causes any instructions in EL0 or EL1 that use the registers associated with floating-point and Advanced SIMD execution to be trapped.

`01` Causes any instructions in EL0 that use the registers associated with floating-point and Advanced SIMD execution to be trapped, but does not cause any instruction in EL1 to be trapped.

`10` Causes any instructions in EL0 or EL1 that use the registers associated with floating-point and Advanced SIMD execution to be trapped.

`11` Does not cause any instruction to be trapped.

Writes to MVFR0, MVFR1 and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

**Note**

- Attempts to write to the FPSID do count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2 and FPEXC are UNDEFINED, and an Undefined instruction exception takes precedence over an exception resulting from an access to an Advanced SIMD or Floating-point register.

**Bits [19:0]**

Reserved, RES0.

## Accessing the CPACR_EL1:

To access the CPACR_EL1:

```
MRS <Xt>, CPACR_EL1 ; Read CPACR_EL1 into Xt
MSR CPACR_EL1, <Xt> ; Write Xt to CPACR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0001 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.18 CPTR_EL2, Architectural Feature Trap Register (EL2)

The CPTR_EL2 characteristics are:

**Purpose**

Controls trapping to EL2 of access to CPACR, CPACR_EL1, Trace functionality and registers associated with floating-point and Advanced SIMD execution. Also controls EL2 access to this functionality.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CPTR_EL3.TCPAC==1, EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register CPTR_EL2 is architecturally mapped to AArch32 System register HCPTR.

If EL2 is not implemented, this register is RES0 from EL3.

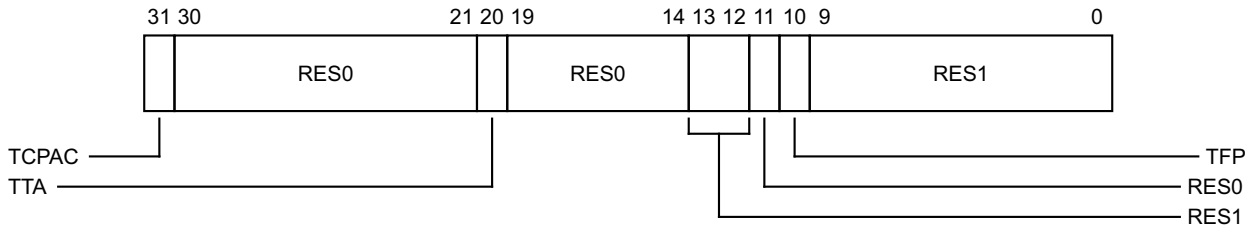
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CPTR_EL2 is a 32-bit register.

#### Field descriptions

The CPTR_EL2 bit assignments are:




**TCPAC, bit [31]**

Traps Non-secure EL1 accesses to the CPACR_EL1 or CPACR to EL2, from both Execution states.

0 This control has no effect on Non-secure EL1 accesses to the CPACR_EL1 or CPACR.

1 Non-secure EL1 accesses to the CPACR_EL1 or CPACR are trapped to EL2.

**Note**

The CPACR_EL1 or CPACR is not accessible at EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [30:21]**

Reserved, RES0.

**TTA, bit [20]**

Traps Non-secure System register accesses to the trace registers to EL2, from both Execution states.

0 Non-secure System register accesses to the trace registers are not trapped to EL2.

1 Any attempt at EL2, or Non-secure EL0 or EL1, to execute a System register access to a trace register is trapped to EL2, subject to the exception prioritization rules.

**Note**

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED. A resulting Undefined Instruction exception is higher priority than a CPTR_EL2.TTA Trap exception.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

**Bits [19:14]**

Reserved, RES0.

**Bits [13:12]**

Reserved, RES1.

**Bit [11]**

Reserved, RES0.

**TFP, bit [10]**

Traps Non-secure accesses to SIMD and floating-point functionality to EL2, from both Execution states.

0 Does not cause any instruction to be trapped.

1 Any attempt at EL2, or Non-secure EL0 or EL1, to execute an instruction that uses the registers associated with Advanced SIMD and floating-point execution is trapped to EL2, subject to the exception prioritization rules.

**Bits [9:0]**

Reserved, RES1.

## Accessing the CPTR_EL2:

To access the CPTR_EL2:

```
MRS <Xt>, CPTR_EL2 ; Read CPTR_EL2 into Xt
MSR CPTR_EL2, <Xt> ; Write Xt to CPTR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0001 | 0001 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.19 CPTR_EL3, Architectural Feature Trap Register (EL3)

The CPTR_EL3 characteristics are:

**Purpose**

Controls trapping to EL3 of access to CPACR_EL1, CPTR_EL2, Trace functionality and registers associated with floating-point and Advanced SIMD execution. Also controls EL3 access to Trace functionality and registers associated with floating-point and Advanced SIMD execution.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

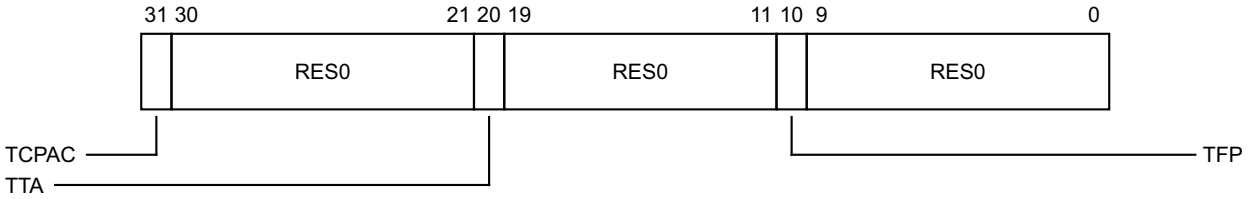
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CPTR_EL3 is a 32-bit register.

#### Field descriptions

The CPTR_EL3 bit assignments are:




**TCPAC, bit [31]**

Traps all of the following to EL3, from both Security states and both Execution states.

- EL2 accesses to the CPTR_EL2 or HCPTR.
- EL2 and EL1 accesses to the CPACR_EL1 or CPACR.

When CPTR_EL3.TCPAC is:

0 This control does not cause any accesses to CPACR_EL1, CPTR_EL2, CPACR, or HCPTR to trap to EL3.

1 EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR, are trapped to EL3 if they are not trapped to EL2 by the CPTR_EL2.TCPAC control.

**Bits [30:21]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TTA, bit [20]**

Traps System register accesses to the trace registers, from all Exception levels, both Security states, and both Execution states, to EL3.

0 Does not cause any instruction to be trapped.

1 Any System register access to the trace registers is trapped to EL3, subject to the exception prioritization rules.

If System register access to trace functionality is not supported, this bit is RES0.

**Bits [19:11]**

Reserved, RES0.

**TFP, bit [10]**

Traps accesses to SIMD and floating-point functionality, from all Exception levels, both Security states, and both Execution states, to EL3.

0 Does not cause any instruction to be trapped.

1 Any attempt at any Exception level to execute an instruction that uses the registers associated with Advanced SIMD and floating-point execution is trapped to EL3, subject to the exception prioritization rules.

**Bits [9:0]**

Reserved, RES0.

## Accessing the CPTR_EL3:

To access the CPTR_EL3:

```
MRS <Xt>, CPTR_EL3 ; Read CPTR_EL3 into Xt
MSR CPTR_EL3, <Xt> ; Write Xt to CPTR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0001 | 0001 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.20 CSSELR_EL1, Cache Size Selection Register

The CSSELR_EL1 characteristics are:

**Purpose**

Selects the current Cache Size ID Register, CCSIDR_EL1, by specifying the required cache level and the cache type (either instruction or data cache).

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

If CSSELR_EL1.Level is programmed to a cache level that is not implemented, then a read of CSSELR_EL1 is CONSTRAINED UNPREDICTABLE, and returns UNKNOWN values for CSSELR_EL1.{Level, InD}.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID2==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register CSSELR_EL1 is architecturally mapped to AArch32 System register CSSELR.

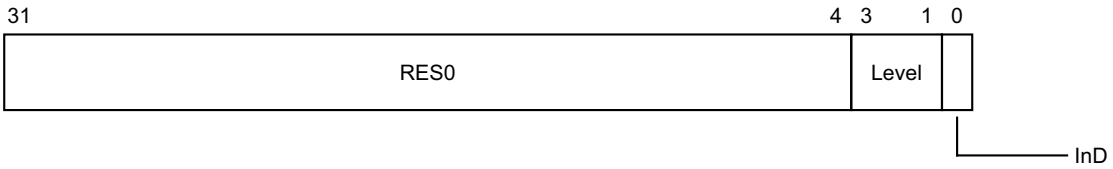
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CSSELR_EL1 is a 32-bit register.

#### Field descriptions

The CSSELR_EL1 bit assignments are:




**Bits [31:4]**

Reserved, RES0.

**Level, bits [3:1]**

Cache level of required cache. Permitted values are:

`000` Level 1 cache

`001` Level 2 cache

`010` Level 3 cache

`011` Level 4 cache

`100` Level 5 cache


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

101 Level 6 cache

110 Level 7 cache

See the Usage constraints for the behavior when the Level is programmed to a reserved value, or a cache level that is not implemented.

**InD, bit [0]**

Instruction not Data bit. Permitted values are:

0 Data or unified cache.

1 Instruction cache.

### Accessing the CSSELR_EL1:

To access the CSSELR_EL1:

```
MRS <Xt>, CSSELR_EL1 ; Read CSSELR_EL1 into Xt
MSR CSSELR_EL1, <Xt> ; Write Xt to CSSELR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 010 | 0000 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.21 CTR_EL0, Cache Type Register

The CTR_EL0 characteristics are:

**Purpose**

Provides information about the architecture of the caches.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID2==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If SCTLR_EL1.UCT==0, EL0 read accesses to this register are trapped to EL1.

**Configurations**

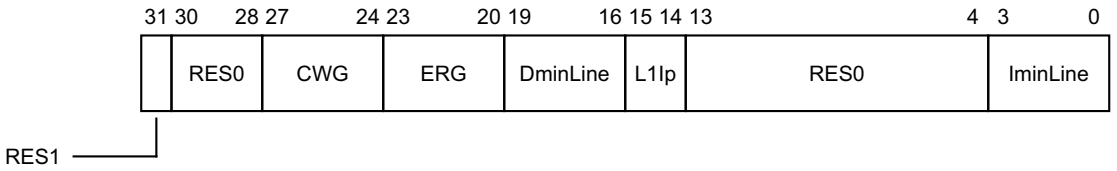
AArch64 System register CTR_EL0 is architecturally mapped to AArch32 System register CTR.

**Attributes**

CTR_EL0 is a 32-bit register.

#### Field descriptions

The CTR_EL0 bit assignments are:




**Bit [31]**

Reserved, RES1.

**Bits [30:28]**

Reserved, RES0.

**CWG, bits [27:24]**

Cache Writeback Granule. $Log_2$ of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

A value of `0b0000` indicates that this register does not provide Cache Writeback Granule information and either:

- The architectural maximum of 512 words (2KB) must be assumed.
- The Cache Writeback Granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than `0b1001` are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**ERG, bits [23:20]**

Exclusives Reservation Granule. $Log_2$ of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

A value of `0b0000` indicates that this register does not provide Exclusives Reservation Granule information and the architectural maximum of 512 words (2KB) must be assumed.

Values greater than `0b1001` are reserved.

**DminLine, bits [19:16]**

$Log_2$ of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

**L1Ip, bits [15:14]**

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache. Possible values of this field are:

`01` ASID-tagged Virtual Index, Virtual Tag (AIVIVT)

`10` Virtual Index, Physical Tag (VIPT)

`11` Physical Index, Physical Tag (PIPT)

Other values are reserved.

**Bits [13:4]**

Reserved, RES0.

**IminLine, bits [3:0]**

$Log_2$ of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

## Accessing the CTR_EL0:

To access the CTR_EL0:

```
MRS <Xt>, CTR_EL0 ; Read CTR_EL0 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 0000 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.22 DACR32_EL2, Domain Access Control Register

The DACR32_EL2 characteristics are:

**Purpose**

Allows access to the AArch32 DACR register from AArch64 state only. Its value has no effect on execution in AArch64 state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register DACR32_EL2 is architecturally mapped to AArch32 System register DACR.

If EL1 does not support AArch32, this register is UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

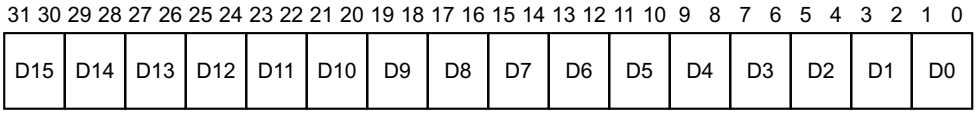
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

DACR32_EL2 is a 32-bit register.

#### Field descriptions

The DACR32_EL2 bit assignments are:




**D<n>, bits [2n+1:2n], for n = 0 to 15**

Domain n access permission, where n = 0 to 15. Permitted values are:

`00` No access. Any access to the domain generates a Domain fault.

`01` Client. Accesses are checked against the permission bits in the translation tables.

`11` Manager. Accesses are not checked against the permission bits in the translation tables.

The value `10` is reserved.

#### Accessing the DACR32_EL2:

To access the DACR32_EL2:

```
MRS <Xt>, DACR32_EL2 ; Read DACR32_EL2 into Xt
MSR DACR32_EL2, <Xt> ; Write Xt to DACR32_EL2
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0011 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.23 DCZID_EL0, Data Cache Zero ID register

The DCZID_EL0 characteristics are:

**Purpose**

Indicates the block size that is written with byte values of 0 by the DC ZVA (Data Cache Zero by Address) system instruction.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RO | RO | RO | RO | RO | RO |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

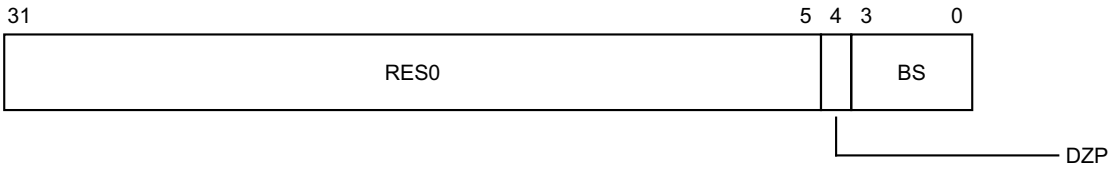
There are no configuration notes.

**Attributes**

DCZID_EL0 is a 32-bit register.

#### Field descriptions

The DCZID_EL0 bit assignments are:




**Bits [31:5]**

Reserved, RES0.

**DZP, bit [4]**

Data Zero prohibited. Permitted values are:

0 DC ZVA instruction is permitted.

1 DC ZVA instruction is prohibited.

The value read from this field is governed by the access state and the values of the HCR_EL2.TDZ and SCTLR_EL1.DZE bits.

**BS, bits [3:0]**

$Log_2$ of the block size in words. The maximum size supported is 2KB (value == 9).

#### Accessing the DCZID_EL0:

To access the DCZID_EL0:

```
MRS <Xt>, DCZID_EL0 ; Read DCZID_EL0 into Xt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 0000 | 0000 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.24 ESR_EL1, Exception Syndrome Register (EL1)

The ESR_EL1 characteristics are:

**Purpose**

Holds syndrome information for an exception taken to EL1.

See also *Exception classes and the ESR_ELx syndrome registers* on page D1-1539.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

ESR_EL1 is made UNKNOWN as a result of an exception return from EL1.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL1, the value of ESR_EL1 is UNKNOWN. The value written to ESR_EL1 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ESR_EL1 is architecturally mapped to AArch32 System register DFSR.

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

ESR_EL1 is a 32-bit register.

#### Field descriptions

See ESR_ELx.

#### Accessing the ESR_EL1:

To access the ESR_EL1:

```
MRS <Xt>, ESR_EL1 ; Read ESR_EL1 into Xt
MSR ESR_EL1, <Xt> ; Write Xt to ESR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0101 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.25 ESR_EL2, Exception Syndrome Register (EL2)

The ESR_EL2 characteristics are:

**Purpose**

Holds syndrome information for an exception taken to EL2.

See also *Exception classes and the ESR_ELx syndrome registers* on page D1-1539.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

ESR_EL2 is made UNKNOWN as a result of an exception return from EL2.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of ESR_EL2 is UNKNOWN. The value written to ESR_EL2 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register ESR_EL2 is architecturally mapped to AArch32 System register HSR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

ESR_EL2 is a 32-bit register.

### Field descriptions

See ESR_ELx.

### Accessing the ESR_EL2:

To access the ESR_EL2:

```
MRS <Xt>, ESR_EL2 ; Read ESR_EL2 into Xt
MSR ESR_EL2, <Xt> ; Write Xt to ESR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0101 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.26 ESR_EL3, Exception Syndrome Register (EL3)

The ESR_EL3 characteristics are:

**Purpose**

Holds syndrome information for an exception taken to EL3.

See also *Exception classes and the ESR_ELx syndrome registers* on page D1-1539.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

ESR_EL3 is made UNKNOWN as a result of an exception return from EL3.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL3, the value of ESR_EL3 is UNKNOWN. The value written to ESR_EL3 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

ESR_EL3 is a 32-bit register.

#### Field descriptions

See ESR_ELx.

#### Accessing the ESR_EL3:

To access the ESR_EL3:

```
MRS <Xt>, ESR_EL3 ; Read ESR_EL3 into Xt
MSR ESR_EL3, <Xt> ; Write Xt to ESR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0101 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.27 ESR_ELx, Exception Syndrome Register (ELx)

This page describes ESR_EL1, ESR_EL2, and ESR_EL3.

The ESR_ELx characteristics are:

**Purpose**

Holds syndrome information for an exception taken to ELx.

See also *Exception classes and the ESR_ELx syndrome registers* on page D1-1539.

**Usage constraints**

ESR_ELx is made UNKNOWN as a result of an exception return from ELx.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to ELx, the value of ESR_ELx is UNKNOWN. The value written to ESR_ELx must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

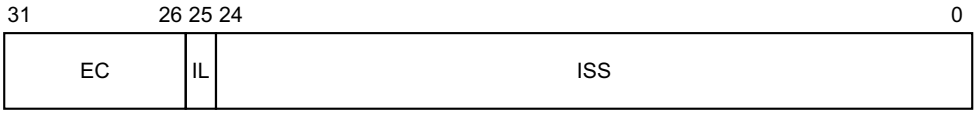
If EL2 is not implemented, ESR_EL2 is RES0 from EL3.

**Attributes**

The ESR_ELx registers are 32-bit registers.

### Field descriptions

The ESR_ELx bit assignments are:




**EC, bits [31:26]**

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

**EC == 000000, Exceptions with an unknown reason**

This value is valid for all described registers.

Unknown or Uncategorized Reason - generally used for exceptions as a result of erroneous execution.

See *ISS encoding for exceptions with an unknown reason* on page D7-1956.

**EC == 000001, Exception from a WFI or WFE instruction**

This value is valid for all described registers.

Exceptions from WFE or WFI from either AArch32 or AArch64 as a result of configurable traps, enables, or disables.

Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.

See *ISS encoding for an exception from a WFI or WFE instruction* on page D7-1958.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EC == 000011, Exception from an MCR or MRC access**

This value is valid for all described registers.

Exceptions from MCR/MRC to CP15 from AArch32 as a result of configurable traps, enables, or disables that do not use code `0b000000`.

See *ISS encoding for an exception from an MCR or MRC access* on page D7-1959.

**EC == 000100, Exception from an MCRR or MRRC access**

This value is valid for all described registers.

Exceptions from MCRR/MRRC to CP15 from AArch32 as a result of configurable traps, enables, or disables that do not use code `0b000000`.

See *ISS encoding for an exception from an MCRR or MRRC access* on page D7-1961.

**EC == 000101, Exception from an MCR or MRC access**

This value is valid for all described registers.

Exceptions from MCR/MRC to CP14 from AArch32 as a result of configurable traps, enables, or disables that do not use code `0b000000`.

See *ISS encoding for an exception from an MCR or MRC access* on page D7-1959.

**EC == 000110, Exception from an LDC or STC access to CP14**

This value is valid for all described registers.

Exceptions from LDC/STC to CP14 from AArch32 as a result of configurable traps, enables, or disables.

The only architected uses of these instructions to access CP14 are:

- An STC to write to DBGDTRRX_EL0 or DBGDTRRXint.
- An LDC to read from DBGDTRTX_EL0 or DBGDTRTXint.

See *ISS encoding for an exception from an LDC or STC access to CP14* on page D7-1963.

**EC == 000111, Exception from an access to an Advanced SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP**

This value is valid for all described registers.

Exceptions from an access to an Advanced SIMD or Floating Point register as a result of CPACR_EL1.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP.

Excludes exceptions resulting from an HCR_EL2.TGE value of 1 or when floating-point and Advanced SIMD are not implemented. These are reported with EC value `0b000000` as described in *EC encodings when routing exceptions to EL2* on page D1-1550.

See *ISS encoding for an exception from an access to an Advanced SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP* on page D7-1965.

**EC == 001000, Exception from an MCR or MRC access**

This value is valid for ESR_EL2.

Exceptions from MRC (or VMRS) to CP10 for MVFR0, MVFR1, MVFR2, or FPSID as a result of ID Group traps from AArch32 unless reported using code `0b000111`.

See *ISS encoding for an exception from an MCR or MRC access* on page D7-1959.

**EC == 001100, Exception from an MCRR or MRRC access**

This value is valid for all described registers.

Exceptions from MCRR/MRRC to CP14 from AArch32 as a result of configurable traps, enables, or disables.

See *ISS encoding for an exception from an MCRR or MRRC access* on page D7-1961.

**EC == 001110, Exception from an Illegal Execution state, or a PC or SP alignment fault**

This value is valid for all described registers.

Exceptions that occur because the value of PSTATE.IL is 1.

See *ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault* on page D7-1967.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EC == 010001, Exception from HVC or SVC instruction execution**

This value is valid for ESR_EL1 and ESR_EL2.

SVC executed in AArch32 state.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TGE is 1.

See *ISS encoding for an exception from HVC or SVC instruction execution* on page D7-1967.

**EC == 010010, Exception from HVC or SVC instruction execution**

This value is valid for ESR_EL2.

HVC that is not disabled executed in AArch32 state.

See *ISS encoding for an exception from HVC or SVC instruction execution* on page D7-1967.

**EC == 010011, Exception from SMC instruction execution in AArch32 state**

This value is valid for ESR_EL2 and ESR_EL3.

SMC that is not disabled executed in AArch32 state.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.

See *ISS encoding for an exception from SMC instruction execution in AArch32 state* on page D7-1968.

**EC == 010101, Exception from HVC or SVC instruction execution**

This value is valid for all described registers.

SVC executed in AArch64 state.

See *ISS encoding for an exception from HVC or SVC instruction execution* on page D7-1967.

**EC == 010110, Exception from HVC or SVC instruction execution**

This value is valid for ESR_EL2 and ESR_EL3.

HVC that is not disabled executed in AArch64 state.

See *ISS encoding for an exception from HVC or SVC instruction execution* on page D7-1967.

**EC == 010111, Exception from SMC instruction execution in AArch64 state**

This value is valid for ESR_EL2 and ESR_EL3.

SMC that is not disabled executed in AArch64 state.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.

See *ISS encoding for an exception from SMC instruction execution in AArch64 state* on page D7-1969.

**EC == 011000, Exception from MSR, MRS, or System instruction execution in AArch64 state**

This value is valid for all described registers.

Exceptions from MSR, MRS, or System AArch64 instructions as a result of configurable traps, enables, or disables, except those reported using EC values `000000`, `000001`, or `000111`.

This include all instructions that cause exceptions that are part of the encoding space defined in *System instruction class encoding overview* on page C5-267, except for those exceptions reported using EC values `0b000000`, `0b000001`, or `0b000111`.

See *ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state* on page D7-1970.

**EC == 011111, IMPLEMENTATION DEFINED exception to EL3**

This value is valid for ESR_EL3.

Reserved for IMPLEMENTATION DEFINED exceptions to EL3.

See *ISS encoding for a IMPLEMENTATION DEFINED exception to EL3* on page D7-1971.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EC == 100000, Exception from an Instruction abort**

This value is valid for all described registers.

Instruction Abort that caused entry from a lower Exception level, where that Exception level could be using AArch64 or using AArch32.

Used for MMU faults generated by instruction accesses and Synchronous external aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See *ISS encoding for an exception from an Instruction abort* on page D7-1972.

**EC == 100001, Exception from an Instruction abort**

This value is valid for all described registers.

Instruction Abort from the current Exception level, where the current Exception level must be using AArch64.

Used for MMU faults generated by instruction accesses and Synchronous external aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See *ISS encoding for an exception from an Instruction abort* on page D7-1972.

**EC == 100010, Exception from an Illegal Execution state, or a PC or SP alignment fault**

This value is valid for all described registers.

PC alignment fault.

See *ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault* on page D7-1967.

**EC == 100100, Exception from a Data abort**

This value is valid for all described registers.

Data Abort that caused entry from a lower Exception level, where that Exception level could be using AArch64 or using AArch32.

Used for MMU faults generated by data accesses, alignment faults other than those caused by the Stack Pointer misalignment, and Synchronous external aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See *ISS encoding for an exception from a Data abort* on page D7-1974.

**EC == 100101, Exception from a Data abort**

This value is valid for all described registers.

Data Abort that caused entry from a current Exception level, where the current Exception level must be using AArch64.

Used for MMU faults generated by data accesses, alignment faults other than those caused by the Stack Pointer misalignment, and Synchronous external aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

See *ISS encoding for an exception from a Data abort* on page D7-1974.

**EC == 100110, Exception from an Illegal Execution state, or a PC or SP alignment fault**

This value is valid for all described registers.

SP alignment fault.

See *ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault* on page D7-1967.

**EC == 101000, Exception from a trapped Floating-point exception**

This value is valid for ESR_EL1 and ESR_EL2.

Exceptions as a result of Floating-point exception from AArch32.

Whether this Exception class is supported is IMPLEMENTATION DEFINED.

See *ISS encoding for an exception from a trapped Floating-point exception* on page D7-1977.

**EC == 101100, Exception from a trapped Floating-point exception**

This value is valid for all described registers.

Exceptions as a result of Floating-point exception from AArch64.

Whether this Exception class is supported is IMPLEMENTATION DEFINED.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

See *ISS encoding for an exception from a trapped Floating-point exception* on page D7-1977.

**EC == 101111, SError interrupt**

This value is valid for all described registers.

SError Interrupt.

See *ISS encoding for an SError interrupt* on page D7-1979.

**EC == 110000, Exception from a Breakpoint or Vector Catch debug event**

This value is valid for ESR_EL1 and ESR_EL2.

Breakpoint debug event that caused entry from a lower Exception level.

See *ISS encoding for an exception from a Breakpoint or Vector Catch debug event* on page D7-1980.

**EC == 110001, Exception from a Breakpoint or Vector Catch debug event**

This value is valid for ESR_EL1 and ESR_EL2.

Breakpoint debug event from the current Exception level.

See *ISS encoding for an exception from a Breakpoint or Vector Catch debug event* on page D7-1980.

**EC == 110010, Exception from a Software Step debug event**

This value is valid for ESR_EL1 and ESR_EL2.

Software step debug event that caused entry from a lower Exception level.

See *ISS encoding for an exception from a Software Step debug event* on page D7-1980.

**EC == 110011, Exception from a Software Step debug event**

This value is valid for ESR_EL1 and ESR_EL2.

Software step debug event from the current Exception level.

See *ISS encoding for an exception from a Software Step debug event* on page D7-1980.

**EC == 110100, Exception from a Watchpoint debug event**

This value is valid for ESR_EL1 and ESR_EL2.

Watchpoint debug event that caused entry from a lower Exception level.

See *ISS encoding for an exception from a Watchpoint debug event* on page D7-1981.

**EC == 110101, Exception from a Watchpoint debug event**

This value is valid for ESR_EL1 and ESR_EL2.

Watchpoint debug event from the current Exception level.

See *ISS encoding for an exception from a Watchpoint debug event* on page D7-1981.

**EC == 111000, Exception from execution of a Software Breakpoint instruction**

This value is valid for ESR_EL1 and ESR_EL2.

BKPT instruction executed in AArch32 state.

See *ISS encoding for an exception from execution of a Software Breakpoint instruction* on page D7-1982.

**EC == 111010, Exception from a Breakpoint or Vector Catch debug event**

This value is valid for ESR_EL2.

AArch32 state Vector catch debug event.

The only case where an exception from a Vector catch debug event is taken to an Exception level that is using AArch64 is when the Vector catch is routed to EL2 and EL2 is using AArch64.

See *ISS encoding for an exception from a Breakpoint or Vector Catch debug event* on page D7-1980.

**EC == 111100, Exception from execution of a Software Breakpoint instruction**

This value is valid for all described registers.

BRK instruction executed in AArch64 state.

This is reported in ESR_EL3 only if a BRK instruction is executed at EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See *ISS encoding for an exception from execution of a Software Breakpoint instruction* on page D7-1982.

All other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

**IL, bit [25]**

Instruction Length for synchronous exceptions. Possible values of this bit are:

`0` 16-bit instruction trapped.

`1` 32-bit instruction trapped. This value is also used when the exception is one of the following:

- An SError interrupt.
- An Instruction Abort exception.
- A PC alignment fault exception.
- An SP alignment fault exception.
- A Data Abort exception for which the value of the ISV bit is 0.
- An Illegal Execution state exception.
- Any debug exception except for Software Breakpoint instruction exceptions. For Software Breakpoint instruction exceptions, this bit has its standard meaning:

  `0` 16-bit T32 BKPT instruction.

  `1` 32-bit A32 BKPT instruction or A64 BRK instruction.
- An exception reported using EC value `0b000000`.

**ISS, bits [24:0]**

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number. For an exception taken from AArch32 state, *Mapping of the general-purpose registers between the Execution states* on page D1-1620 defines this view of the specified AArch32 register. If the AArch32 register descriptor is `0b1111`, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value `0b11111`.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  - The value `0b11111`.

When the EC field is `0b000000`, indicating an exception with an unknown reason, the ISS field is not valid, RES0.

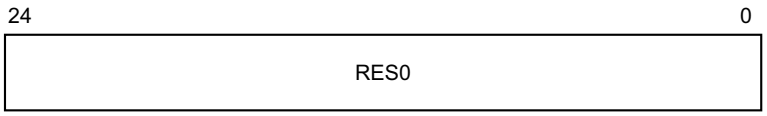
The following subsections describe each ISS format.

### *ISS encoding for exceptions with an unknown reason*

This encoding is used by Unknown or Uncategorized Reason - generally used for exceptions as a result of erroneous execution.

The ISS encoding for these exceptions is:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [24:0]**

Reserved, RES0.

An exception with an unknown reason occurs in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes at the current Exception level and Security state.
  - Instruction encodings for instructions not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is unallocated in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is unallocated in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction in Non-secure state from EL0 when the value of HCR_EL2.TGE is 1.
  - A DCPS2 instruction from EL1 or EL0 when the value of SCR_EL3.NS is 0, or when EL2 is not implemented.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon. See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-1602.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (Banked register) or an MSR (Banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of `0b000111`.

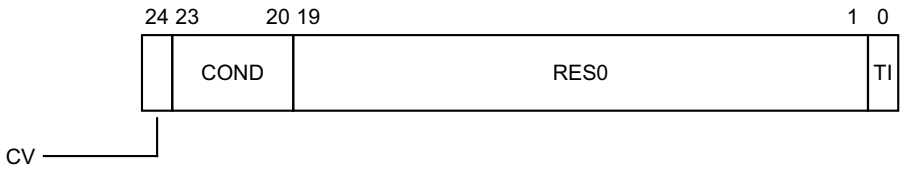

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *ISS encoding for an exception from a WFI or WFE instruction*

This encoding is used by Exceptions from WFE or WFI from either AArch32 or AArch64 as a result of configurable traps, enables, or disables.

Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.

The ISS encoding for these exceptions is:




**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to `0b1110`.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

**Bits [19:1]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TI, bit [0]**

Trapped instruction. Possible values of this bit are:

`0` WFI trapped.

`1` WFE trapped.

The following sections describe configuration settings for generating this exception:

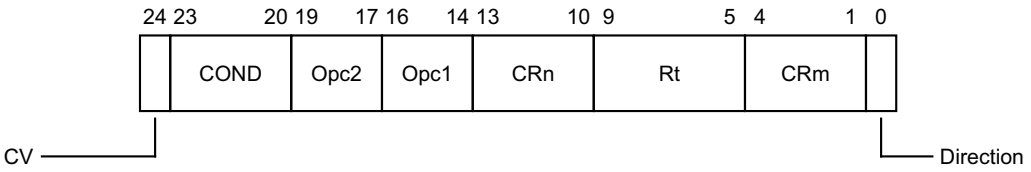
- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-1578.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page D1-1594.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-1603.

### *ISS encoding for an exception from an MCR or MRC access*

This encoding is used by:

- Exceptions from MCR/MRC to CP15 from AArch32 as a result of configurable traps, enables, or disables that do not use code `0b000000`.
- Exceptions from MCR/MRC to CP14 from AArch32 as a result of configurable traps, enables, or disables that do not use code `0b000000`.
- Exceptions from MRC (or VMRS) to CP10 for MVFR0, MVFR1, MVFR2, or FPSID as a result of ID Group traps from AArch32 unless reported using code `0b000111`.

The ISS encoding for these exceptions is:




**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to `0b1110`.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

**Opc2, bits [19:17]**

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value 0b000.

**Opc1, bits [16:14]**

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value 0b111.

**CRn, bits [13:10]**

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

**Rt, bits [9:5]**

The Rt value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-1620.

**CRm, bits [4:1]**

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value 0b0000.

**Direction, bit [0]**

Indicates the direction of the trapped instruction. The possible values of this bit are:

0 Write to coprocessor. MCR instruction.

1 Read from coprocessor. MRC or VMRS instruction.

The following sections describe configuration settings for generating exceptions that are reported using EC value 0b000011:

- *Traps to EL1 of EL0 accesses to the Generic Timer registers* on page D1-1582.
- *Traps to EL1 of EL0 accesses to Performance Monitors registers* on page D1-1582.
- *Traps to EL2 of Non-secure EL1 accesses to virtual memory control registers* on page D1-1585.
- *Traps to EL2 of Non-secure EL1 execution of TLB maintenance instructions* on page D1-1587.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of cache maintenance instructions* on page D1-1587.
- *Traps to EL2 of Non-secure EL1 accesses to the Auxiliary Control Register* on page D1-1589.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page D1-1589.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Traps to EL2 of Non-secure EL0 and EL1 accesses to the ID registers* on page D1-1591.
- *Trapping to EL2 of Non-secure EL1 accesses to the CPACR_EL1 or CPACR* on page D1-1595.
- *General trapping to EL2 of Non-secure EL0 and EL1 accesses to System registers, from AArch32 state only* on page D1-1596.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to the Generic Timer registers* on page D1-1600.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page D1-1600.
- *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-1602.
- *Trapping to EL3 of EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR* on page D1-1605.
- *Traps to EL3 of EL2, EL1, and EL0 accesses to Performance Monitors registers* on page D1-1609.

The following sections describe configuration settings for generating exceptions that are reported using EC value `0b000101`:

- *Traps to EL1 of EL0 and EL1 System register accesses to the trace registers* on page D1-1580.
- *Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers* on page D1-1581.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to the ID registers* on page D1-1591, for trapped accesses to the JIDR.
- *Traps to EL2 of Non-secure System register accesses to the trace registers* on page D1-1596.
- *Trapping System register accesses to Debug ROM registers to EL2* on page D1-1598.
- *Trapping System register accesses to powerdown debug registers to EL2* on page D1-1598.
- *Trapping general System register accesses to debug registers to EL2* on page D1-1599.
- *Traps to EL3 of all System register accesses to the trace registers* on page D1-1606.
- *Trapping System register accesses to powerdown debug registers to EL3* on page D1-1607.
- *Trapping general System register accesses to debug registers to EL3* on page D1-1608.

*Traps to EL2 of Non-secure EL0 and EL1 accesses to the ID registers* on page D1-1591 describes configuration settings for generating exceptions that are reported using EC value `0b001000`.

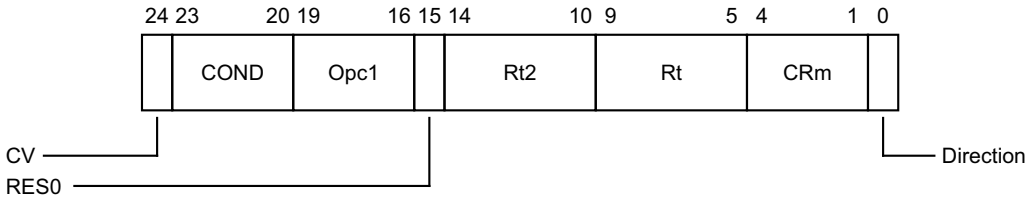
### *ISS encoding for an exception from an MCRR or MRRC access*

This encoding is used by:

- Exceptions from MCRR/MRRC to CP15 from AArch32 as a result of configurable traps, enables, or disables that do not use code `0b000000`.
- Exceptions from MCRR/MRRC to CP14 from AArch32 as a result of configurable traps, enables, or disables.

The ISS encoding for these exceptions is:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to `0b1110`.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

**Opc1, bits [19:16]**

The Opc1 value from the issued instruction.

**Bit [15]**

Reserved, RES0.

**Rt2, bits [14:10]**

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-1620.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Rt, bits [9:5]**

The Rt value from the issued instruction, the first general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-1620.

**CRm, bits [4:1]**

The CRm value from the issued instruction.

**Direction, bit [0]**

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0` Write to coprocessor. MCRR instruction.

`1` Read from coprocessor. MRRC instruction.

The following sections describe configuration settings for generating exceptions that are reported using EC value `0b000100`:

- *Traps to EL1 of EL0 accesses to the Generic Timer registers* on page D1-1582.
- *Traps to EL1 of EL0 accesses to Performance Monitors registers* on page D1-1582.
- *Traps to EL2 of Non-secure EL1 accesses to virtual memory control registers* on page D1-1585.
- *General trapping to EL2 of Non-secure EL0 and EL1 accesses to System registers, from AArch32 state only* on page D1-1596.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to the Generic Timer registers* on page D1-1600.
- *Traps to EL2 of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page D1-1600.
- *Traps to EL3 of EL2, EL1, and EL0 accesses to Performance Monitors registers* on page D1-1609.

The following sections describe configuration settings for generating exceptions that are reported using EC value `0b001100`:

- *Traps to EL1 of EL0 and EL1 System register accesses to the trace registers* on page D1-1580.
- *Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers* on page D1-1581.
- *Traps to EL2 of Non-secure System register accesses to the trace registers* on page D1-1596.
- *Trapping System register accesses to Debug ROM registers to EL2* on page D1-1598.
- *Traps to EL3 of all System register accesses to the trace registers* on page D1-1606.
- *Trapping System register accesses to powerdown debug registers to EL2* on page D1-1598.

### *ISS encoding for an exception from an LDC or STC access to CP14*

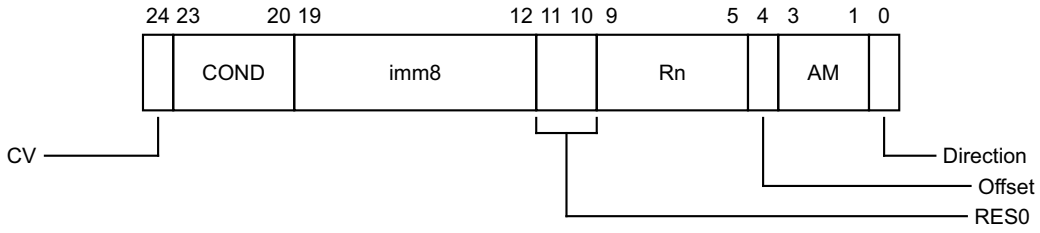
This encoding is used by Exceptions from LDC/STC to CP14 from AArch32 as a result of configurable traps, enables, or disables.

The only architected uses of these instructions to access CP14 are:

- An STC to write to DBGDTRRX_EL0 or DBGDTRRXint.
- An LDC to read from DBGDTRTX_EL0 or DBGDTRTXint.

The ISS encoding for these exceptions is:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to `0b1110`.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

**imm8, bits [19:12]**

The immediate value from the issued instruction.

**Bits [11:10]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Rn, bits [9:5]**

The Rn value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-1620.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

**Offset, bit [4]**

Indicates whether the offset is added or subtracted:

`0` Subtract offset.

`1` Add offset.

This bit corresponds to the U bit in the instruction encoding.

**AM, bits [3:1]**

Addressing mode. The permitted values of this field are:

`000` Immediate unindexed.

`001` Immediate post-indexed.

`010` Immediate offset.

`011` Immediate pre-indexed.

`100` Literal unindexed.
A32 instruction set only.
For a trapped LDC T32 instruction or STC T32 instruction, this encoding is reserved.

`110` Literal offset.
For the STC instruction, valid only in the A32 instruction set.
For a trapped STC T32 instruction, this encoding is reserved.

The values `0b101` and `0b111` are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

**Direction, bit [0]**

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0` Write to memory. STC instruction.

`1` Read from memory. LDC instruction.

The following sections describe the configuration settings for the traps that are reported using EC value `0b000110`:

- *Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers* on page D1-1581.
- *Trapping general System register accesses to debug registers to EL2* on page D1-1599.
- *Trapping general System register accesses to debug registers to EL3* on page D1-1608.

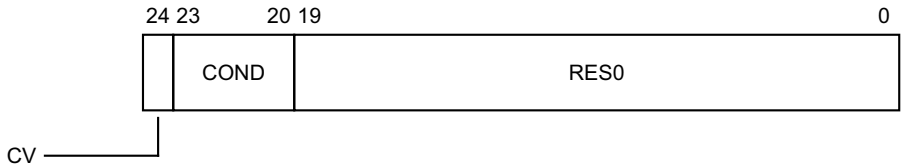
### *ISS encoding for an exception from an access to an Advanced SIMD or floating-point register, resulting from CPACR_EL1.FPEN or CPTR_ELx.TFP*

This encoding is used by Exceptions from an access to an Advanced SIMD or Floating Point register as a result of CPACR_EL1.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP.

Excludes exceptions resulting from an HCR_EL2.TGE value of 1 or when floating-point and Advanced SIMD are not implemented. These are reported with EC value `0b000000` as described in *EC encodings when routing exceptions to EL2* on page D1-1550.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The ISS encoding for these exceptions is:




The accesses covered by this trap include:

- Execution of Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.

For an implementation that does not include floating-point or Advanced SIMD, the exception is reported using the EC value 000000.

**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to `0b1110`.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

**Bits [19:0]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following sections describe the configuration settings for the traps that are reported using EC value 0b000111:

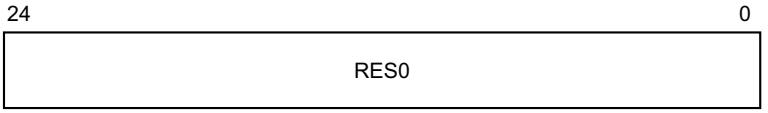
- *Traps to EL1 of EL0 and EL1 accesses to SIMD and floating-point functionality* on page D1-1581.
- *General trapping to EL2 of Non-secure accesses to the SIMD and floating-point registers* on page D1-1595.
- *Traps to EL3 of all System register accesses to the trace registers* on page D1-1606.

### ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

This encoding is used by:

- Exceptions that occur because the value of PSTATE.IL is 1.
- PC alignment fault.
- SP alignment fault.

The ISS encoding for these exceptions is:




**Bits [24:0]**

Reserved, RES0.

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. *Stack pointer alignment checking* on page D1-1531 describes the configuration settings for generating SP alignment fault exceptions.

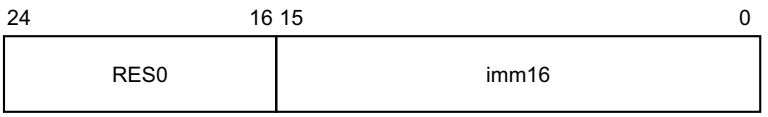
### ISS encoding for an exception from HVC or SVC instruction execution

This encoding is used by:

- SVC executed in AArch32 state.

  This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TGE is 1.
- HVC that is not disabled executed in AArch32 state.
- SVC executed in AArch64 state.
- HVC that is not disabled executed in AArch64 state.

The ISS encoding for these exceptions is:




**Bits [24:16]**

Reserved, RES0.

**imm16, bits [15:0]**

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an A32 or T32 SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

For T32 and A32 instructions, see SVC, and HVC.

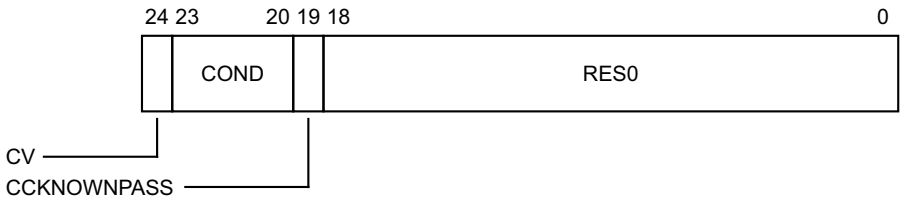
For A64 instructions, see SVC and HVC.

### *ISS encoding for an exception from SMC instruction execution in AArch32 state*

This encoding is used by SMC that is not disabled executed in AArch32 state.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.

The ISS encoding for these exceptions is:




For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from Non-secure EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

**COND, bits [23:20]**

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to `0b1110`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to `0b1110`.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to `0b1110`, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

**CCKNOWNPASS, bit [19]**

Indicates whether the instruction might have failed its condition code check.

`0` The instruction was unconditional, or was conditional and passed its condition code check.

`1` The instruction was conditional, and might have failed its condition code check.

**Note**

In an implementation in which an SMC instruction that fails it code check is not trapped, this field can always return the value 0.

**Bits [18:0]**

Reserved, RES0.

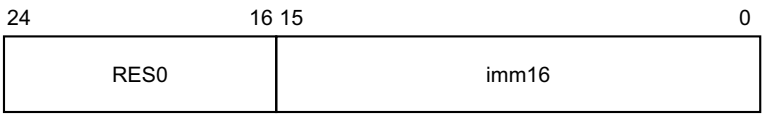
*Traps to EL2 of Non-secure EL1 execution of SMC instructions* on page D1-1590 describes the configuration settings for trapping SMC instructions from Non-secure EL1 modes, and *System calls* on page D1-1610 describes the case where these exceptions are trapped to EL3.

### *ISS encoding for an exception from SMC instruction execution in AArch64 state*

This encoding is used by SMC that is not disabled executed in AArch64 state.

This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.

The ISS encoding for these exceptions is:




**Bits [24:16]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**imm16, bits [15:0]**

The value of the immediate field from the issued SMC instruction.

The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from Non-secure EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

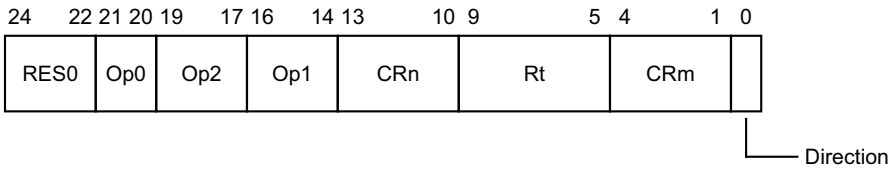
*Traps to EL2 of Non-secure EL1 execution of SMC instructions* on page D1-1590 describes the configuration settings for trapping SMC instructions from Non-secure EL1 modes, and *System calls* on page D1-1610 describes the case where these exceptions are trapped to EL3.

### *ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state*

This encoding is used by Exceptions from MSR, MRS, or System AArch64 instructions as a result of configurable traps, enables, or disables, except those reported using EC values `000000`, `000001`, or `000111`.

This include all instructions that cause exceptions that are part of the encoding space defined in *System instruction class encoding overview* on page C5-267, except for those exceptions reported using EC values `0b000000`, `0b000001`, or `0b000111`.

The ISS encoding for these exceptions is:




**Bits [24:22]**

Reserved, RES0.

**Op0, bits [21:20]**

The Op0 value from the issued instruction.

**Op2, bits [19:17]**

The Op2 value from the issued instruction.

**Op1, bits [16:14]**

The Op1 value from the issued instruction.

**CRn, bits [13:10]**

The CRn value from the issued instruction.

**Rt, bits [9:5]**

The Rt value from the issued instruction, the general-purpose register used for the transfer.

**CRm, bits [4:1]**

The CRm value from the issued instruction.

**Direction, bit [0]**

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0` Write access, including MSR instructions.

`1` Read access, including MRS instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For exceptions caused by System instructions, see *System* on page C4-193 for the encoding values returned by an instruction.

The following sections describe configuration settings for generating the exception that is reported using EC value `0b011000`:

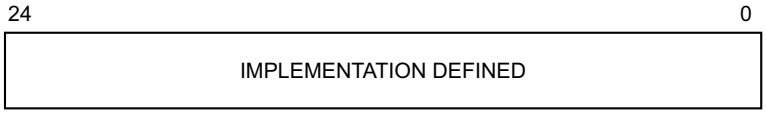
- In *EL1 configurable controls* on page D1-1576.
  - *Traps to EL1 of EL0 execution of cache maintenance instructions* on page D1-1577.
  - *Traps to EL1 of EL0 accesses to the CTR_EL0* on page D1-1577.
  - *Traps to EL1 of EL0 execution of DC ZVA instructions* on page D1-1578.
  - *Traps to EL1 of EL0 accesses to the PSTATE.{D, A, I, F} interrupt masks* on page D1-1579.
  - *Traps to EL1 of EL0 and EL1 System register accesses to the trace registers* on page D1-1580.
  - *Traps to EL1 of EL0 accesses to the Debug Communications Channel (DCC) registers* on page D1-1581.
  - *Traps to EL1 of EL0 accesses to the Generic Timer registers* on page D1-1582.
  - *Traps to EL1 of EL0 accesses to Performance Monitors registers* on page D1-1582.
- In *EL2 configurable controls* on page D1-1584.
  - *Traps to EL2 of Non-secure EL1 accesses to virtual memory control registers* on page D1-1585.
  - *Traps to EL2 of Non-secure EL0 and EL1 execution of DC ZVA instructions* on page D1-1586.
  - *Traps to EL2 of Non-secure EL1 execution of TLB maintenance instructions* on page D1-1587.
  - *Traps to EL2 of Non-secure EL0 and EL1 execution of cache maintenance instructions* on page D1-1587.
  - *Traps to EL2 of Non-secure EL1 accesses to the Auxiliary Control Register* on page D1-1589.
  - *Traps to EL2 of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page D1-1589.
  - *Traps to EL2 of Non-secure EL0 and EL1 accesses to the ID registers* on page D1-1591.
  - *Trapping to EL2 of Non-secure EL1 accesses to the CPACR_EL1 or CPACR* on page D1-1595.
  - *Traps to EL2 of Non-secure System register accesses to the trace registers* on page D1-1596.
  - *Trapping System register accesses to Debug ROM registers to EL2* on page D1-1598.
  - *Trapping System register accesses to powerdown debug registers to EL2* on page D1-1598.
  - *Traps to EL2 of Non-secure EL0 and EL1 accesses to the Generic Timer registers* on page D1-1600.
  - *Trapping general System register accesses to debug registers to EL2* on page D1-1599.
  - *Traps to EL2 of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page D1-1600.
- In *EL3 configurable controls* on page D1-1601.
  - *Traps to EL3 of Secure EL1 accesses to the Counter-timer Physical Secure timer registers* on page D1-1604.
  - *Trapping to EL3 of EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR* on page D1-1605.
  - *Traps to EL3 of all System register accesses to the trace registers* on page D1-1606.
  - *Trapping System register accesses to powerdown debug registers to EL3* on page D1-1607.
  - *Trapping general System register accesses to debug registers to EL3* on page D1-1608.
  - *Traps to EL3 of EL2, EL1, and EL0 accesses to Performance Monitors registers* on page D1-1609.

#### *ISS encoding for a IMPLEMENTATION DEFINED exception to EL3*

This encoding is used by Reserved for IMPLEMENTATION DEFINED exceptions to EL3.

The ISS encoding for these exceptions is:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**IMPLEMENTATION DEFINED, bits [24:0]**

IMPLEMENTATION DEFINED.

### *ISS encoding for an exception from an Instruction abort*

This encoding is used by:

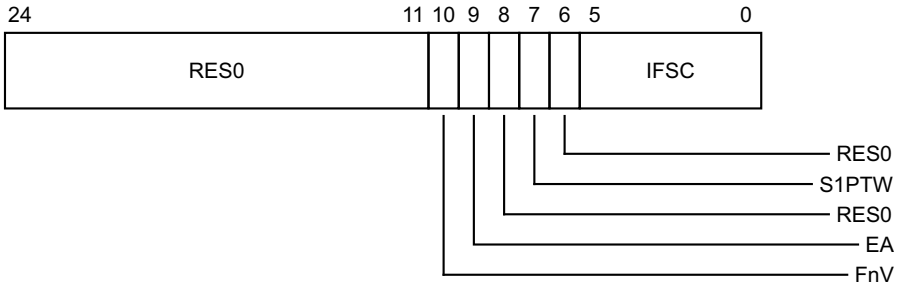
- Instruction Abort that caused entry from a lower Exception level, where that Exception level could be using AArch64 or using AArch32.

  Used for MMU faults generated by instruction accesses and Synchronous external aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

- Instruction Abort from the current Exception level, where the current Exception level must be using AArch64.

  Used for MMU faults generated by instruction accesses and Synchronous external aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

The ISS encoding for these exceptions is:




**Bits [24:11]**

Reserved, RES0.

**FnV, bit [10]**

FAR not Valid, for a Synchronous external abort other than a Synchronous external abort on a translation table walk.

`0` FAR is valid.

`1` FAR is not valid, and holds an UNKNOWN value.

This field is only valid if the IFSC code is `010000`. It is RES0 for all other aborts.

**EA, bit [9]**

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of external aborts.

For any abort other than an External abort this bit returns a value of 0.

**Bit [8]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**S1PTW, bit [7]**

For a stage 2 fault, indicates whether the fault was a fault on the stage 2 translation of an address accessed during a stage 1 translation table walk:

`0` Fault not on a stage 2 translation for a stage 1 translation table walk.

`1` Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For a fault other than a stage 2 fault, this bit is RES0.

**Bit [6]**

Reserved, RES0.

**IFSC, bits [5:0]**

Instruction Fault Status Code. Possible values of this field are:

`000000` Address size fault, level 0 of translation or translation table base register

`000001` Address size fault, level 1

`000010` Address size fault, level 2

`000011` Address size fault, level 3

`000100` Translation fault, level 0

`000101` Translation fault, level 1

`000110` Translation fault, level 2

`000111` Translation fault, level 3

`001001` Access flag fault, level 1

`001010` Access flag fault, level 2

`001011` Access flag fault, level 3

`001101` Permission fault, level 1

`001110` Permission fault, level 2

`001111` Permission fault, level 3

`010000` Synchronous external abort, other than synchronous parity or ECC error, not on translation table walk

`011000` Synchronous parity or ECC error on memory access, not on translation table walk

`010100` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 0

`010101` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 1

`010110` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 2

`010111` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 3

`011100` Synchronous parity or ECC error on memory access on translation table walk, level 0

`011101` Synchronous parity or ECC error on memory access on translation table walk, level 1

`011110` Synchronous parity or ECC error on memory access on translation table walk, level 2

`011111` Synchronous parity or ECC error on memory access on translation table walk, level 3

`100001` Alignment fault

`110000` TLB conflict abort

All other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 0 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

**Note**

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

#### *ISS encoding for an exception from a Data abort*

This encoding is used by:

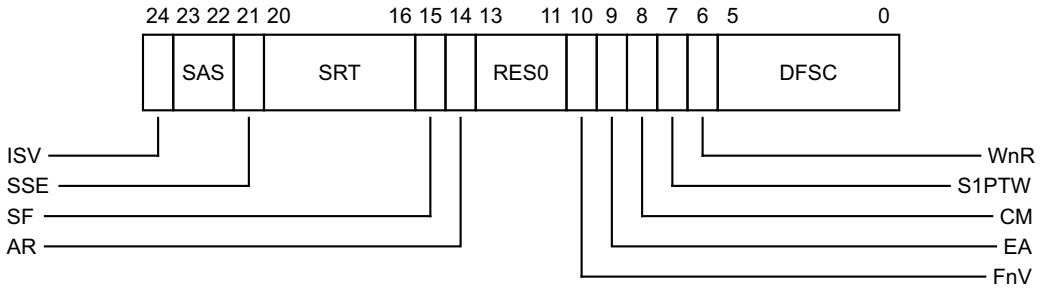
- Data Abort that caused entry from a lower Exception level, where that Exception level could be using AArch64 or using AArch32.

  Used for MMU faults generated by data accesses, alignment faults other than those caused by the Stack Pointer misalignment, and Synchronous external aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

- Data Abort that caused entry from a current Exception level, where the current Exception level must be using AArch64.

  Used for MMU faults generated by data accesses, alignment faults other than those caused by the Stack Pointer misalignment, and Synchronous external aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.

The ISS encoding for these exceptions is:




**ISV, bit [24]**

Instruction syndrome valid. Indicates whether the syndrome information in ISS[23:14] is valid.

`0` No valid instruction syndrome. ISS[23:14] are RES0.

`1` ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults reported in ESR_EL2 except the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with `0b11111`), including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- AArch32 instructions where the instruction:
  - Is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
  - Is not performing register writeback.
  - Is not using R15 as a source or destination register.

ISV is 0 for all faults reported in ESR_EL1 or ESR_EL3.

For ISS reporting, a stage 2 abort on a stage 1 translation table does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

The value of ISV on a Synchronous external abort on a stage 2 translation table walk is IMPLEMENTATION DEFINED.

**SAS, bits [23:22]**

Syndrome Access Size. When ISV is 1, indicates the size of the access attempted by the faulting operation.

`00` Byte

`01` Halfword

`10` Word

`11` Doubleword

This field is RES0 when the value of ISV is 0.

**SSE, bit [21]**

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

`0` Sign-extension not required.

`1` Data item must be sign-extended.

For all other operations this bit is 0.

This field is RES0 when the value of ISV is 0.

**SRT, bits [20:16]**

Syndrome Register transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction. If the exception was taken from an Exception level that is using AArch32 then this is the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states* on page D1-1620.

This field is RES0 when the value of ISV is 0.

**SF, bit [15]**

Width of the register accessed by the instruction is Sixty-Four. When ISV is 1, the possible values of this bit are:

`0` Instruction loads/stores a 32-bit wide register.

`1` Instruction loads/stores a 64-bit wide register.

**Note**

This field specifies the register width identified by the instruction, not the Execution state.

This field is RES0 when the value of ISV is 0.

**AR, bit [14]**

Acquire/Release. When ISV is 1, the possible values of this bit are:

`0` Instruction did not have acquire/release semantics.

`1` Instruction did have acquire/release semantics.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is RES0 when the value of ISV is 0.

**Bits [13:11]**

Reserved, RES0.

**FnV, bit [10]**

FAR not Valid, for a Synchronous external abort other than a Synchronous external abort on a translation table walk.

0 FAR is valid.

1 FAR is not valid, and holds an UNKNOWN value.

This field is only valid if the DFSC code is 010000. It is RES0 for all other aborts.

**EA, bit [9]**

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of external aborts.

For any abort other than an External abort this bit returns a value of 0.

**CM, bit [8]**

Indicates the data fault came from a Cache Maintenance Instruction, other than DC ZVA instruction, or from a synchronous fault on an address translation instruction that translates a VA to a PA.

**S1PTW, bit [7]**

For a stage 2 fault, indicates whether the fault was a fault on the stage 2 translation of an address accessed during a stage 1 translation table walk:

0 Fault not on a stage 2 translation for a stage 1 translation table walk.

1 Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For a fault other than a stage 2 fault, this bit is RES0.

**WnR, bit [6]**

Write not Read. Indicates whether a synchronous abort was caused by a write instruction or a read instruction. The possible values of this bit are:

0 Abort caused by a read instruction.

1 Abort caused by a write instruction.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For an asynchronous Data Abort exception this bit is UNKNOWN.

**DFSC, bits [5:0]**

Data Fault Status Code. Possible values of this field are:

000000 Address size fault, level 0 of translation or translation table base register

000001 Address size fault, level 1

000010 Address size fault, level 2

000011 Address size fault, level 3

000100 Translation fault, level 0

000101 Translation fault, level 1

000110 Translation fault, level 2

000111 Translation fault, level 3

001001 Access flag fault, level 1

001010 Access flag fault, level 2

001011 Access flag fault, level 3


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| | |
|---|---|
| 001101 | Permission fault, level 1 |
| 001110 | Permission fault, level 2 |
| 001111 | Permission fault, level 3 |
| 010000 | Synchronous external abort, other than synchronous parity or ECC error, not on translation table walk |
| 011000 | Synchronous parity or ECC error on memory access, not on translation table walk |
| 010100 | Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 0 |
| 010101 | Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 1 |
| 010110 | Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 2 |
| 010111 | Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 3 |
| 011100 | Synchronous parity or ECC error on memory access on translation table walk, level 0 |
| 011101 | Synchronous parity or ECC error on memory access on translation table walk, level 1 |
| 011110 | Synchronous parity or ECC error on memory access on translation table walk, level 2 |
| 011111 | Synchronous parity or ECC error on memory access on translation table walk, level 3 |
| 100001 | Alignment fault |
| 110000 | TLB conflict abort |
| 110100 | IMPLEMENTATION DEFINED fault (Lockdown fault) |
| 110101 | IMPLEMENTATION DEFINED fault (Unsupported Exclusive access fault) |
| 111101 | Section Domain Fault, used only for faults reported in the PAR_EL1 |
| 111110 | Page Domain Fault, used only for faults reported in the PAR_EL1 |

All other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 0 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

**Note**

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

#### *ISS encoding for an exception from a trapped Floating-point exception*

This encoding is used by:

- Exceptions as a result of Floating-point exception from AArch32.

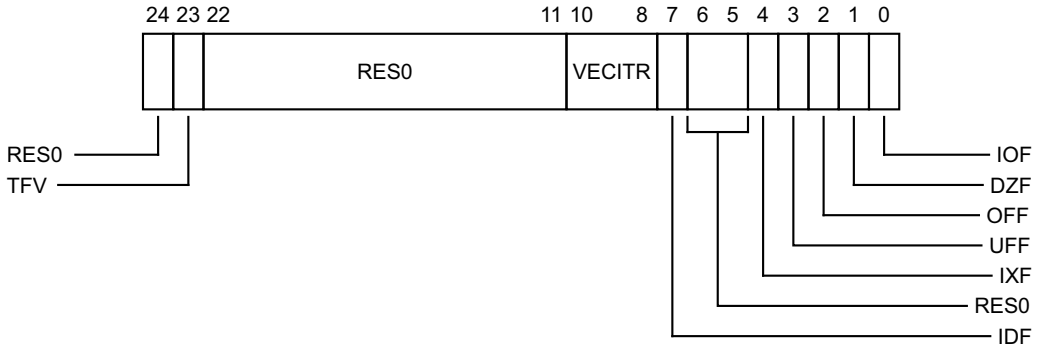
  Whether this Exception class is supported is IMPLEMENTATION DEFINED.

- Exceptions as a result of Floating-point exception from AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Whether this Exception class is supported is IMPLEMENTATION DEFINED.

The ISS encoding for these exceptions is:




**Bit [24]**

Reserved, RES0.

**TFV, bit [23]**

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about a trapped fault. The possible values of this bit are:

`0` The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about a trapped fault and are UNKNOWN.

`1` The IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about one or more trapped faults. All floating-point exceptions indicated by these bits have occurred since the bits were last cleared to 0.

**Bits [22:11]**

Reserved, RES0.

**VECITR, bits [10:8]**

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES0.

For a trapped floating-point exception from a scalar floating point instruction executed in AArch64 state this field is RES0.

For a trapped floating-point exception from a vector floating point instruction executed in AArch64 state this field holds the number of the element that is being reported in ESR_ELx. That is, it holds the number of the element that generated the reported floating-point exception.

For an instruction that combines two adjacent elements to creating a single element, VECITR reports the lower-numbered input element. This can occur with the pairwise instructions, for example the FADDP instruction.

For an instruction that produces a single value from all the elements in the input vector, the only possible exceptions result from the input operands, and are an Invalid Operation exception on a Signaling NaN, or an Input Denormal exception. In these cases, VECITR reports the element number that is being reported in the ESR. This can occur with the across-lanes instructions, for example FMAXV.

VECITR holds an element number as an unsigned binary number.

**IDF, bit [7]**

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

`0` Input denormal floating-point exception has not occurred.

`1` Input denormal floating-point exception has occurred since the bit was last set to 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [6:5]**

Reserved, RES0.

**IXF, bit [4]**

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0 Inexact floating-point exception has not occurred.

1 Inexact floating-point exception has occurred since the bit was last set to 0.

**UFF, bit [3]**

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0 Underflow floating-point exception has not occurred.

1 Underflow floating-point exception has occurred since the bit was last set to 0.

**OFF, bit [2]**

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0 Overflow floating-point exception has not occurred.

1 Overflow floating-point exception has occurred since the bit was last set to 0.

**DZF, bit [1]**

Divide-by-zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0 Divide-by-zero floating-point exception has not occurred.

1 Divide-by-zero floating-point exception has occurred since the bit was last set to 0.

**IOF, bit [0]**

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0 Invalid Operation floating-point exception has not occurred.

1 Invalid Operation floating-point exception has occurred since the bit was last set to 0.

In an implementation where the SIMD and floating-point implementation supports the trapping of floating-point exceptions:

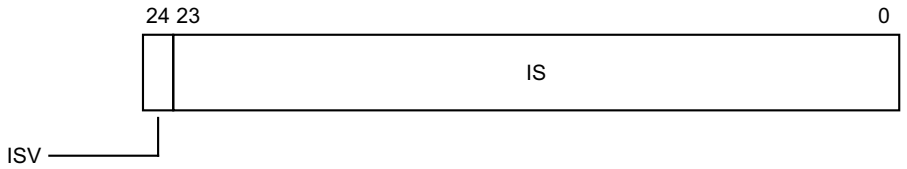
- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

### *ISS encoding for an SError interrupt*

This encoding is used by SError Interrupt.

The ISS encoding for these exceptions is:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**ISV, bit [24]**

Instruction syndrome valid. Indicates whether the rest of the syndrome information in this register is valid.

`0` No valid instruction syndrome. ISS[23:0] are RES0.

`1` ISS[23:0] hold a valid instruction syndrome.

**IS, bits [23:0]**

IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError interrupt. Only valid if bit[24] of this register is 1. If bit[24] is 0, this field is RES0.

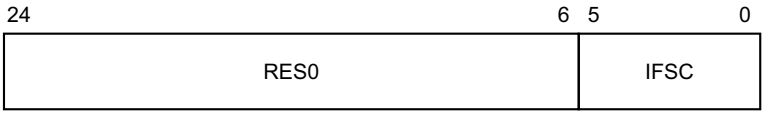
### *ISS encoding for an exception from a Breakpoint or Vector Catch debug event*

This encoding is used by:

- Breakpoint debug event that caused entry from a lower Exception level.
- Breakpoint debug event from the current Exception level.
- AArch32 state Vector catch debug event.

  The only case where an exception from a Vector catch debug event is taken to an Exception level that is using AArch64 is when the Vector catch is routed to EL2 and EL2 is using AArch64.

The ISS encoding for these exceptions is:




**Bits [24:6]**

Reserved, RES0.

**IFSC, bits [5:0]**

Instruction Fault Status Code. This field is set to `0b100010`, to indicate a Debug event.

For more information about generating these exceptions:

- For exceptions from AArch64, see *Breakpoint exceptions* on page D2-1653.
- For exceptions from AArch32, see *Breakpoint exceptions* on page G2-3898 and *Vector Catch exceptions* on page G2-3937.

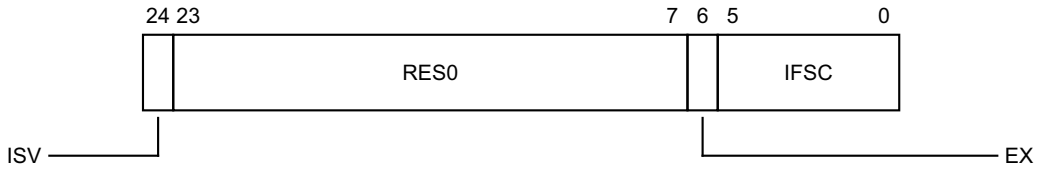
### *ISS encoding for an exception from a Software Step debug event*

This encoding is used by:

- Software step debug event that caused entry from a lower Exception level.
- Software step debug event from the current Exception level.

The ISS encoding for these exceptions is:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**ISV, bit [24]**

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

`0` EX bit is RES0.

`1` EX bit is valid.

See the EX bit description for more information.

**Bits [23:7]**

Reserved, RES0.

**EX, bit [6]**

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

`0` An instruction other than a Load-Exclusive instruction was stepped.

`1` A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

**IFSC, bits [5:0]**

Instruction Fault Status Code. This field is set to `0b100010`, to indicate a Debug event.

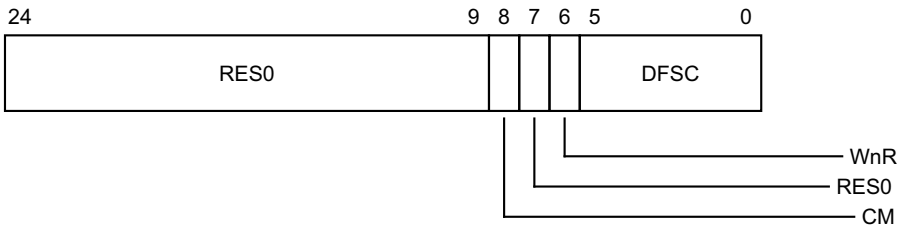
For more information about generating these exceptions, see *Software Step exceptions* on page D2-1686.

### *ISS encoding for an exception from a Watchpoint debug event*

This encoding is used by:

- Watchpoint debug event that caused entry from a lower Exception level.
- Watchpoint debug event from the current Exception level.

The ISS encoding for these exceptions is:




**Bits [24:9]**

Reserved, RES0.

**CM, bit [8]**

Indicates the data fault came from a Cache Maintenance Instruction, other than DC ZVA instruction, or from a synchronous fault on an address translation instruction that translates a VA to a PA.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bit [7]**

Reserved, RES0.

**WnR, bit [6]**

Write not Read. Indicates whether the abort was caused by a write instruction or a read instruction. The possible values of this bit are:

`0` Abort caused by a read instruction.

`1` Abort caused by a write instruction.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

**DFSC, bits [5:0]**

Data Fault Status Code. This field is set to `0b100010`, to indicate a Debug event.

For more information about generating these exceptions, see *Watchpoint exceptions* on page D2-1671.

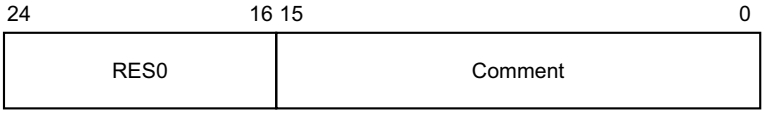
### *ISS encoding for an exception from execution of a Software Breakpoint instruction*

This encoding is used by:

- BKPT instruction executed in AArch32 state.
- BRK instruction executed in AArch64 state.

  This is reported in ESR_EL3 only if a BRK instruction is executed at EL3.

The ISS encoding for these exceptions is:




**Bits [24:16]**

Reserved, RES0.

**Comment, bits [15:0]**

Set to the instruction comment field value, zero extended as necessary. For the AArch32 BKPT instructions, the comment field is described as the immediate field.

For more information about generating these exceptions, see *Software Breakpoint Instruction exceptions* on page D2-1650.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.28 FAR_EL1, Fault Address Register (EL1)

The FAR_EL1 characteristics are:

**Purpose**

Holds the faulting Virtual Address for all synchronous instruction or data aborts, or exceptions from a misaligned PC or a Watchpoint debug event, taken to EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

Execution at EL0 makes FAR_EL1 become UNKNOWN.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register FAR_EL1[31:0] is architecturally mapped to AArch32 System register DFAR (NS).

AArch64 System register FAR_EL1[63:32] is architecturally mapped to AArch32 System register IFAR (NS).

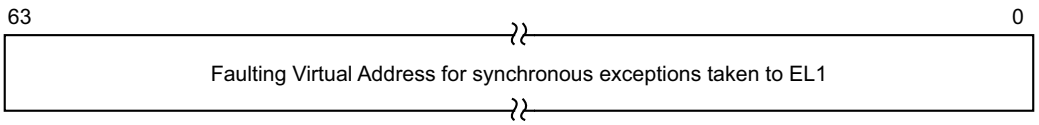
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

FAR_EL1 is a 64-bit register.

### Field descriptions

The FAR_EL1 bit assignments are:




**Bits [63:0]**

Faulting Virtual Address for synchronous exceptions taken to EL1. Exceptions that set the FAR_EL1 are instruction aborts (EC 0x20 or 0x21), data aborts (EC 0x24 or 0x25), a misaligned PC exception (EC 0x22), or a Watchpoint exception (EC 0x34 or 0x35). ESR_EL1.EC holds the EC value for the exception.

For a synchronous external abort other than a synchronous external abort on a translation table walk, this field is valid only if ESR_EL1.FnV is 0, and the FAR_EL1 is UNKNOWN if ESR_EL1.FnV is 1.

For all other exceptions taken to EL1, the FAR_EL1 is UNKNOWN.

If a memory fault that sets FAR_EL1 is generated from a data cache maintenance or DC ZVA instruction, this field holds the address specified in the register argument of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the exception that updates FAR_EL1 is taken from an Exception level that is using AArch32, the top 32 bits are all zero, unless the faulting address is generated by a load or store instruction that sequentially increments from address `0xFFFFFFFF`. This is an UNPREDICTABLE condition, and in this case the upper 32-bits are set to `0x00000001`.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see *Address tagging in AArch64 state* on page D4-1742.

**Note**

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL1 is made UNKNOWN on an exception return from EL1.

### Accessing the FAR_EL1:

To access the FAR_EL1:

```
MRS <Xt>, FAR_EL1 ; Read FAR_EL1 into Xt
MSR FAR_EL1, <Xt> ; Write Xt to FAR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0110 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.29 FAR_EL2, Fault Address Register (EL2)

The FAR_EL2 characteristics are:

**Purpose**

Holds the faulting Virtual Address for all synchronous instruction or data aborts, or exceptions from a misaligned PC or a Watchpoint debug event, taken to EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

Execution at EL1 or EL0 makes FAR_EL2 become UNKNOWN.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register FAR_EL2[31:0] is architecturally mapped to AArch32 System register HDFAR.

AArch64 System register FAR_EL2[63:32] is architecturally mapped to AArch32 System register HIFAR.

AArch64 System register FAR_EL2[31:0] is architecturally mapped to AArch32 System register DFAR (S) when EL2 is implemented.

AArch64 System register FAR_EL2[63:32] is architecturally mapped to AArch32 System register IFAR (S) when EL2 is implemented.

If EL2 is not implemented, this register is RES0 from EL3.

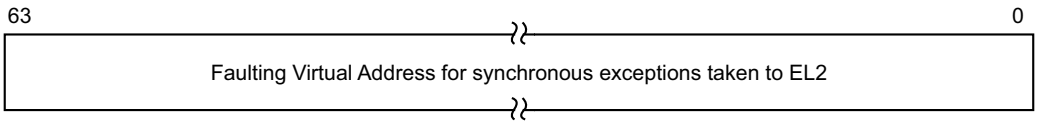
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

FAR_EL2 is a 64-bit register.

#### Field descriptions

The FAR_EL2 bit assignments are:




**Bits [63:0]**

Faulting Virtual Address for synchronous exceptions taken to EL2. Exceptions that set the FAR_EL2 are instruction aborts (EC 0x20 or 0x21), data aborts (EC 0x24 or 0x25), a misaligned PC exception (EC 0x22), or a Watchpoint exception (EC 0x34 or 0x35). ESR_EL2.EC holds the EC value for the exception.

For a synchronous external abort other than a synchronous external abort on a translation table walk, this field is valid only if ESR_EL2.FnV is 0, and the FAR_EL2 is UNKNOWN if ESR_EL2.FnV is 1.

For all other exceptions taken to EL2, the FAR_EL2 is UNKNOWN.

If a memory fault that sets FAR_EL2 is generated from a data cache maintenance or DC ZVA instruction, this field holds the address specified in the register argument of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the exception that updates FAR_EL2 is taken from an Exception level that is using AArch32, the top 32-bits are all zero, unless the faulting address is generated by a load or store instruction that sequentially increments from address 0xFFFFFFFF. This is an UNPREDICTABLE condition, and in this case the upper 32-bits are set to 0x00000001.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see *Address tagging in AArch64 state* on page D4-1742.

**Note**

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL2 is made UNKNOWN on an exception return from EL2.

### Accessing the FAR_EL2:

To access the FAR_EL2:

```
MRS <Xt>, FAR_EL2 ; Read FAR_EL2 into Xt
MSR FAR_EL2, <Xt> ; Write Xt to FAR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0110 | 0000 | 000 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### D7.2.30 FAR_EL3, Fault Address Register (EL3)

The FAR_EL3 characteristics are:

**Purpose**

Holds the faulting Virtual Address for all synchronous instruction or data aborts, or exceptions from a misaligned PC, taken to EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

Execution at EL2, EL1 or EL0 makes FAR_EL3 become UNKNOWN.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

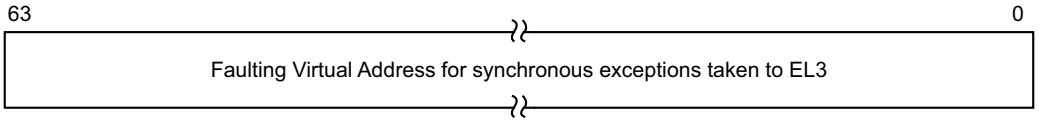
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

FAR_EL3 is a 64-bit register.

#### Field descriptions

The FAR_EL3 bit assignments are:




**Bits [63:0]**

Faulting Virtual Address for synchronous exceptions taken to EL3. Exceptions that set the FAR_EL3 are instruction aborts (EC `0x20` or `0x21`), data aborts (EC `0x24` or `0x25`), a misaligned PC exception (EC `0x22`). ESR_EL3.EC holds the EC value for the exception.

For a synchronous external abort other than a synchronous external abort on a translation table walk, this field is valid only if ESR_EL3.FnV is 0, and the FAR_EL3 is UNKNOWN if ESR_EL3.FnV is 1.

For all other exceptions taken to EL3, the FAR_EL3 is UNKNOWN.

If a memory fault that sets FAR_EL3 is generated from a data cache maintenance or DC ZVA instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL3 is taken from an EL using AArch32, the top 32-bits are all zero, unless the faulting address is generated by a load or store instruction that sequentially increments from address `0xffffffff`. This is an UNPREDICTABLE condition, and in this case the upper 32-bits are set to `0x00000001`.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see *Address tagging in AArch64 state* on page D4-1742.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lowest address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL3 is made UNKNOWN on an exception return from EL3.

### Accessing the FAR_EL3:

To access the FAR_EL3:

```
MRS <Xt>, FAR_EL3 ; Read FAR_EL3 into Xt
MSR FAR_EL3, <Xt> ; Write Xt to FAR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0110 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.31 FPEXC32_EL2, Floating-Point Exception Control register

The FPEXC32_EL2 characteristics are:

**Purpose**

Allows access to the AArch32 register FPEXC from AArch64 state only. Its value has no effect on execution in AArch64 state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CPTR_EL2.TFP==1, Non-secure EL2 accesses to this register are trapped to EL2.

If CPTR_EL3.TFP==1, EL2 and EL3 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register FPEXC32_EL2 is architecturally mapped to AArch32 System register FPEXC.

If EL1 cannot use AArch32, this register is UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

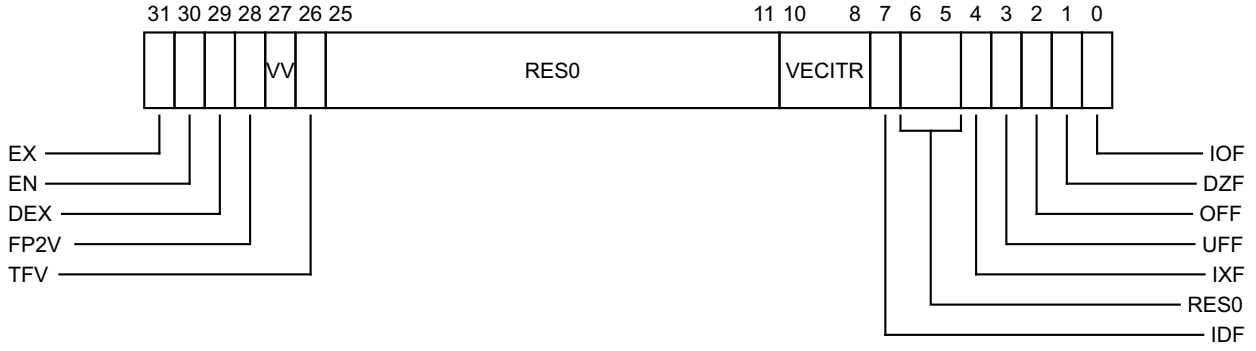
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

FPEXC32_EL2 is a 32-bit register.

### Field descriptions

The FPEXC32_EL2 bit assignments are:




**EX, bit [31]**

Exception bit. In ARMv8, this bit is RAZ/WI.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EN, bit [30]**

Enables access to the Advanced SIMD and floating-point functionality from all Exception levels, except that setting this field to 0 does not disable the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

`0` Accesses to the FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers, are UNDEFINED at all Exception levels.

`1` This control permits access to the Advanced SIMD and floating-point functionality at all Exception levels.

Execution of floating-point and Advanced SIMD instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.
- If executing in Non-secure state:
  - HCPTR.TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
  - CPACR.ASEDIS.
  - If executing in Non-secure state, HCPTR.TASE and NSACR.NSTRCDIS.

See the descriptions of the controls for more information.

**Note**

When executing at EL0 using AArch32 with EL1 using AArch64, the PE behaves as if the value of FPEXC.EN bit is 1.

**DEX, bit [29]**

Defined synchronous floating-point exception bit.

This bit indicates the cause of an Undefined Instruction exception from the attempted execution of a floating-point instruction, and whether the FPEXC.TFV bit is valid.

The meaning of this bit is:

`0` The exception occurred because of the execution of an unallocated instruction in the CP10 or CP11 encoding space. If the FPEXC.TFV bit is a RW bit then it is invalid and UNKNOWN. If the FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} bits are RW bits then they are invalid and UNKNOWN.

`1` The FPEXC.TFV bit is valid and indicates the cause of the exception.

On an exception caused by a trapped floating-point exception, the exception-handling routine must clear this bit to 0.

On an implementation that both does not support trapping of floating-point exceptions and implements the FPSCR.{Stride, Len} fields as RAZ, this bit is RES0.

**FP2V, bit [28]**

FPINST2 instruction valid bit. In ARMv8, this bit is RES0.

**VV, bit [27]**

VECITR valid bit. In ARMv8, this bit is RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TFV, bit [26]**

Trapped Fault Valid bit. Valid only when the value of FPEXC.DEX is 1. When valid, it indicates the cause of the exception and therefore whether the FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} bits are valid.

0 The exception was caused by the execution of a floating-point VABS, VADD, VDIV, VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VMOV, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VSQRT, or VSUB instruction when one or both of FPSCR.{Stride, Len} was non-zero. If the FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} bits are RW then they are invalid and UNKNOWN.

1 FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} indicate the presence of trapped floating-point exceptions that had occurred at the time of the exception. Bits are set for all trapped exceptions that had occurred at the time of the exception.

This bit returns a status value and ignores writes.

When the value of FPEXC.DEX is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

On an implementation that supports the trapping of floating-point exceptions and implements FPSCR.{Stride, Len} as RAZ, this bit is RAO/WI.

**Bits [25:11]**

Reserved, RES0.

**VECITR, bits [10:8]**

Vector iteration count. In ARMv8, this field is RES1.

**IDF, bit [7]**

Input Denormal trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Input Denormal exception occurred while FPSCR.IDE was 1:

0 Input denormal exception has not occurred.

1 Input denormal exception has occurred.

Input Denormal exceptions can occur only when FPSCR.FZ is 1.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

**Bits [6:5]**

Reserved, RES0.

**IXF, bit [4]**

Inexact trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Inexact exception occurred while FPSCR.IXE was 1:

0 Inexact exception has not occurred.

1 Inexact exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**UFF, bit [3]**

Underflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Underflow exception occurred while FPSCR.UFE was 1:

0 Underflow exception has not occurred.

1 Underflow exception has occurred.

Underflow trapped exceptions can occur only when FPSCR.FZ is 0.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

**OFF, bit [2]**

Overflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Overflow exception occurred while FPSCR.OFE was 1:

0 Overflow exception has not occurred.

1 Overflow exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

**DZF, bit [1]**

Divide-by-zero trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether a Divide-by-zero exception occurred while FPSCR.DZE was 1:

0 Divide-by-zero exception has not occurred.

1 Divide-by-zero exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

**IOF, bit [0]**

Invalid Operation trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Invalid Operation exception occurred while FPSCR.IOE was 1:

0 Invalid Operation exception has not occurred.

1 Invalid Operation exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

## Accessing the FPEXC32_EL2:

To access the FPEXC32_EL2:

```
MRS <Xt>, FPEXC32_EL2 ; Read FPEXC32_EL2 into Xt
MSR FPEXC32_EL2, <Xt> ; Write Xt to FPEXC32_EL2
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0101 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.32 HACR_EL2, Hypervisor Auxiliary Control Register

The HACR_EL2 characteristics are:

**Purpose**

Controls trapping to EL2 of IMPLEMENTATION DEFINED aspects of Non-secure EL1 or EL0 operation.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register HACR_EL2 is architecturally mapped to AArch32 System register HACR.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HACR_EL2 is a 32-bit register.

#### Field descriptions

The HACR_EL2 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the HACR_EL2:

To access the HACR_EL2:

```
MRS <Xt>, HACR_EL2 ; Read HACR_EL2 into Xt
MSR HACR_EL2, <Xt> ; Write Xt to HACR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0001 | 0001 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.33 HCR_EL2, Hypervisor Configuration Register

The HCR_EL2 characteristics are:

**Purpose**

Provides configuration controls for virtualization, including defining whether various Non-secure operations are trapped to EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register HCR_EL2[31:0] is architecturally mapped to AArch32 System register HCR.

AArch64 System register HCR_EL2[63:32] is architecturally mapped to AArch32 System register HCR2.

If EL2 is not implemented, this register is RES0 from EL3.

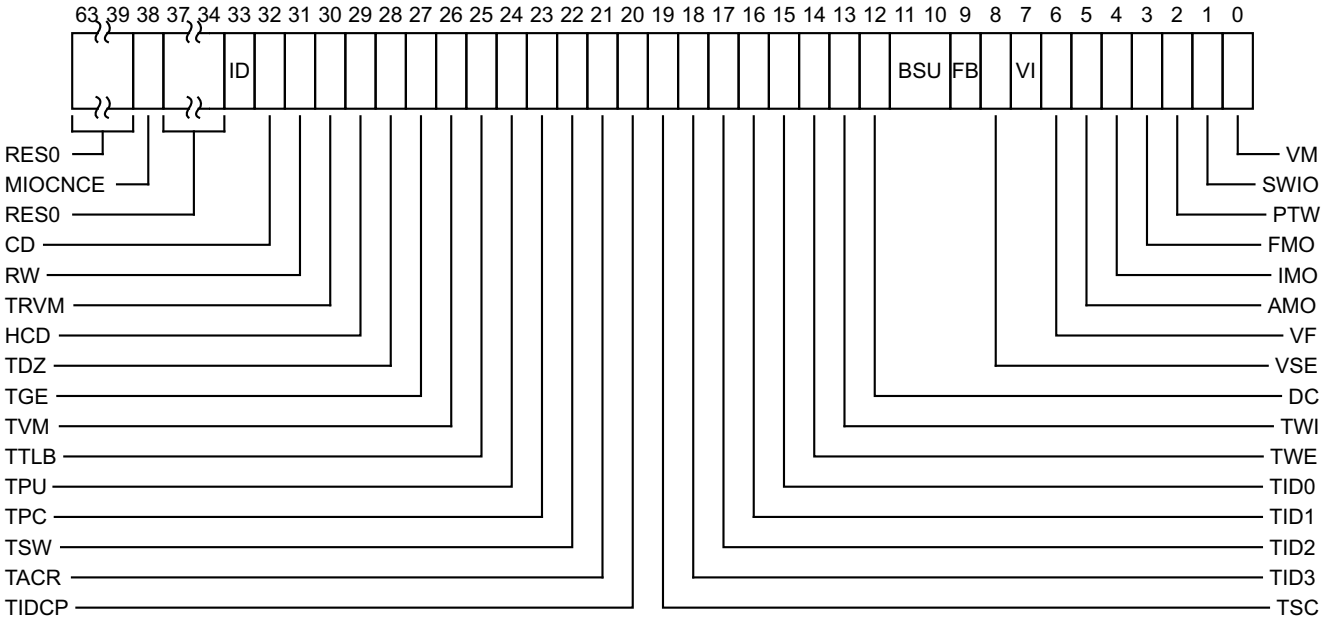
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HCR_EL2 is a 64-bit register.

#### Field descriptions

The HCR_EL2 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [63:39]**

Reserved, RES0.

**MIOCNCE, bit [38]**

Mismatched Inner/Outer Cacheable Non-Coherency Enable, for the Non-secure EL1&0 translation regime.

0 For the Non-secure EL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there must be no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

1 For the Non-secure EL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there might be a loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

For more information see *Mismatched memory attributes* on page B2-104.

This field can be implemented as RAZ/WI.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**Bits [37:34]**

Reserved, RES0.

**ID, bit [33]**

Stage 2 Instruction cache disable. For the Non-secure EL1&0 translation regime, when HCR_EL2.VM==1, this control forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

0 This control has no effect on stage 2 of the Non-secure EL1&0 translation regime.

1 For the Non-secure EL1&0 translation regime, forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

This bit has no effect on the EL2 or EL3 translation regimes.

**CD, bit [32]**

Stage 2 Data cache disable. For the Non-secure EL1&0 translation regime, when HCR_EL2.VM==1, this control forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

0 This control has no effect on stage 2 of the Non-secure EL1&0 translation regime for data accesses and translation table walks.

1 For the Non-secure EL1&0 translation regime, forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

This bit has no effect on the EL2 or EL3 translation regimes.

**RW, bit [31]**

Execution state control for lower Exception levels:

0 Lower levels are all AArch32.

1 The Execution state for EL1 is AArch64. The Execution state for EL0 is determined by the current value of PSTATE.nRW when executing at EL0.

If all lower Exception levels cannot use AArch32 then this bit is RAO/WI.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In an implementation that includes EL3, when SCR_EL3.NS==0, the PE behaves as if this bit has the same value as the SCR_EL3.RW bit for all purposes other than returning the result of a direct read of HCR_EL2.

The RW bit is permitted to be cached in a TLB.

**TRVM, bit [30]**

Trap Reads of Virtual Memory controls. Traps Non-secure EL1 reads of the virtual memory control registers to EL2, from both Execution states. The registers for which read accesses are trapped are as follows:

Non-secure EL1 using AArch64: SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, CONTEXTIDR_EL1.

Non-secure EL1 using AArch32: SCTLR, TTBR0, TTBR1, TTBCR, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

`0` This control has no effect on Non-secure EL1 read accesses to Virtual Memory controls.

`1` Non-secure EL1 read accesses to the specified Virtual Memory controls are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**HCD, bit [29]**

HVC instruction disable. Disables Non-secure state execution of HVC instructions, from both Execution states.

`0` HVC instruction execution is enabled at EL2 and Non-secure EL1.

`1` HVC instructions are UNDEFINED at EL2 and Non-secure EL1. The Undefined Instruction exception is taken to the Exception level at which the HVC instruction is executed.

**Note**

HVC instructions are always UNDEFINED at EL0.

This bit is only implemented if EL3 is not implemented. Otherwise, it is RES0.

**TDZ, bit [28]**

Trap DC ZVA instructions. Traps Non-secure EL0 and EL1 execution of DC ZVA instructions to EL2, from AArch64 state only.

`0` This control has no effect on the Non-secure EL0 and EL1 execution of DC ZVA instructions.

`1` Any attempt to execute a DC ZVA instruction at Non-secure EL0 using AArch64 or Non-secure EL1 using AArch64 is trapped to EL2.

Reading the DCZID_EL0 returns a value that indicates that DC ZVA instructions are not supported.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TGE, bit [27]**

Trap General Exceptions, from Non-secure EL0.

`0` This control has no effect on execution at EL0.

`1` When the value of SCR_EL3.NS is 0, this control has no effect on execution at EL0.

When the value of SCR_EL3.NS is 1, then:

- All exceptions that would be routed to EL1 are routed to EL2.
- The SCTLR_EL1.M field, or the SCTLR.M field if EL1 is using AArch32, is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR_EL1 or SCTLR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The HCR_EL2.{FMO, IMO, AMO} fields are treated as being 1 for all purposes other than returning the result of a direct read of HCR_EL2.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.
- An exception return to EL1 is treated as an illegal exception return.
- The MDCR_EL2.{TDRA,TDOSA,TDA} fields are treated as being 1 for all purposes other than returning the result of a direct read of MDCR_EL2.

HCR_EL2.TGE must not be cached in a TLB.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TVM, bit [26]**

Trap Virtual Memory controls. Traps Non-secure EL1 writes to the virtual memory control registers to EL2, from both Execution states. The registers for which write accesses are trapped are as follows:

Non-secure EL1 using AArch64: SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, CONTEXTIDR_EL1.

Non-secure EL1 using AArch32: SCTLR, TTBR0, TTBR1, TTBCR, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

`0` This control has no effect on Non-secure EL1 write accesses to EL1 virtual memory control registers.

`1` Non-secure EL1 write accesses to the specified EL1 virtual memory control registers are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TTLB, bit [25]**

Trap TLB maintenance instructions. Traps Non-secure EL1 execution of TLB maintenance instructions to EL2, from both Execution states. This applies to the following instructions:

Non-secure EL1 using AArch64: TLBI VMALLE1IS, TLBI VAE1IS, TLBI ASIDE1IS, TLBI VAAE1IS, TLBI VALE1IS, TLBI VAALE1IS, TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, TLBI VAALE1.

Non-secure EL1 using AArch32: TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS, ITLBIALL, ITLBIMVA, ITLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL

`0` This control has no effect on Non-secure EL1 execution of TLB maintenance instructions.

`1` Non-secure EL1 execution of the specific TLB maintenance instructions are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TPU, bit [24]**

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions at Non-secure EL1 or EL0 using AArch64, and at Non-secure EL1 using AArch32, to EL2. This applies to the following instructions:

Non-secure EL0 using AArch64: IC IVAU, DC CVAU. However, if the value of SCTLR_EL1.UCI is 0 these instructions are UNDEFINED at EL0 and the Undefined Instruction exception is higher priority than this trap to EL2.

Non-secure EL1 using AArch64: IC IVAU, IC IALLU, IC IALLUIS, DC CVAU.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Non-secure EL1 using AArch32: ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAU.

**Note**

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and:

- IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.
- ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always UNDEFINED at EL0 using AArch32.

`0` This control has no effect on the execution of cache maintenance instructions.

`1` Non-secure execution of the specified instructions is trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TPC, bit [23]**

Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps execution of those cache maintenance instructions at Non-secure EL1 or EL0 using AArch64, and at Non-secure EL1 using AArch32, to EL2. This applies to the following instructions:

Non-secure EL0 using AArch64: DC CIVAC, DC CVAC. However, if the value of SCTLR_EL1.UCI is 0 these instructions are UNDEFINED at EL0 and the Undefined Instruction exception is higher priority than this trap to EL2.

Non-secure EL1 using AArch64: DC IVAC, DC CIVAC, DC CVAC.

Non-secure EL1 using AArch32: DCIMVAC, DCCIMVAC, DCCMVAC.

**Note**

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and:

- DC IVAC is always UNDEFINED at EL0 using AArch64.
- DCIMVAC, DCCIMVAC, and DCCMVAC are always UNDEFINED at EL0 using AArch32.

`0` This control has no effect on the execution of cache maintenance instructions.

`1` Non-secure execution of the specified instructions is trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TSW, bit [22]**

Trap data or unified cache maintenance instructions that operate by Set/Way. Traps execution of those cache maintenance instructions at Non-secure EL1 using AArch64, and at Non-secure EL1 using AArch32, to EL2. This applies to the following instructions:

Non-secure EL1 using AArch64: DC ISW, DC CSW, DC CISW.

Non-secure EL1 using AArch32: DCISW, DCCSW, DCCISW.

**Note**

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

`0` This control has no effect on the execution of cache maintenance instructions.

`1` Non-secure execution of the specified instructions is trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TACR, bit [21]**

Trap Auxiliary Control Registers. Traps Non-secure EL1 accesses to the Auxiliary Control Registers to EL2, from both Execution states. This applies to the following register accesses:

- Non-secure EL1 using AArch64: ACTLR_EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Non-secure EL1 using AArch32: ACTLR and, if implemented, ACTLR2.

0 This control has no effect on Non-secure EL1 accesses to the Auxiliary Control Registers.

1 Non-secure EL1 accesses to the specified registers are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TIDCP, bit [20]**

Trap IMPLEMENTATION DEFINED functionality. Traps Non-secure EL1 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL2. This applies to the following register accesses:

AArch64: The following reserved encoding spaces:

- IMPLEMENTATION DEFINED system operations, which are accessed using SYS and SYSL, with CRm == {13, 15}.
- IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name.

AArch32: MCR and MRC instructions as follows:

- All CP15, CRn==9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}.
- All CP15, CRn==10, opc1 =={0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0-7}.
- All CP15, CRn==11, opc1=={0-7}, CRm == {c0-c8, c15}, opc2 == {0-7}.

When HCR_EL2.TIDCP is set to 1, it is IMPLEMENTATION DEFINED whether any of this functionality accessed from Non-secure EL0 is trapped to EL2. If it is not, it is UNDEFINED, and the PE takes an Undefined Instruction exception to EL1.

0 This control has no effect on the encodings reserved for IMPLEMENTATION DEFINED functionality.

1 Non-secure EL1 accesses to or execution of the specified encodings reserved for IMPLEMENTATION DEFINED functionality are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TSC, bit [19]**

Trap SMC instructions. Traps Non-secure EL1 execution of SMC instructions to EL2, from both Execution states.

0 This control has no effect on the execution of SMC instructions.

1 Any attempt to execute an SMC instruction at Non-secure EL1 using AArch64 or Non-secure EL1 using AArch32 is trapped to EL2, regardless of the value of SCR_EL3.SMD.

In AArch32 state, the ARMv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

If EL3 is not implemented, this bit is RES0.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TID3, bit [18]**

Trap ID group 3. Traps Non-secure EL1 reads of the following registers to EL2:

AArch64: ID_PFR0_EL1, ID_PFR1_EL1, ID_DFR0_EL1, ID_AFR0_EL1, ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, ID_ISAR5_EL1, MVFR0_EL1, MVFR1_EL1, MVFR2_EL1, ID_AA64PFR0_EL1, ID_AA64PFR1_EL1, ID_AA64DFR0_EL1, ID_AA64DFR1_EL1, ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1, ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, ID_AA64AFR0_EL1, ID_AA64AFR1_EL1, and ID_MMFR4_EL1,


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

except that if ID_MMFR4_EL1 is implemented as RAZ/WI then it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4_EL1 are trapped. It is IMPLEMENTATION DEFINED whether this field traps MRS accesses to encodings in the following range that are not already mentioned in this field description:

- Op0 == 3, op1 == 0, CRn == c0, CRm == {c2-c7}, op2 == {0-7}.

AArch32: ID_PFR0, ID_PFR1, ID_DFR0, ID_AFR0, ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5, MVFR0, MVFR1, MVFR2, and ID_MMFR4, except that if ID_MMFR4 is implemented as RAZ/WI then it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4 are trapped. MRC access to any of the following CP15 encodings are also trapped:

- opc1 == 0, CRn == c0, CRm == {c3-c7}, opc2 == {0,1}.
- opc1 == 0, CRn == c0, CRm == c3, opc2 == 2.
- opc1 == 0, CRn == c0, CRm == c5, opc2 == {4,5}.

AArch32: It is IMPLEMENTATION DEFINED whether this bit traps MRC accesses to encodings in the following range that are not already mentioned in this field description:

- CP15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}.

`0` This control has no effect on Non-secure EL1 reads of the ID group 3 registers.

`1` The specified Non-secure EL1 read accesses to ID group 3 registers are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TID2, bit [17]**

Trap ID group 2. Traps the following register accesses to EL2:

AArch64:

- Non-secure EL1 reads of CTR_EL0, CCSIDR_EL1, CLIDR_EL1, and CSSELR_EL1.
- Non-secure EL0 reads of CTR_EL0, except that if the value of SCTLR_EL1.UCT is 0 then EL0 reads of CTR_EL0 are UNDEFINED and the Undefined Instruction exception takes precedence over this trap.
- Non-secure EL1 writes to CSSELR_EL1.

AArch32:

- Non-secure EL1 reads of the CTR, CCSIDR, CLIDR, and CSSELR.
- Non-secure EL1 writes to the CSSELR.

`0` This control has no effect on Non-secure EL1 and EL0 accesses to the ID group 2 registers.

`1` The specified Non-secure EL1 and EL0 accesses to ID group 2 registers are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TID1, bit [16]**

Trap ID group 1. Traps Non-secure EL1 reads of the following registers are trapped to EL2:

AArch64: REVIDR_EL1, AIDR_EL1.

AArch32: TCMTR, TLBTR, REVIDR, AIDR.

`0` This control has no effect on Non-secure EL1 reads of the ID group 1 registers.

`1` The specified Non-secure EL1 read accesses to ID group 1 registers are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TID0, bit [15]**

Trap ID group 0. Traps the following register accesses to EL2:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

AArch64: None.

AArch32:

- Non-secure EL1 reads of the JIDR.
- If the JIDR is RAZ from Non-secure EL0, Non-secure EL0 of the JIDR.
- Non-secure EL1 reads of the FPSID.

**Note**

- It is IMPLEMENTATION DEFINED whether the JIDR is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0 then the Undefined Instruction exception takes precedence over this trap.
- The FPSID is not accessible at EL0 using AArch32.
- When the FPSID is accessible, a T32 or A32 VMSR FPSID, <Rt> instruction is permitted but is ignored. The execution of this instruction is not trapped by this trap.

`0` This control has no effect on Non-secure EL1 reads of the ID group 0 registers.

`1` The specified Non-secure EL1 read accesses to ID group 0 registers are trapped to EL2.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**TWE, bit [14]**

Traps Non-secure EL0 and EL1 execution of WFE instructions to EL2, from both Execution states.

`0` This control has no effect on the execution of WFE instructions at Non-secure EL1 or Non-secure EL0.

`1` Any attempt to execute a WFE instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state, except that when the value of SCTLR_EL1.nTWE is 0, the trap of EL0 execution to EL1 takes precedence over this trap.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**Note**

Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

**TWI, bit [13]**

Traps Non-secure EL0 and EL1 execution of WFI instructions to EL2, from both Execution states.

`0` This control has no effect on the execution of WFI instructions at Non-secure EL1 or Non-secure EL0.

`1` Any attempt to execute a WFI instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state, except that when the value of SCTLR_EL1.nTWI is 0, the trap of EL0 execution to EL1 takes precedence over this trap.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

**DC, bit [12]**

Default Cacheable.

0 This control has no effect on the Non-secure EL1&0 translation regime.

1 In Non-secure state:

- When EL1 is using AArch64, the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of SCTLR_EL1.
- When EL1 is using AArch32, the PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of SCTLR.
- The PE behaves as if the value of the HCR_EL2.VM field is 1 for all purposes other than returning the value of a direct read of HCR_EL2.
- The memory type produced by stage 1 of the EL1&0 translation regime is Normal Non-Shareable, Inner Write Back Read-Write Allocate, Outer Write Back Read-Write Allocate.

For the Non-secure PL1&0 translation regime, when this field is 0 and stage 1 translation is disabled, the memory attribute for data accesses assigned by the stage 1 translation is Device-nGnRnE.

This bit is permitted to be cached in a TLB.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**BSU, bits [11:10]**

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from Non-secure EL1 or Non-secure EL0:

00 No effect

01 Inner Shareable

10 Outer Shareable

11 Full system

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**FB, bit [9]**

Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from Non-secure EL1:

AArch32: BPIALL, TLBIALL, TLBIMVA, TLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, ITLBIALL, ITLBIMVA, ITLBIASID, TLBIMVAA, ICIALLU, TLBIMVAL, TLBIMVAAL.

AArch64: TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, TLBI VAALE1, IC IALLU.

0 This field has no effect on the operation of the specified instructions.

1 When one of the specified instruction is executed at Non-secure EL1, the instruction is broadcast within the Inner Shareable shareability domain.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**VSE, bit [8]**

Virtual System Error or Asynchronous Abort.

0 This mechanism is not making a virtual System Error or Asynchronous Abort pending.

1 A virtual System Error or Asynchronous Abort is pending because of this mechanism.

The virtual System Error or Asynchronous Abort is only enabled when the value of HCR_EL2.AMO bit is 1.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**VI, bit [7]**

Virtual IRQ Interrupt.

0 This mechanism is not making a virtual IRQ pending.

1 A virtual IRQ is pending because of this mechanism.

The virtual IRQ is only enabled when the value of HCR_EL2.IMO bit is 1.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**VF, bit [6]**

Virtual FIQ Interrupt.

0 This mechanism is not making a virtual FIQ pending.

1 A virtual FIQ is pending because of this mechanism.

The virtual FIQ is only enabled when the value of HCR_EL2.FMO bit is 1.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**AMO, bit [5]**

Asynchronous External Abort and SError Interrupt routing.

0 When executing at Non-secure Exception levels below EL2, physical Asynchronous External Aborts and SError Interrupts are not taken to EL2.

When executing at EL2 using AArch64, physical Asynchronous External Aborts and SError Interrupts are not taken unless they are routed to EL3 by the SCR_EL3.EA bit.

Virtual Asynchronous External Aborts and SError Interrupts interrupts are disabled.

1 When executing at any Exception level in Non-secure state:

- Physical Asynchronous External Aborts and SError Interrupts are taken to EL2 unless they are routed to EL3.
- Virtual Asynchronous External Aborts and SError Interrupts are enabled.

If the value of HCR_EL2.TGE is 1, then in Non-secure state the HCR_EL2.AMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

For more information, see *Asynchronous exception routing* on page D1-1569.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**IMO, bit [4]**

Physical IRQ Routing.

0 When executing at Non-secure Exception levels below EL2, physical IRQ interrupts are not taken to EL2.

When executing at EL2 using AArch64, physical IRQ interrupts are not taken unless they are routed to EL3 by the SCR_EL3.IRQ bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Virtual IRQ interrupts are disabled.

1 When executing at any Exception level in Non-secure state:

- Physical IRQ interrupts are taken to EL2 unless they are routed to EL3.
- Virtual IRQ interrupts are enabled in Non-secure state.

If the value of HCR_EL2.TGE is 1, then in Non-secure state the HCR_EL2.IMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

For more information, see *Asynchronous exception routing* on page D1-1569.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**FMO, bit [3]**

Physical FIQ Routing.

0 When executing at Non-secure Exception levels below EL2, physical FIQ interrupts are not taken to EL2.

When executing at EL2 using AArch64, physical FIQ interrupts are not taken unless they are routed to EL3 by the SCR_EL3.FIQ bit.

Virtual FIQ interrupts are disabled.

1 When executing at any Exception level in Non-secure state:

- Physical FIQ interrupts are taken to EL2 unless they are routed to EL3.
- Virtual FIQ interrupts are enabled in Non-secure state.

If the value of HCR_EL2.TGE is 1, then in Non-secure state the HCR_EL2.FMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

For more information, see *Asynchronous exception routing* on page D1-1569.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**PTW, bit [2]**

Protected Table Walk. In the Non-secure EL1&0 translation regime, a translation table access made as part of a stage 1 translation table walk is subject to a stage 2 translation. The combining of the memory type attributes from the two stages of translation means the access might be made to a type of Device memory. If this occurs then the value of this bit determines the behavior:

0 The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.

1 The memory access generates a stage 2 Permission fault.

This bit is permitted to be cached in a TLB.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**SWIO, bit [1]**

Set/Way Invalidation Override. Causes Non-secure EL1 execution of the data cache invalidate by set/way instructions to be treated as data cache clean and invalidate by set/way:

0 This control has no effect on the operation of data cache invalidate by set/way instructions.

1 Data cache invalidate by set/way instructions operate as data cache clean and invalidate by set/way.

When the value of this bit is 1:

AArch32: DCISW is executed as DCCISW.

AArch64: DC ISW is executed as DC CISW.

This bit can be implemented as RES1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

**VM, bit [0]**

Virtualization enable. Enables stage 2 address translation for the Non-secure EL1&0 translation regime. Possible values of this bit are:

`0` Non-secure EL1&0 stage 2 address translation disabled.

`1` Non-secure EL1&0 stage 2 address translation enabled.

When the value of this bit is 1, data cache invalidate instructions executed at Non-secure EL1 operate as data cache clean and invalidate instructions. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR_EL2.SWIO bit.

This bit is permitted to be cached in a TLB.

In an implementation that includes EL3, when the value of SCR_EL3.NS is 0 the PE behaves as if this field is 0 for all purposes other than returning the result of a direct read of HCR_EL2.

### Accessing the HCR_EL2:

To access the HCR_EL2:

```
MRS <Xt>, HCR_EL2 ; Read HCR_EL2 into Xt
MSR HCR_EL2, <Xt> ; Write Xt to HCR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0001 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.34 HPFAR_EL2, Hypervisor IPA Fault Address Register

The HPFAR_EL2 characteristics are:

**Purpose**

Holds the faulting IPA for some aborts on a stage 2 translation taken to EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

Execution at EL1 or EL0 makes HPFAR_EL2 become UNKNOWN.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register HPFAR_EL2[31:0] is architecturally mapped to AArch32 System register HPFAR.

If EL2 is not implemented, this register is RES0 from EL3.

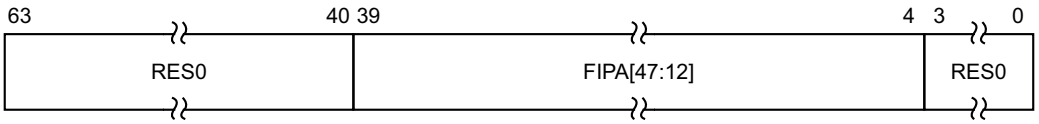
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HPFAR_EL2 is a 64-bit register.

#### Field descriptions

The HPFAR_EL2 bit assignments are:




**Bits [63:40]**

Reserved, RES0.

**FIPA[47:12], bits [39:4]**

Bits [47:12] of the faulting intermediate physical address. For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

The HPFAR_EL2 is written for:

- Translation or Access faults in the second stage of translation.
- An abort in the second stage of translation performed during the translation table walk of a first stage translation, caused by a Translation fault, an Access flag fault, or a Permission fault.
- A stage 2 Address size fault.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the instruction or data abort. It is the lowest address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

For all other exceptions taken to EL2, this register is UNKNOWN.

**Bits [3:0]**

Reserved, RES0.

### Accessing the HPFAR_EL2:

To access the HPFAR_EL2:

```
MRS <Xt>, HPFAR_EL2 ; Read HPFAR_EL2 into Xt
MSR HPFAR_EL2, <Xt> ; Write Xt to HPFAR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0110 | 0000 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.35 HSTR_EL2, Hypervisor System Trap Register

The HSTR_EL2 characteristics are:

**Purpose**

Controls trapping to Hyp mode of Non-secure accesses, at EL1 or lower in AArch32, to the CP15 primary coprocessor registers, {c0-c3,c5-c13,c15} using MCR, MRC, MCRR, and MRRC instructions.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register HSTR_EL2 is architecturally mapped to AArch32 System register HSTR.

If EL2 is not implemented, this register is RES0 from EL3.

If no Exception level can use AArch32, then this register is RES0

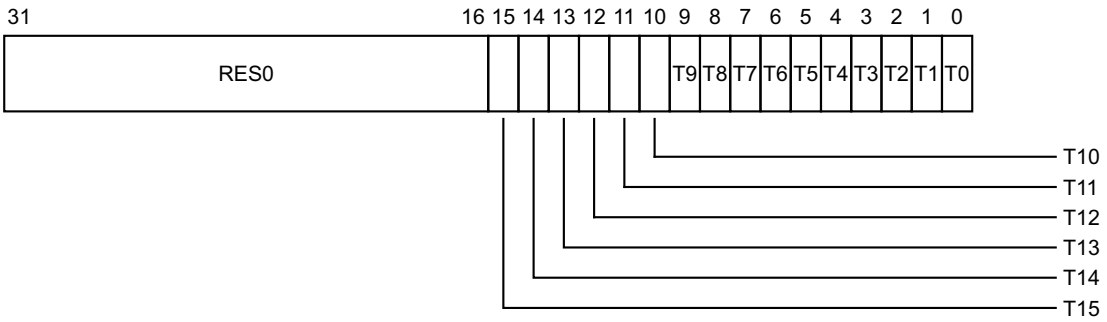
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HSTR_EL2 is a 32-bit register.

### Field descriptions

The HSTR_EL2 bit assignments are:




**Bits [31:16]**

Reserved, RES0.

**T<n>, bit [n], for n = 0 to 15**

HSTR_EL2.{T0-T3, T5-T13, T15} trap Non-secure EL1 or EL0 accesses, using MCR, MRC, MCRR, and MRRC instructions, to the CP15 System registers, by the accessed primary CP15 register number, {c0-c3, c5-c13, c15}, when EL1 or EL0 is using AArch32.

`0` Non-secure EL0 or EL1 accesses, from AArch32, to the corresponding CP15 System registers are not trapped to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 Any Non-secure EL1 MCR, MRC, MCRR, or MRRC access, from AArch32, to a corresponding register is trapped to EL2.

A Non-secure EL0 MCR, MRC, MCRR, or MRRC access, from AArch32, to the corresponding CP15 System register is trapped to EL2 only if it is not UNDEFINED when this bit is zero.

The primary CP15 register number is the value of <CRn> for an MCR or MRC instruction, or the value of <CRm> for an MCRR or MRRC instruction.

For example, when HSTR_EL2.T7 is 1:

- Any 32-bit CP15 access from a Non-secure EL1 mode using AArch32, using an MCR or MRC instruction with <CRn> set to c7, is trapped to EL2.
- Any 64-bit CP15 access from a Non-secure EL1 mode using AArch32, using an MCRR or MRRC instruction with CRm set to c7, is trapped to EL2.

Fields T4 and T14 are RES0.

### Accessing the HSTR_EL2:

To access the HSTR_EL2:

```
MRS <Xt>, HSTR_EL2 ; Read HSTR_EL2 into Xt
MSR HSTR_EL2, <Xt> ; Write Xt to HSTR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0001 | 0001 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.36 ID_AA64AFR0_EL1, AArch64 Auxiliary Feature Register 0

The ID_AA64AFR0_EL1 characteristics are:

**Purpose**

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch64.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

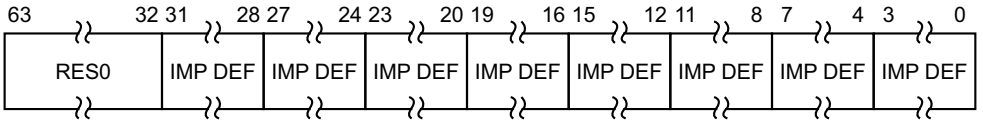
There are no configuration notes.

**Attributes**

ID_AA64AFR0_EL1 is a 64-bit register.

## Field descriptions

The ID_AA64AFR0_EL1 bit assignments are:




**Bits [63:32]**

Reserved, RES0.

**IMPLEMENTATION DEFINED, bits [31:28]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [27:24]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [23:20]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [19:16]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [15:12]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [11:8]**

IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IMPLEMENTATION DEFINED, bits [7:4]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [3:0]**

IMPLEMENTATION DEFINED.

### Accessing the ID_AA64AFR0_EL1:

To access the ID_AA64AFR0_EL1:

```
MRS <Xt>, ID_AA64AFR0_EL1 ; Read ID_AA64AFR0_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0101 | 100 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## D7.2.37 ID_AA64AFR1_EL1, AArch64 Auxiliary Feature Register 1

The ID_AA64AFR1_EL1 characteristics are:

**Purpose**

Reserved for future expansion of information about the IMPLEMENTATION DEFINED features of the PE in AArch64.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

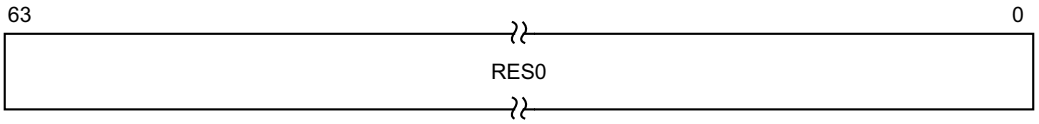
There are no configuration notes.

**Attributes**

ID_AA64AFR1_EL1 is a 64-bit register.

### Field descriptions

The ID_AA64AFR1_EL1 bit assignments are:




**Bits [63:0]**

Reserved, RES0.

### Accessing the ID_AA64AFR1_EL1:

To access the ID_AA64AFR1_EL1:

```
MRS <Xt>, ID_AA64AFR1_EL1 ; Read ID_AA64AFR1_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0101 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.38 ID_AA64DFR0_EL1, AArch64 Debug Feature Register 0

The ID_AA64DFR0_EL1 characteristics are:

**Purpose**

Provides top level information about the debug system in AArch64.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

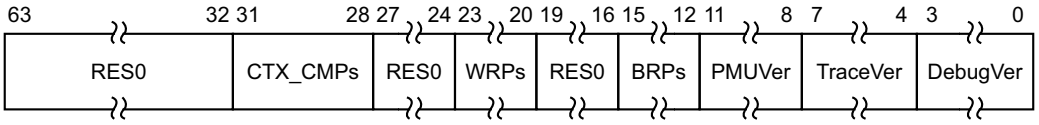
AArch64 System register ID_AA64DFR0_EL1 is architecturally mapped to External register EDDFR.

**Attributes**

ID_AA64DFR0_EL1 is a 64-bit register.

#### Field descriptions

The ID_AA64DFR0_EL1 bit assignments are:




**Bits [63:32]**

Reserved, RES0.

**CTX_CMPs, bits [31:28]**

Number of breakpoints that are context-aware, minus 1. These are the highest numbered breakpoints.

**Bits [27:24]**

Reserved, RES0.

**WRPs, bits [23:20]**

Number of watchpoints, minus 1. The value of `0b0000` is reserved.

**Bits [19:16]**

Reserved, RES0.

**BRPs, bits [15:12]**

Number of breakpoints, minus 1. The value of `0b0000` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**PMUVer, bits [11:8]**

Performance Monitors extension version. Indicates whether System register interface to Performance Monitors extension is implemented. Defined values are:

`0000` Performance Monitors extension System registers not implemented.

`0001` Performance Monitors extension System registers implemented, PMUv3.

`1111` IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported.

All other values are reserved.

**TraceVer, bits [7:4]**

Trace support. Indicates whether System register interface to a trace macrocell is implemented. Defined values are:

`0000` Trace macrocell System registers not implemented.

`0001` Trace macrocell System registers implemented.

All other values are reserved.

A value of `0000` only indicates that no System register interface to a trace macrocell is implemented. A trace macrocell might nevertheless be implemented without a System register interface.

**DebugVer, bits [3:0]**

Debug architecture version. Indicates presence of ARMv8 debug architecture.

`0110` ARMv8 debug architecture.

All other values are reserved.

## Accessing the ID_AA64DFR0_EL1:

To access the ID_AA64DFR0_EL1:

```
MRS <Xt>, ID_AA64DFR0_EL1 ; Read ID_AA64DFR0_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.39 ID_AA64DFR1_EL1, AArch64 Debug Feature Register 1

The ID_AA64DFR1_EL1 characteristics are:

**Purpose**

Reserved for future expansion of top level information about the debug system in AArch64.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

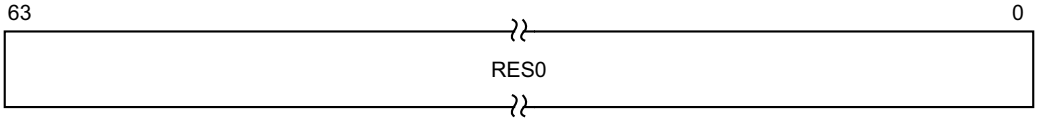
There are no configuration notes.

**Attributes**

ID_AA64DFR1_EL1 is a 64-bit register.

#### Field descriptions

The ID_AA64DFR1_EL1 bit assignments are:




**Bits [63:0]**

Reserved, RES0.

#### Accessing the ID_AA64DFR1_EL1:

To access the ID_AA64DFR1_EL1:

```
MRS <Xt>, ID_AA64DFR1_EL1 ; Read ID_AA64DFR1_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0101 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.40 ID_AA64ISAR0_EL1, AArch64 Instruction Set Attribute Register 0

The ID_AA64ISAR0_EL1 characteristics are:

**Purpose**

Provides information about the instructions implemented in AArch64 state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

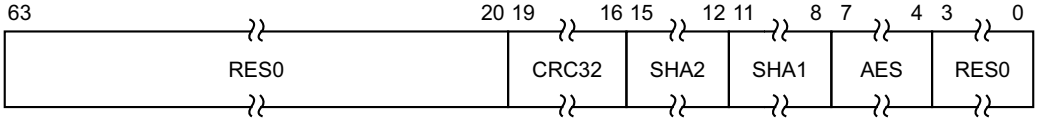
There are no configuration notes.

**Attributes**

ID_AA64ISAR0_EL1 is a 64-bit register.

#### Field descriptions

The ID_AA64ISAR0_EL1 bit assignments are:




**Bits [63:20]**

Reserved, RES0.

**CRC32, bits [19:16]**

CRC32 instructions in AArch64. Defined values are:

0000 No CRC32 instructions implemented.

0001 CRC32B, CRC32H, CRC32W, CRC32X, CRC32CB, CRC32CH, CRC32CW, and CRC32CX instructions implemented.

All other values are reserved.

**SHA2, bits [15:12]**

SHA2 instructions in AArch64. Defined values are:

0000 No SHA2 instructions implemented.

0001 SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 instructions implemented.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SHA1, bits [11:8]**

SHA1 instructions in AArch64. Defined values are:

`0000` No SHA1 instructions implemented.

`0001` SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 instructions implemented.

All other values are reserved.

**AES, bits [7:4]**

AES instructions in AArch64. Defined values are:

`0000` No AES instructions implemented.

`0001` AESE, AESD, AESMC, and AESIMC instructions implemented.

`0010` As for `0001`, plus PMULL/PMULL2 instructions operating on 64-bit data quantities.

All other values are reserved.

**Bits [3:0]**

Reserved, RES0.

## Accessing the ID_AA64ISAR0_EL1:

To access the ID_AA64ISAR0_EL1:

```
MRS <Xt>, ID_AA64ISAR0_EL1 ; Read ID_AA64ISAR0_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0110 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.41 ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1

The ID_AA64ISAR1_EL1 characteristics are:

**Purpose**

Reserved for future expansion of the information about the instructions implemented in AArch64 state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

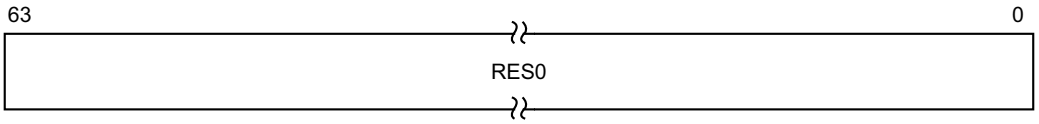
There are no configuration notes.

**Attributes**

ID_AA64ISAR1_EL1 is a 64-bit register.

### Field descriptions

The ID_AA64ISAR1_EL1 bit assignments are:




**Bits [63:0]**

Reserved, RES0.

### Accessing the ID_AA64ISAR1_EL1:

To access the ID_AA64ISAR1_EL1:

```
MRS <Xt>, ID_AA64ISAR1_EL1 ; Read ID_AA64ISAR1_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0110 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.42 ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0

The ID_AA64MMFR0_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch64.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

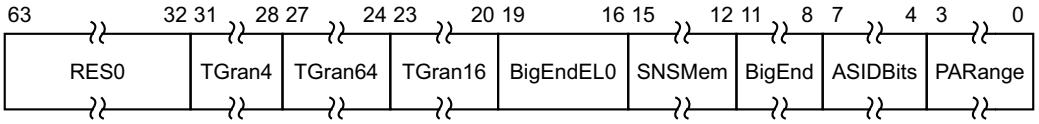
There are no configuration notes.

**Attributes**

ID_AA64MMFR0_EL1 is a 64-bit register.

### Field descriptions

The ID_AA64MMFR0_EL1 bit assignments are:




**Bits [63:32]**

Reserved, RES0.

**TGran4, bits [31:28]**

Support for 4KB memory translation granule size. Defined values are:

0000 4KB granule supported.

1111 4KB granule not supported.

All other values are reserved.

**TGran64, bits [27:24]**

Support for 64KB memory translation granule size. Defined values are:

0000 64KB granule supported.

1111 64KB granule not supported.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TGran16, bits [23:20]**

Support for 16KB memory translation granule size. Defined values are:

`0000` 16KB granule not supported.

`0001` 16KB granule supported.

All other values are reserved.

**BigEndEL0, bits [19:16]**

Mixed-endian support at EL0 only. Defined values are:

`0000` No mixed-endian support at EL0. The SCTLR_EL1.E0E bit has a fixed value.

`0001` Mixed-endian support at EL0. The SCTLR_EL1.E0E bit can be configured.

All other values are reserved.

This field is invalid and is RES0 if the BigEnd field, bits [11:8], is not `0000`.

**SNSMem, bits [15:12]**

Secure versus Non-secure Memory distinction. Defined values are:

`0000` Does not support a distinction between Secure and Non-secure Memory.

`0001` Does support a distinction between Secure and Non-secure Memory.

All other values are reserved.

**BigEnd, bits [11:8]**

Mixed-endian configuration support. Defined values are:

`0000` No mixed-endian support. The SCTLR_ELx.EE bits have a fixed value. See the BigEndEL0 field, bits[19:16], for whether EL0 supports mixed-endian.

`0001` Mixed-endian support. The SCTLR_ELx.EE and SCTLR_EL1.E0E bits can be configured.

All other values are reserved.

**ASIDBits, bits [7:4]**

Number of ASID bits. Defined values are:

`0000` 8 bits.

`0010` 16 bits.

All other values are reserved.

**PARange, bits [3:0]**

Physical Address range supported. Defined values are:

`0000` 32 bits, 4GB.

`0001` 36 bits, 64GB.

`0010` 40 bits, 1TB.

`0011` 42 bits, 4TB.

`0100` 44 bits, 16TB.

`0101` 48 bits, 256TB.

All other values are reserved.

## Accessing the ID_AA64MMFR0_EL1:

To access the ID_AA64MMFR0_EL1:

```
MRS <Xt>, ID_AA64MMFR0_EL1 ; Read ID_AA64MMFR0_EL1 into Xt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0111 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.43 ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1

The ID_AA64MMFR1_EL1 characteristics are:

**Purpose**

Reserved for future expansion of the information about the implemented memory model and memory management support in AArch64.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

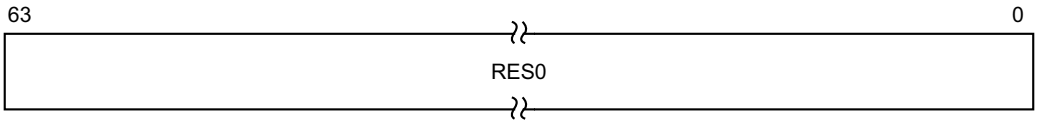
There are no configuration notes.

**Attributes**

ID_AA64MMFR1_EL1 is a 64-bit register.

#### Field descriptions

The ID_AA64MMFR1_EL1 bit assignments are:




**Bits [63:0]**

Reserved, RES0.

#### Accessing the ID_AA64MMFR1_EL1:

To access the ID_AA64MMFR1_EL1:

```
MRS <Xt>, ID_AA64MMFR1_EL1 ; Read ID_AA64MMFR1_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0111 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.44 ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0

The ID_AA64PFR0_EL1 characteristics are:

**Purpose**

Provides additional information about implemented PE features in AArch64.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

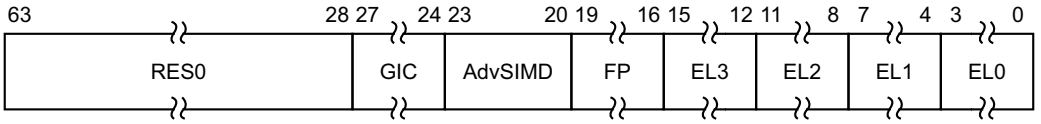
AArch64 System register ID_AA64PFR0_EL1 is architecturally mapped to External register EDPFR.

**Attributes**

ID_AA64PFR0_EL1 is a 64-bit register.

#### Field descriptions

The ID_AA64PFR0_EL1 bit assignments are:




**Bits [63:28]**

Reserved, RES0.

**GIC, bits [27:24]**

System register GIC interface support. Defined values are:

0000 No System register interface to the GIC is supported.

0001 System register interface to the GIC CPU interface is supported.

All other values are reserved.

**AdvSIMD, bits [23:20]**

Advanced SIMD. Defined values are:

0000 Advanced SIMD is implemented.

1111 Advanced SIMD is not implemented.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**FP, bits [19:16]**

Floating-point. Defined values are:

0000 Floating-point is implemented.

1111 Floating-point is not implemented.

All other values are reserved.

**EL3, bits [15:12]**

EL3 Exception level handling. Defined values are:

0000 EL3 is not implemented.

0001 EL3 can be executed in AArch64 state only.

0010 EL3 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

**EL2, bits [11:8]**

EL2 Exception level handling. Defined values are:

0000 EL2 is not implemented.

0001 EL2 can be executed in AArch64 state only.

0010 EL2 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

**EL1, bits [7:4]**

EL1 Exception level handling. Defined values are:

0001 EL1 can be executed in AArch64 state only.

0010 EL1 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

**EL0, bits [3:0]**

EL0 Exception level handling. Defined values are:

0001 EL0 can be executed in AArch64 state only.

0010 EL0 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

## Accessing the ID_AA64PFR0_EL1:

To access the ID_AA64PFR0_EL1:

```
MRS <Xt>, ID_AA64PFR0_EL1 ; Read ID_AA64PFR0_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0100 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.45 ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1

The ID_AA64PFR1_EL1 characteristics are:

**Purpose**

Reserved for future expansion of information about implemented PE features in AArch64.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

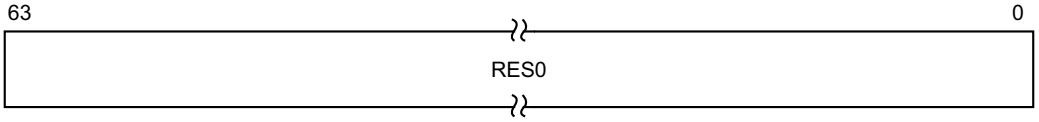
There are no configuration notes.

**Attributes**

ID_AA64PFR1_EL1 is a 64-bit register.

#### Field descriptions

The ID_AA64PFR1_EL1 bit assignments are:




**Bits [63:0]**

Reserved, RES0.

#### Accessing the ID_AA64PFR1_EL1:

To access the ID_AA64PFR1_EL1:

```
MRS <Xt>, ID_AA64PFR1_EL1 ; Read ID_AA64PFR1_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0100 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.46 ID_AFR0_EL1, AArch32 Auxiliary Feature Register 0

The ID_AFR0_EL1 characteristics are:

**Purpose**

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with the Main ID Register, MIDR_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_AFR0_EL1 is architecturally mapped to AArch32 System register ID_AFR0.

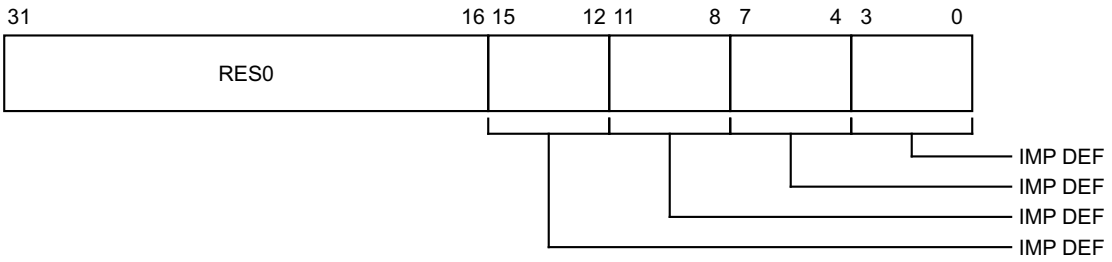
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_AFR0_EL1 is a 32-bit register.

#### Field descriptions

The ID_AFR0_EL1 bit assignments are:




**Bits [31:16]**

Reserved, RES0.

**IMPLEMENTATION DEFINED, bits [15:12]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [11:8]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [7:4]**

IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IMPLEMENTATION DEFINED, bits [3:0]**

IMPLEMENTATION DEFINED.

## Accessing the ID_AFR0_EL1:

To access the ID_AFR0_EL1:

```
MRS <Xt>, ID_AFR0_EL1 ; Read ID_AFR0_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0001 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.47 ID_DFR0_EL1, AArch32 Debug Feature Register 0

The ID_DFR0_EL1 characteristics are:

**Purpose**

Provides top level information about the debug system in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with the Main ID Register, MIDR_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_DFR0_EL1 is architecturally mapped to AArch32 System register ID_DFR0.

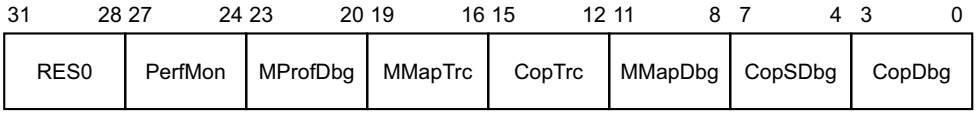
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_DFR0_EL1 is a 32-bit register.

#### Field descriptions

The ID_DFR0_EL1 bit assignments are:




**Bits [31:28]**

Reserved, RES0.

**PerfMon, bits [27:24]**

Performance Monitors. Support for coprocessor-based ARM Performance Monitors Extension, for A and R profile processors. Defined values are:

`0000` Performance Monitors Extension System registers not implemented.

`0001` Support for Performance Monitors Extension version 1 (PMUv1) System registers.

`0010` Support for Performance Monitors Extension version 2 (PMUv2) System registers.

`0011` Support for Performance Monitors Extension version 3 (PMUv3) System registers.

1111 IMPLEMENTATION DEFINED form of Performance Monitors System registers supported. PMUv3 not supported.

All other values are reserved.

In ARMv8-A the permitted values are `0000`, `0011`, and 1111.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In ARMv7, the value 0000 can mean that PMUv1 is implemented. PMUv1 is not permitted in an ARMv8 implementation.

**MProfDbg, bits [23:20]**

M Profile Debug. Support for memory-mapped debug model for M profile processors. Defined values are:

0000 Not supported.

0001 Support for M profile Debug architecture, with memory-mapped access.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**MMapTrc, bits [19:16]**

Memory Mapped Trace. Support for memory-mapped trace model. Defined values are:

0000 Not supported.

0001 Support for ARM trace architecture, with memory-mapped access.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

In the Trace registers, the ETMIDR gives more information about the implementation.

**CopTrc, bits [15:12]**

Coprocessor Trace. Support for coprocessor-based trace model. Defined values are:

0000 Not supported.

0001 Support for ARM trace architecture, with CP14 access.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

In the Trace registers, the ETMIDR gives more information about the implementation.

**MMapDbg, bits [11:8]**

Memory Mapped Debug. Support for v7 memory-mapped debug model, for A and R profile processors.

In ARMv8-A this field is RES0.

The optional memory map defined by ARMv8 is not compatible with ARMv7.

**CopSDbg, bits [7:4]**

Coprocessor Secure Debug. Support for a Secure debug model accessed through a conceptual coprocessor, for an A profile processor that includes EL3.

If EL3 is not implemented and the implemented Security state is Non-Secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].

**CopDbg, bits [3:0]**

Coprocessor Debug. Support for coprocessor based debug model, for A and R profile processors. Defined values are:

0000 Not supported.

0010 Support for ARMv6, v6 Debug architecture, with CP14 access.

0011 Support for ARMv6, v6.1 Debug architecture, with CP14 access.

0100 Support for ARMv7, v7 Debug architecture, with CP14 access.

0101 Support for ARMv7, v7.1 Debug architecture, with CP14 access.

0110 Support for ARMv8 debug architecture, with CP14 access.

All other values are reserved.

In ARMv8-A the permitted values are 0000, and 0110.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the ID_DFR0_EL1:

To access the ID_DFR0_EL1:

```
MRS <Xt>, ID_DFR0_EL1 ; Read ID_DFR0_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0001 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.48 ID_ISAR0_EL1, AArch32 Instruction Set Attribute Register 0

The ID_ISAR0_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_ISAR0_EL1 is architecturally mapped to AArch32 System register ID_ISAR0.

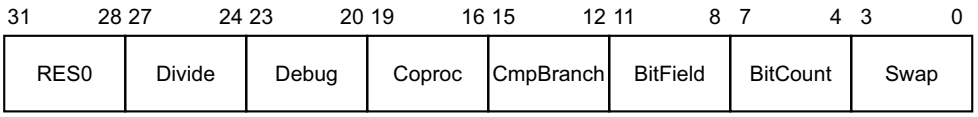
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_ISAR0_EL1 is a 32-bit register.

#### Field descriptions

The ID_ISAR0_EL1 bit assignments are:




**Bits [31:28]**

Reserved, RES0.

**Divide, bits [27:24]**

Indicates the implemented Divide instructions. Defined values are:

`0000` None implemented.

`0001` Adds SDIV and UDIV in the T32 instruction set.

`0010` As for `0001`, and adds SDIV and UDIV in the A32 instruction set.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Debug, bits [23:20]**

Indicates the implemented Debug instructions. Defined values are:

`0000` None implemented.

`0001` Adds BKPT.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Coproc, bits [19:16]**

Indicates the implemented Coprocessor instructions. Defined values are:

`0000` None implemented, except for instructions separately attributed by the architecture, including CP15, CP14, and Advanced SIMD and VFP.

`0001` Adds generic CDP, LDC, MCR, MRC, and STC.

`0010` As for `0001`, and adds generic CDP2, LDC2, MCR2, MRC2, and STC2.

`0011` As for `0010`, and adds generic MCRR and MRRC.

`0100` As for `0011`, and adds generic MCRR2 and MRRC2.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**CmpBranch, bits [15:12]**

Indicates the implemented combined Compare and Branch instructions in the T32 instruction set. Defined values are:

`0000` None implemented.

`0001` Adds CBNZ and CBZ.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**BitField, bits [11:8]**

Indicates the implemented BitField instructions. Defined values are:

`0000` None implemented.

`0001` Adds BFC, BFI, SBFX, and UBFX.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**BitCount, bits [7:4]**

Indicates the implemented Bit Counting instructions. Defined values are:

`0000` None implemented.

`0001` Adds CLZ.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Swap, bits [3:0]**

Indicates the implemented Swap instructions in the A32 instruction set. Defined values are:

`0000` None implemented.

`0001` Adds SWP and SWPB.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the ID_ISAR0_EL1:

To access the ID_ISAR0_EL1:

```
MRS <Xt>, ID_ISAR0_EL1 ; Read ID_ISAR0_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.49 ID_ISAR1_EL1, AArch32 Instruction Set Attribute Register 1

The ID_ISAR1_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_ISAR0_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_ISAR1_EL1 is architecturally mapped to AArch32 System register ID_ISAR1.

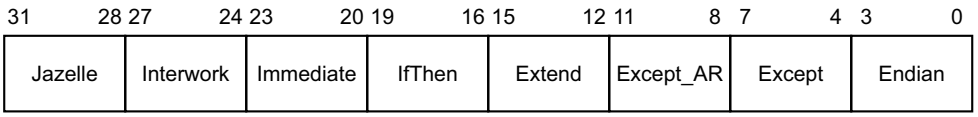
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_ISAR1_EL1 is a 32-bit register.

#### Field descriptions

The ID_ISAR1_EL1 bit assignments are:




**Jazelle, bits [31:28]**

Indicates the implemented Jazelle extension instructions. Defined values are:

`0000` No support for Jazelle.

`0001` Adds the BXJ instruction, and the J bit in the PSR. This setting might indicate a trivial implementation of the Jazelle extension.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Interwork, bits [27:24]**

Indicates the implemented Interworking instructions. Defined values are:

`0000` None implemented.

`0001` Adds the BX instruction, and the T bit in the PSR.

`0010` As for `0001`, and adds the BLX instruction. PC loads have BX-like behavior.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0011 As for 0010, and guarantees that data-processing instructions in the A32 instruction set with the PC as the destination and the S bit clear have BX-like behavior.

All other values are reserved.

In ARMv8-A the only permitted value is 0011.

**Immediate, bits [23:20]**

Indicates the implemented data-processing instructions with long immediates. Defined values are:

0000 None implemented.

0001 Adds:

- The MOVT instruction.
- The MOV instruction encodings with zero-extended 16-bit immediates.
- The T32 ADD and SUB instruction encodings with zero-extended 12-bit immediates, and the other ADD, ADR, and SUB encodings cross-referenced by the pseudocode for those encodings.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**IfThen, bits [19:16]**

Indicates the implemented If-Then instructions in the T32 instruction set. Defined values are:

0000 None implemented.

0001 Adds the IT instructions, and the IT bits in the PSRs.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**Extend, bits [15:12]**

Indicates the implemented Extend instructions. Defined values are:

0000 No scalar sign-extend or zero-extend instructions are implemented, where scalar instructions means non-Advanced SIMD instructions.

0001 Adds the SXTB, SXTH, UXTB, and UXTH instructions.

0010 As for 0001, and adds the SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

**Except_AR, bits [11:8]**

Indicates the implemented A and R profile exception-handling instructions. Defined values are:

0000 None implemented.

0001 Adds the SRS and RFE instructions, and the A and R profile forms of the CPS instruction.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**Except, bits [7:4]**

Indicates the implemented exception-handling instructions in the ARM instruction set. Defined values are:

0000 Not implemented. This indicates that the User bank and Exception return forms of the LDM and STM instructions are not implemented.

0001 Adds the LDM (exception return), LDM (user registers), and STM (user registers) instruction versions.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In ARMv8-A the only permitted value is 0001.

**Endian, bits [3:0]**

Indicates the implemented Endian instructions. Defined values are:

0000 None implemented.

0001 Adds the SETEND instruction, and the E bit in the PSRs.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

### Accessing the ID_ISAR1_EL1:

To access the ID_ISAR1_EL1:

```
MRS <Xt>, ID_ISAR1_EL1 ; Read ID_ISAR1_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.50 ID_ISAR2_EL1, AArch32 Instruction Set Attribute Register 2

The ID_ISAR2_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_ISAR2_EL1 is architecturally mapped to AArch32 System register ID_ISAR2.

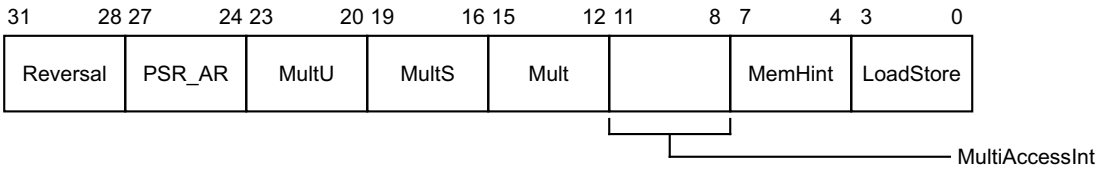
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_ISAR2_EL1 is a 32-bit register.

## Field descriptions

The ID_ISAR2_EL1 bit assignments are:




**Reversal, bits [31:28]**

Indicates the implemented Reversal instructions. Defined values are:

`0000` None implemented.

`0001` Adds the REV, REV16, and REVSH instructions.

`0010` As for `0001`, and adds the RBIT instruction.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

**PSR_AR, bits [27:24]**

Indicates the implemented A and R profile instructions to manipulate the PSR. Defined values are:

`0000` None implemented.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0001 Adds the MRS and MSR instructions, and the exception return forms of data-processing instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

The exception return forms of the data-processing instructions are:

- In the A32 instruction set, data-processing instructions with the PC as the destination and the S bit set. These instructions might be affected by the WithShifts attribute.
- In the T32 instruction set, the SUBS PC,LR,#N instruction.

**MultU, bits [23:20]**

Indicates the implemented advanced unsigned Multiply instructions. Defined values are:

0000 None implemented.

0001 Adds the UMULL and UMLAL instructions.

0010 As for 0001, and adds the UMAAL instruction.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

**MultS, bits [19:16]**

Indicates the implemented advanced signed Multiply instructions. Defined values are:

0000 None implemented.

0001 Adds the SMULL and SMLAL instructions.

0010 As for 0001, and adds the SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, and SMULWT instructions. Also adds the Q bit in the PSRs.

0011 As for 0010, and adds the SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSDX, SMLSLD, SMLSLDX, SMMLA, SMMLAR, SMMLS, SMMLSR, SMMUL, SMMULR, SMUAD, SMUADX, SMUSD, and SMUSDX instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0011.

**Mult, bits [15:12]**

Indicates the implemented additional Multiply instructions. Defined values are:

0000 No additional instructions implemented. This means only MUL is implemented.

0001 Adds the MLA instruction.

0010 As for 0001, and adds the MLS instruction.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

**MultiAccessInt, bits [11:8]**

Indicates the support for interruptible multi-access instructions. Defined values are:

0000 No support. This means the LDM and STM instructions are not interruptible.

0001 LDM and STM instructions are restartable.

0010 LDM and STM instructions are continuable.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**MemHint, bits [7:4]**

Indicates the implemented Memory Hint instructions. Defined values are:

`0000` None implemented.

`0001` Adds the PLD instruction.

`0010` Adds the PLD instruction. (`0001` and `0010` have identical effects.)

`0011` As for `0001` (or `0010`), and adds the PLI instruction.

`0100` As for `0011`, and adds the PLDW instruction.

All other values are reserved.

In ARMv8-A the only permitted value is `0100`.

**LoadStore, bits [3:0]**

Indicates the implemented additional load/store instructions. Defined values are:

`0000` No additional load/store instructions implemented.

`0001` Adds the LDRD and STRD instructions.

`0010` As for `0001`, and adds the Load Acquire (LDAB, LDAH, LDA, LDAEXB, LDAEXH, LDAEX, LDAEXD) and Store Release (STLB, STLH, STL, STLEXB, STLEXH, STLEX, STLEXD) instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

## Accessing the ID_ISAR2_EL1:

To access the ID_ISAR2_EL1:

```
MRS <Xt>, ID_ISAR2_EL1 ; Read ID_ISAR2_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0010 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.51 ID_ISAR3_EL1, AArch32 Instruction Set Attribute Register 3

The ID_ISAR3_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_ISAR3_EL1 is architecturally mapped to AArch32 System register ID_ISAR3.

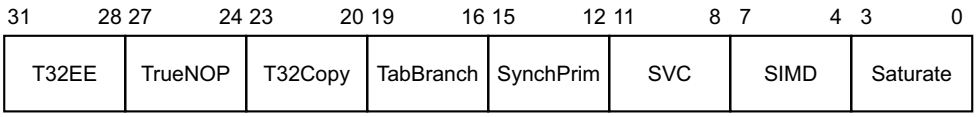
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_ISAR3_EL1 is a 32-bit register.

### Field descriptions

The ID_ISAR3_EL1 bit assignments are:




**T32EE, bits [31:28]**

Indicates the implemented T32EE instructions. Defined values are:

`0000` None implemented.

`0001` Adds the ENTERX and LEAVEX instructions, and modifies the load behavior to include null checking.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**TrueNOP, bits [27:24]**

Indicates the implemented true NOP instructions. Defined values are:

`0000` None implemented. This means there are no NOP instructions that do not have any register dependencies.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0001 Adds true NOP instructions in both the T32 and A32 instruction sets. This also permits additional NOP-compatible hints.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**T32Copy, bits [23:20]**

Indicates the support for T32 non flag-setting MOV instructions. Defined values are:

0000 Not supported. This means that in the T32 instruction set, encoding T1 of the MOV (register) instruction does not support a copy from a low register to a low register.

0001 Adds support for T32 instruction set encoding T1 of the MOV (register) instruction, copying from a low register to a low register.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**TabBranch, bits [19:16]**

Indicates the implemented Table Branch instructions in the T32 instruction set. Defined values are:

0000 None implemented.

0001 Adds the TBB and TBH instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**SynchPrim, bits [15:12]**

Used in conjunction with ID_ISAR4.SynchPrim_frac to indicate the implemented Synchronization Primitive instructions. Defined values are:

0000 If SynchPrim_frac == 0000, no Synchronization Primitives implemented.

0001 If SynchPrim_frac == 0000, adds the LDREX and STREX instructions.
If SynchPrim_frac == 0011, also adds the CLREX, LDREXB, STREXB, and STREXH instructions.

0010 If SynchPrim_frac == 0000, as for [0001, 0011] and also adds the LDREXD and STREXD instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In ARMv8-A the only permitted value is 0010.

**SVC, bits [11:8]**

Indicates the implemented SVC instructions. Defined values are:

0000 Not implemented.

0001 Adds the SVC instruction.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**SIMD, bits [7:4]**

Indicates the implemented SIMD instructions. Defined values are:

0000 None implemented.

0001 Adds the SSAT and USAT instructions, and the Q bit in the PSRs.

0011 As for 0001, and adds the PKHBT, PKHTB, QADD16, QADD8, QASX, QSUB16, QSUB8, QSAX, SADD16, SADD8, SASX, SEL, SHADD16, SHADD8, SHASX, SHSUB16, SHSUB8, SHSAX, SSAT16, SSUB16, SSUB8, SSAX, SXTAB16, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSUB16, UHSUB8, UHSAX, UQADD16, UQADD8, UQASX, UQSUB16, UQSUB8, UQSAX, USAD8, USADA8, USAT16, USUB16, USUB8, USAX, UXTAB16, and UXTB16 instructions. Also adds support for the GE[3:0] bits in the PSRs.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

In ARMv8-A the only permitted value is 0011.

The SIMD field relates only to implemented instructions that perform SIMD operations on the general-purpose registers. In an implementation that supports floating-point and Advanced SIMD instructions, MVFR0, MVFR1, and MVFR2 give information about the implemented Advanced SIMD instructions.

**Saturate, bits [3:0]**

Indicates the implemented Saturate instructions. Defined values are:

0000 None implemented. This means no non-Advanced SIMD saturate instructions are implemented.

0001 Adds the QADD, QDADD, QDSUB, and QSUB instructions, and the Q bit in the PSRs.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

### Accessing the ID_ISAR3_EL1:

To access the ID_ISAR3_EL1:

```
MRS <Xt>, ID_ISAR3_EL1 ; Read ID_ISAR3_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0010 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.52 ID_ISAR4_EL1, AArch32 Instruction Set Attribute Register 4

The ID_ISAR4_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, and ID_ISAR5_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_ISAR4_EL1 is architecturally mapped to AArch32 System register ID_ISAR4.

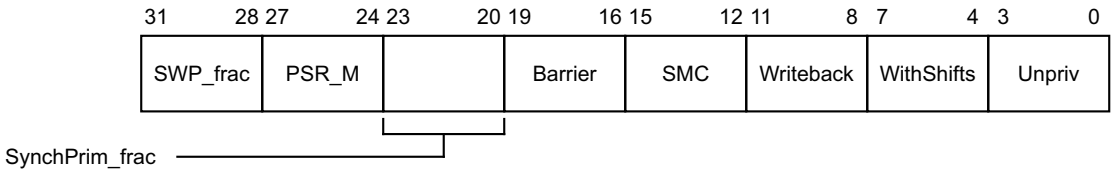
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_ISAR4_EL1 is a 32-bit register.

## Field descriptions

The ID_ISAR4_EL1 bit assignments are:




**SWP_frac, bits [31:28]**

Indicates support for the memory system locking the bus for SWP or SWPB instructions. Defined values are:

`0000` SWP or SWPB instructions not implemented.

`0001` SWP or SWPB implemented but only in a uniprocessor context. SWP and SWPB do not guarantee whether memory accesses from other masters can come between the load memory access and the store memory access of the SWP or SWPB.

All other values are reserved. This field is valid only if the ID_ISAR0.Swap_instrs field is `0000`.

In ARMv8-A the only permitted value is `0000`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**PSR_M, bits [27:24]**

Indicates the implemented M profile instructions to modify the PSRs. Defined values are:

`0000` None implemented.

`0001` Adds the M profile forms of the CPS, MRS, and MSR instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**SynchPrim_frac, bits [23:20]**

Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented Synchronization Primitive instructions. Possible values are:

`0000` If SynchPrim == `0000`, no Synchronization Primitives implemented. If SynchPrim == `0001`, adds the LDREX and STREX instructions. If SynchPrim == `0010`, also adds the CLREX, LDREXB, LDREXH, STREXB, STREXH, LDREXD, and STREXD instructions.

`0011` If SynchPrim == `0001`, adds the LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In ARMv8-A the only permitted value is `0000`.

**Barrier, bits [19:16]**

Indicates the implemented Barrier instructions in the A32 and T32 instruction sets. Defined values are:

`0000` None implemented. Barrier operations are provided only as CP15 operations.

`0001` Adds the DMB, DSB, and ISB barrier instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**SMC, bits [15:12]**

Indicates the implemented SMC instructions. Defined values are:

`0000` None implemented.

`0001` Adds the SMC instruction.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Writeback, bits [11:8]**

Indicates the support for Writeback addressing modes. Defined values are:

`0000` Basic support. Only the LDM, STM, PUSH, POP, SRS, and RFE instructions support writeback addressing modes. These instructions support all of their writeback addressing modes.

`0001` Adds support for all of the writeback addressing modes.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**WithShifts, bits [7:4]**

Indicates the support for instructions with shifts. Defined values are:

`0000` Nonzero shifts supported only in MOV and shift instructions.

`0001` Adds support for shifts of loads and stores over the range LSL 0-3.

`0011` As for `0001`, and adds support for other constant shift options, both on load/store and other instructions.

`0100` As for `0011`, and adds support for register-controlled shift options.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

In ARMv8-A the only permitted value is 0100.

**Unpriv, bits [3:0]**

Indicates the implemented unprivileged instructions. Defined values are:

0000 None implemented. No T variant instructions are implemented.

0001 Adds the LDRBT, LDRT, STRBT, and STRT instructions.

0010 As for 0001, and adds the LDRHT, LDRSBT, LDRSHT, and STRHT instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

### Accessing the ID_ISAR4_EL1:

To access the ID_ISAR4_EL1:

```
MRS <Xt>, ID_ISAR4_EL1 ; Read ID_ISAR4_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0010 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.53 ID_ISAR5_EL1, AArch32 Instruction Set Attribute Register 5

The ID_ISAR5_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, and ID_ISAR4_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_ISAR5_EL1 is architecturally mapped to AArch32 System register ID_ISAR5.

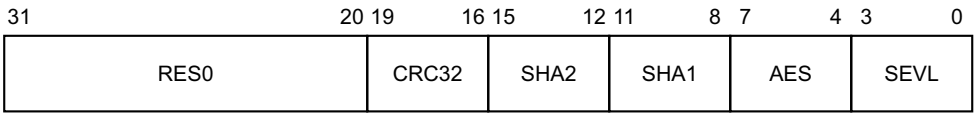
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_ISAR5_EL1 is a 32-bit register.

#### Field descriptions

The ID_ISAR5_EL1 bit assignments are:




**Bits [31:20]**

Reserved, RES0.

**CRC32, bits [19:16]**

Indicates whether CRC32 instructions are implemented in AArch32.

`0000` No CRC32 instructions implemented.

`0001` CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW instructions implemented.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and `0001`.

**SHA2, bits [15:12]**

Indicates whether SHA2 instructions are implemented in AArch32.

`0000` No SHA2 instructions implemented.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0001 SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

**SHA1, bits [11:8]**

Indicates whether SHA1 instructions are implemented in AArch32.

0000 No SHA1 instructions implemented.

0001 SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

**AES, bits [7:4]**

Indicates whether AES instructions are implemented in AArch32.

0000 No AES instructions implemented.

0001 AESE, AESD, AESMC, and AESIMC implemented.

0010 As for 0001, plus PMULL/PMULL2 instructions operating on 64-bit data quantities.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0010.

**SEVL, bits [3:0]**

Indicates whether the SEVL instruction is implemented in AArch32.

0000 SEVL is implemented as a NOP.

0001 SEVL is implemented as Send Event Local.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

### Accessing the ID_ISAR5_EL1:

To access the ID_ISAR5_EL1:

```
MRS <Xt>, ID_ISAR5_EL1 ; Read ID_ISAR5_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0010 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.54 ID_MMFR0_EL1, AArch32 Memory Model Feature Register 0

The ID_MMFR0_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_MMFR1_EL1, ID_MMFR2_EL1, and ID_MMFR3_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_MMFR0_EL1 is architecturally mapped to AArch32 System register ID_MMFR0.

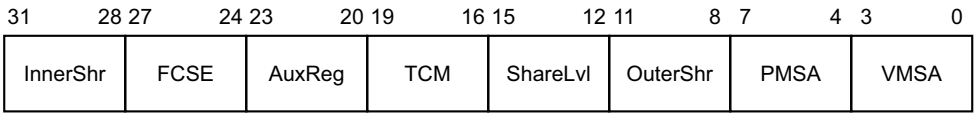
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_MMFR0_EL1 is a 32-bit register.

### Field descriptions

The ID_MMFR0_EL1 bit assignments are:




**InnerShr, bits [31:28]**

Innermost Shareability. Indicates the innermost shareability domain implemented. Defined values are:

`0000` Implemented as Non-cacheable.

`0001` Implemented with hardware coherency support.

1111 Shareability ignored.

All other values are reserved.

In ARMv8-A the permitted values are `0000`, `0001`, and 1111.

This field is valid only if the implementation distinguishes between Inner Shareable and Outer Shareable, by implementing two levels of shareability, as indicated by the value of the Shareability levels field, bits[15:12].

When the Shareability level field is zero, this field is UNK.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**FCSE, bits [27:24]**

Indicates whether the implementation includes the FCSE. Defined values are:

0000 Not supported.

0001 Support for FCSE.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**AuxReg, bits [23:20]**

Auxiliary Registers. Indicates support for Auxiliary registers. Defined values are:

0000 None supported.

0001 Support for Auxiliary Control Register only.

0010 Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

**Note**

Accesses to unimplemented Auxiliary registers are UNDEFINED.

**TCM, bits [19:16]**

Indicates support for TCMs and associated DMAs. Defined values are:

0000 Not supported.

0001 Support is IMPLEMENTATION DEFINED. ARMv7 requires this setting.

0010 Support for TCM only, ARMv6 implementation.

0011 Support for TCM and DMA, ARMv6 implementation.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**ShareLvl, bits [15:12]**

Shareability Levels. Indicates the number of shareability levels implemented. Defined values are:

0000 One level of shareability implemented.

0001 Two levels of shareability implemented.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**OuterShr, bits [11:8]**

Outermost Shareability. Indicates the outermost shareability domain implemented. Defined values are:

0000 Implemented as Non-cacheable.

0001 Implemented with hardware coherency support.

1111 Shareability ignored.

All other values are reserved.

In ARMv8-A the permitted values are 0000, 0001, and 1111.

**PMSA, bits [7:4]**

Indicates support for a PMSA. Defined values are:

0000 Not supported.

0001 Support for IMPLEMENTATION DEFINED PMSA.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`0010` Support for PMSAv6, with a Cache Type Register implemented.

`0011` Support for PMSAv7, with support for memory subsections. ARMv7-R profile.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**VMSA, bits [3:0]**

Indicates support for a VMSA. Defined values are:

`0000` Not supported.

`0001` Support for IMPLEMENTATION DEFINED VMSA.

`0010` Support for VMSAv6, with Cache and TLB Type Registers implemented.

`0011` Support for VMSAv7, with support for remapping and the Access flag. ARMv7-A profile.

`0100` As for `0011`, and adds support for the PXN bit in the Short-descriptor translation table format descriptors.

`0101` As for `0100`, and adds support for the Long-descriptor translation table format.

All other values are reserved.

In ARMv8-A the only permitted value is `0101`.

### Accessing the ID_MMFR0_EL1:

To access the ID_MMFR0_EL1:

```
MRS <Xt>, ID_MMFR0_EL1 ; Read ID_MMFR0_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0001 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.55 ID_MMFR1_EL1, AArch32 Memory Model Feature Register 1

The ID_MMFR1_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_MMFR0_EL1, ID_MMFR2_EL1, and ID_MMFR3_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_MMFR1_EL1 is architecturally mapped to AArch32 System register ID_MMFR1.

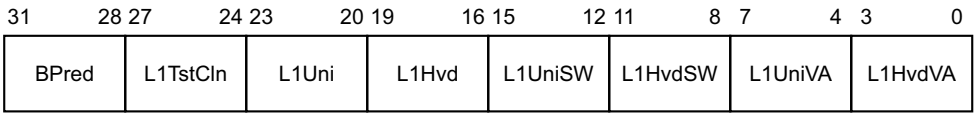
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_MMFR1_EL1 is a 32-bit register.

## Field descriptions

The ID_MMFR1_EL1 bit assignments are:




**BPred, bits [31:28]**

Branch Predictor. Indicates branch predictor management requirements. Defined values are:

`0000` No branch predictor, or no MMU present. Implies a fixed MPU configuration.

`0001` Branch predictor requires flushing on:

- Enabling or disabling a stage of address translation.
- Writing new data to instruction locations.
- Writing new mappings to the translation tables.
- Changes to the TTBR0, TTBR1, or TTBCR registers.
- Changes to the ContextID or ASID, or to the FCSE ProcessID if this is supported.

`0010` Branch predictor requires flushing on:

- Enabling or disabling a stage of address translation.
- Writing new data to instruction locations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Writing new mappings to the translation tables.
- Any change to the TTBR0, TTBR1, or TTBCR registers without a change to the corresponding ContextID or ASID, or FCSE ProcessID if this is supported.

`0011` Branch predictor requires flushing only on writing new data to instruction locations.

`0100` For execution correctness, branch predictor requires no flushing at any time.

All other values are reserved.

In ARMv8-A the permitted values are `0010`, `0011`, or `0100`. For values other than `0000` and `0100` the ARM Architecture Reference Manual, or the product documentation, might give more information about the required maintenance.

**L1TstCln, bits [27:24]**

Level 1 cache Test and Clean. Indicates the supported Level 1 data cache test and clean operations, for Harvard or unified cache implementations. Defined values are:

`0000` None supported.

`0001` Supported Level 1 data cache test and clean operations are:

- Test and clean data cache.

`0010` As for `0001`, and adds:

- Test, clean, and invalidate data cache.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**L1Uni, bits [23:20]**

Level 1 Unified cache. Indicates the supported entire Level 1 cache maintenance operations, for a unified cache implementation. Defined values are:

`0000` None supported.

`0001` Supported entire Level 1 cache operations are:

- Invalidate cache, including branch predictor if appropriate.
- Invalidate branch predictor, if appropriate.

`0010` As for `0001`, and adds:

- Clean cache, using a recursive model that uses the cache dirty status bit.
- Clean and invalidate cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**L1Hvd, bits [19:16]**

Level 1 Harvard cache. Indicates the supported entire Level 1 cache maintenance operations, for a Harvard cache implementation. Defined values are:

`0000` None supported.

`0001` Supported entire Level 1 cache operations are:

- Invalidate instruction cache, including branch predictor if appropriate.
- Invalidate branch predictor, if appropriate.

`0010` As for `0001`, and adds:

- Invalidate data cache.
- Invalidate data cache and instruction cache, including branch predictor if appropriate.

`0011` As for `0010`, and adds:

- Clean data cache, using a recursive model that uses the cache dirty status bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Clean and invalidate data cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**L1UniSW, bits [15:12]**

Level 1 Unified cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a unified cache implementation. Defined values are:

0000 None supported.

0001 Supported Level 1 unified cache line maintenance operations by set/way are:

- Clean cache line by set/way.

0010 As for 0001, and adds:

- Clean and invalidate cache line by set/way.

0011 As for 0010, and adds:

- Invalidate cache line by set/way.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**L1HvdSW, bits [11:8]**

Level 1 Harvard cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a Harvard cache implementation. Defined values are:

0000 None supported.

0001 Supported Level 1 Harvard cache line maintenance operations by set/way are:

- Clean data cache line by set/way.
- Clean and invalidate data cache line by set/way.

0010 As for 0001, and adds:

- Invalidate data cache line by set/way.

0011 As for 0010, and adds:

- Invalidate instruction cache line by set/way.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**L1UniVA, bits [7:4]**

Level 1 Unified cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a unified cache implementation. Defined values are:

0000 None supported.

0001 Supported Level 1 unified cache line maintenance operations by VA are:

- Clean cache line by VA.
- Invalidate cache line by VA.
- Clean and invalidate cache line by VA.

0010 As for 0001, and adds:

- Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**L1HvdVA, bits [3:0]**

Level 1 Harvard cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a Harvard cache implementation. Defined values are:

`0000` None supported.

`0001` Supported Level 1 Harvard cache line maintenance operations by VA are:

- Clean data cache line by VA.
- Invalidate data cache line by VA.
- Clean and invalidate data cache line by VA.
- Clean instruction cache line by VA.

`0010` As for `0001`, and adds:

- Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

## Accessing the ID_MMFR1_EL1:

To access the ID_MMFR1_EL1:

```
MRS <Xt>, ID_MMFR1_EL1 ; Read ID_MMFR1_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0001 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.56 ID_MMFR2_EL1, AArch32 Memory Model Feature Register 2

The ID_MMFR2_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_MMFR0_EL1, ID_MMFR1_EL1, and ID_MMFR3_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_MMFR2_EL1 is architecturally mapped to AArch32 System register ID_MMFR2.

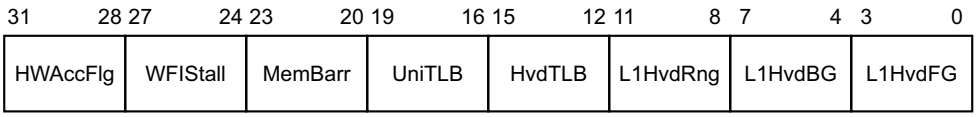
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_MMFR2_EL1 is a 32-bit register.

#### Field descriptions

The ID_MMFR2_EL1 bit assignments are:




**HWAccFlg, bits [31:28]**

Hardware Access Flag. In earlier versions of the ARM Architecture, this field indicates support for a Hardware Access flag, as part of the VMSAv7 implementation. Defined values are:

`0000` Not supported.

`0001` Support for VMSAv7 Access flag, updated in hardware.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**WFIStall, bits [27:24]**

Wait For Interrupt Stall. Indicates the support for Wait For Interrupt (WFI) stalling. Defined values are:

`0000` Not supported.

`0001` Support for WFI stalling.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

**MemBarr, bits [23:20]**

Memory Barrier. Indicates the supported CP15 memory barrier operations:

0000 None supported.

0001 Supported CP15 Memory barrier operations are:

- Data Synchronization Barrier (DSB), which in previous versions of the ARM architecture was named Data Write Barrier (DWB).

0010 As for 0001, and adds:

- Instruction Synchronization Barrier (ISB), which in previous versions of the ARM architecture was called Prefetch Flush.
- Data Memory Barrier (DMB).

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

ARM deprecates the use of these operations. ID_ISAR4.Barrier_instrs indicates the level of support for the preferred barrier instructions.

**UniTLB, bits [19:16]**

Unified TLB. Indicates the supported TLB maintenance operations, for a unified TLB implementation. Defined values are:

0000 Not supported.

0001 Supported unified TLB maintenance operations are:

- Invalidate all entries in the TLB.
- Invalidate TLB entry by VA.

0010 As for 0001, and adds:

- Invalidate TLB entries by ASID match.

0011 As for 0010, and adds:

- Invalidate instruction TLB and data TLB entries by VA All ASID. This is a shared unified TLB operation.

0100 As for 0011, and adds:

- Invalidate Hyp mode unified TLB entry by VA.
- Invalidate entire Non-secure PL1&0 unified TLB.
- Invalidate entire Hyp mode unified TLB.

0101 As for 0100, and adds the following operations: TLBIMVALIS, TLBIMVAALIS, TLBIMVALHIS, TLBIMVAL, TLBIMVAAL, TLBIMVALH.

0110 As for 0101, and adds the following operations: TLBIIPAS2IS, TLBIIPAS2LIS, TLBIIPAS2, TLBIIPAS2L.

All other values are reserved.

In ARMv8-A the only permitted value is 0110.

**HvdTLB, bits [15:12]**

If the Unified TLB field (UniTLB, bits [19:16]) is not 0000, then the meaning of this field is IMPLEMENTATION DEFINED. ARM deprecates the use of this field by software.

**L1HvdRng, bits [11:8]**

Level 1 Harvard cache Range. Indicates the supported Level 1 cache maintenance range operations, for a Harvard cache implementation. Defined values are:

0000 Not supported.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0001 Supported Level 1 Harvard cache maintenance range operations are:

- Invalidate data cache range by VA.
- Invalidate instruction cache range by VA.
- Clean data cache range by VA.
- Clean and invalidate data cache range by VA.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**L1HvdBG, bits [7:4]**

Level 1 Harvard cache Background fetch. Indicates the supported Level 1 cache background fetch operations, for a Harvard cache implementation. When supported, background fetch operations are non-blocking operations. Defined values are:

0000 Not supported.

0001 Supported Level 1 Harvard cache background fetch operations are:

- Fetch instruction cache range by VA.
- Fetch data cache range by VA.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**L1HvdFG, bits [3:0]**

Level 1 Harvard cache Foreground fetch. Indicates the supported Level 1 cache foreground fetch operations, for a Harvard cache implementation. When supported, foreground fetch operations are blocking operations. Defined values are:

0000 Not supported.

0001 Supported Level 1 Harvard cache foreground fetch operations are:

- Fetch instruction cache range by VA.
- Fetch data cache range by VA.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

## Accessing the ID_MMFR2_EL1:

To access the ID_MMFR2_EL1:

```
MRS <Xt>, ID_MMFR2_EL1 ; Read ID_MMFR2_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0001 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.57 ID_MMFR3_EL1, AArch32 Memory Model Feature Register 3

The ID_MMFR3_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_MMFR0_EL1, ID_MMFR1_EL1, and ID_MMFR2_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_MMFR3_EL1 is architecturally mapped to AArch32 System register ID_MMFR3.

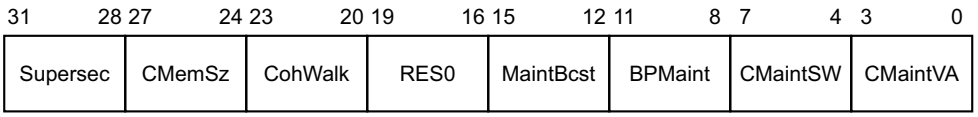
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_MMFR3_EL1 is a 32-bit register.

### Field descriptions

The ID_MMFR3_EL1 bit assignments are:




**Supersec, bits [31:28]**

Supersections. On a VMSA implementation, indicates whether Supersections are supported. Defined values are:

`0000` Supersections supported.

1111 Supersections not supported.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and 1111.

**CMemSz, bits [27:24]**

Cached Memory Size. Indicates the physical memory size supported by the caches. Defined values are:

`0000` 4GB, corresponding to a 32-bit physical address range.

`0001` 64GB, corresponding to a 36-bit physical address range.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0010 1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In ARMv8-A the permitted values are 0000, 0001, and 0010.

**CohWalk, bits [23:20]**

Coherent Walk. Indicates whether Translation table updates require a clean to the point of unification. Defined values are:

0000 Updates to the translation tables require a clean to the point of unification to ensure visibility by subsequent translation table walks.

0001 Updates to the translation tables do not require a clean to the point of unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**Bits [19:16]**

Reserved, RES0.

**MaintBcst, bits [15:12]**

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast. Defined values are:

0000 Cache, TLB, and branch predictor operations only affect local structures.

0001 Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.

0010 Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

**BPMaint, bits [11:8]**

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations, Defined values are:

0000 None supported.

0001 Supported branch predictor maintenance operations are:

- Invalidate all branch predictors.

0010 As for 0001, and adds:

- Invalidate branch predictors by VA.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

**CMaintSW, bits [7:4]**

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches. Defined values are:

0000 None supported.

0001 Supported hierarchical cache maintenance instructions by set/way are:

- Invalidate data cache by set/way.
- Clean data cache by set/way.
- Clean and invalidate data cache by set/way.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a unified cache implementation, the data cache maintenance instructions apply to the unified caches.

**CMaintVA, bits [3:0]**

Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches. Defined values are:

`0000` None supported.

`0001` Supported hierarchical cache maintenance instructions by VA are:

- Invalidate data cache by VA.
- Clean data cache by VA.
- Clean and invalidate data cache by VA.
- Invalidate instruction cache by VA.
- Invalidate all instruction cache entries.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

In a unified cache implementation, data cache maintenance instructions apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

## Accessing the ID_MMFR3_EL1:

To access the ID_MMFR3_EL1:

```
MRS <Xt>, ID_MMFR3_EL1 ; Read ID_MMFR3_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0001 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.58 ID_MMFR4_EL1, AArch32 Memory Model Feature Register 4

The ID_MMFR4_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1 then:

- If the register is not RAZ/WI then Non-secure EL1 read accesses to this register are trapped to EL2.
- Otherwise, it is IMPLEMENTATION DEFINED whether Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_MMFR4_EL1 is architecturally mapped to AArch32 System register ID_MMFR4.

In an implementation that does not include ACTLR2 and HACTLR2 this register is RAZ/WI.

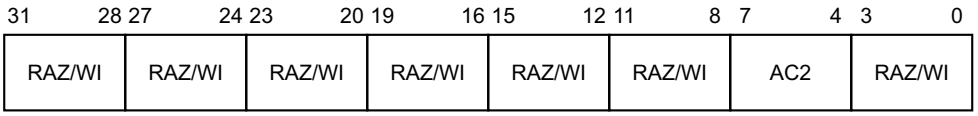
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_MMFR4_EL1 is a 32-bit register.

## Field descriptions

The ID_MMFR4_EL1 bit assignments are:




**AC2, bits [7:4]**

Indicates the extension of the ACTLR and HACTLR registers using ACTLR2 and HACTLR2. Defined values are:

`0000` ACTLR2 and HACTLR2 are not implemented.

`0001` ACTLR2 and HACTLR2 are implemented.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and `0001`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the ID_MMFR4_EL1:

To access the ID_MMFR4_EL1:

```
MRS <Xt>, ID_MMFR4_EL1 ; Read ID_MMFR4_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0010 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.59 ID_PFR0_EL1, AArch32 Processor Feature Register 0

The ID_PFR0_EL1 characteristics are:

**Purpose**

Gives top-level information about the instruction sets supported by the PE in AArch32.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_PFR1_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_PFR0_EL1 is architecturally mapped to AArch32 System register ID_PFR0.

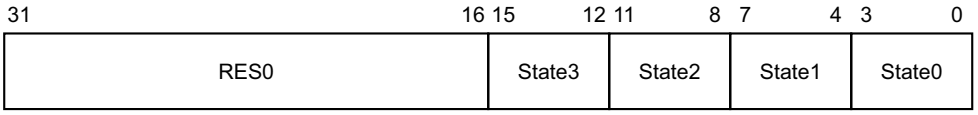
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_PFR0_EL1 is a 32-bit register.

#### Field descriptions

The ID_PFR0_EL1 bit assignments are:




**Bits [31:16]**

Reserved, RES0.

**State3, bits [15:12]**

T32EE instruction set support. Defined values are:

`0000` Not implemented.

`0001` T32EE instruction set implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**State2, bits [11:8]**

Jazelle extension support. Defined values are:

`0000` Not implemented.

`0001` Jazelle extension implemented, without clearing of JOSCR.CV on exception entry.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0010 Jazelle extension implemented, with clearing of JOSCR.CV on exception entry.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**State1, bits [7:4]**

T32 instruction set support. Defined values are:

0000 T32 instruction set not implemented.

0001 T32 encodings before the introduction of Thumb-2 technology implemented:

- All instructions are 16-bit.
- A BL or BLX is a pair of 16-bit instructions.
- 32-bit instructions other than BL and BLX cannot be encoded.

0011 T32 encodings after the introduction of Thumb-2 technology implemented, for all 16-bit and 32-bit T32 basic instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0011.

**State0, bits [3:0]**

A32 instruction set support. Defined values are:

0000 A32 instruction set not implemented.

0001 A32 instruction set implemented.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

### Accessing the ID_PFR0_EL1:

To access the ID_PFR0_EL1:

```
MRS <Xt>, ID_PFR0_EL1 ; Read ID_PFR0_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.60 ID_PFR1_EL1, AArch32 Processor Feature Register 1

The ID_PFR1_EL1 characteristics are:

**Purpose**

Gives information about the AArch32 programmers' model.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

Must be interpreted with ID_PFR0_EL1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register ID_PFR1_EL1 is architecturally mapped to AArch32 System register ID_PFR1.

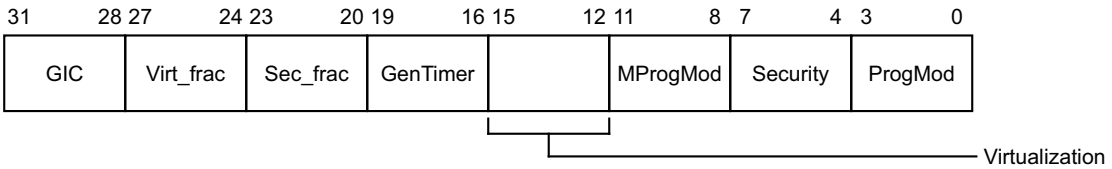
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

ID_PFR1_EL1 is a 32-bit register.

### Field descriptions

The ID_PFR1_EL1 bit assignments are:




**GIC, bits [31:28]**

System register GIC CPU interface. Defined values are:

`0000` No System register interface to the GIC CPU interface is supported.

`0001` System register interface to the GIC CPU interface is supported.

All other values are reserved.

**Virt_frac, bits [27:24]**

Virtualization fractional field. When the Virtualization field is `0000`, determines the support for features from the ARMv7 Virtualization Extensions. Defined values are:

`0000` No features from the ARMv7 Virtualization Extensions are implemented.

`0001` The following features of the ARMv7 Virtualization Extensions are implemented:

- The SCR.SIF bit, if EL3 is implemented.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The modifications to the SCR.AW and SCR.FW bits described in the Virtualization Extensions, if EL3 is implemented.
- The MSR (Banked register) and MRS (Banked register) instructions.
- The ERET instruction.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**Note**

The ID_ISAR registers do not identify whether the instructions added by the Virtualization Extensions are implemented.

This field is only valid when ID_PFR1_EL1[15:12] == 0, otherwise it holds the value 0000.

**Sec_frac, bits [23:20]**

Security fractional field. When the Security field is 0000, determines the support for features from the ARMv7 Security Extensions. Defined values are:

0000 No features from the ARMv7 Security Extensions are implemented.

0001 The following features from the ARMv7 Security Extensions are implemented:

- The VBAR register.
- The TTBCR.PD0 and TTBCR.PD1 bits.

0010 As for 0001, plus the ability to access Secure or Non-secure physical memory is supported.

All other values are reserved.

In ARMv8-A the permitted values are 0000, 0001, and 0010.

This field is only valid when ID_PFR1_EL1[7:4] == 0, otherwise it holds the value 0000.

**GenTimer, bits [19:16]**

Generic Timer support. Defined values are:

0000 Not implemented.

0001 Generic Timer implemented.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**Virtualization, bits [15:12]**

Virtualization support. Defined values are:

0000 EL2, Hyp mode, and the HVC instruction not implemented.

0001 EL2, Hyp mode, the HVC instruction, and all the features described by Virt_frac == 0001 implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

**Note**

The ID_ISARs do not identify whether the HVC instruction is implemented.

**MProgMod, bits [11:8]**

M profile programmers' model support. Defined values are:

0000 Not supported.

0010 Support for two-stack programmers' model.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In ARMv8-A the only permitted value is 0000.

**Security, bits [7:4]**

Security support. Defined values are:

0000 EL3, Monitor mode, and the SMC instruction not implemented.

0001 EL3, Monitor mode, the SMC instruction, and all the features described by Sec_frac == 0001 implemented.

0010 As for 0001, and adds the ability to set the NSACR.RFR bit. Not permitted in ARMv8 as the NSACR.RFR bit is RES0.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

**ProgMod, bits [3:0]**

Support for the standard programmers' model for ARMv4 and later. Model must support User, FIQ, IRQ, Supervisor, Abort, Undefined, and System modes. Defined values are:

0000 Not supported.

0001 Supported.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

## Accessing the ID_PFR1_EL1:

To access the ID_PFR1_EL1:

```
MRS <Xt>, ID_PFR1_EL1 ; Read ID_PFR1_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.61 IFSR32_EL2, Instruction Fault Status Register (EL2)

The IFSR32_EL2 characteristics are:

**Purpose**

Allows access to the AArch32 IFSR register from AArch64 state only. Its value has no effect on execution in AArch64 state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register IFSR32_EL2 is architecturally mapped to AArch32 System register IFSR.

If EL1 is AArch64 only, this register is UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

RW fields in this register reset to architecturally UNKNOWN values.

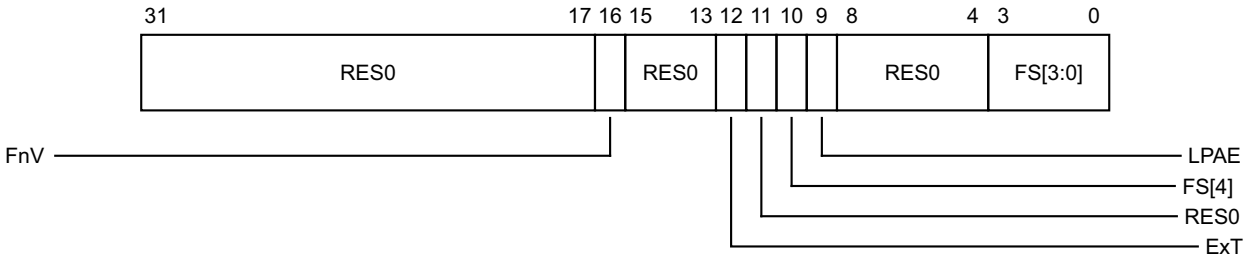
**Attributes**

IFSR32_EL2 is a 32-bit register.

### Field descriptions

The IFSR32_EL2 bit assignments are:

***When TTBCR.EAE==0:***




**Bits [31:17]**

Reserved, RES0.

**FnV, bit [16]**

FAR not Valid, for a Synchronous external abort other than a Synchronous external abort on a translation table walk.

0 IFAR is valid.

1 IFAR is not valid, and holds an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is only valid for a Synchronous external abort other than a Synchronous external abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

**Bits [15:13]**

Reserved, RES0.

**ExT, bit [12]**

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

**Bit [11]**

Reserved, RES0.

**FS[4], bit [10]**

See FS[3:0], bits [3:0] for description of the FS field.

**LPAE, bit [9]**

On taking a Data Abort exception, this bit is set as follows:

`0` Using the Short-descriptor translation table formats.

`1` Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

**Bits [8:4]**

Reserved, RES0.

**FS[3:0], bits [3:0]**

Fault status bits. Interpreted with bit [10]. Possible values of FS[4:0] are:

`00001` PC alignment fault

`00010` Debug event

`00011` Access flag fault, level 1

`00101` Translation fault, level 1

`00110` Access flag fault, level 2

`00111` Translation fault, level 2

`01000` Synchronous external abort, other than synchronous parity or ECC error, not on translation table walk

`01001` Domain fault, level 1

`01011` Domain fault, level 2

`01100` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 1

`01101` Permission fault, level 1

`01110` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 2

`01111` Permission fault, level 2

`10000` TLB conflict abort

`10100` IMPLEMENTATION DEFINED fault (Lockdown fault)

`11001` Synchronous parity or ECC error on memory access, not on translation table walk

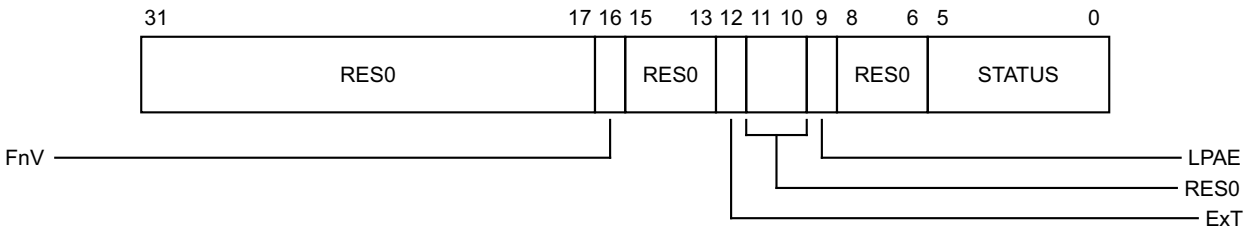
`11100` Synchronous parity or ECC error on translation table walk, level 1

`11110` Synchronous parity or ECC error on translation table walk, level 2


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

***When TTBCR.EAE==1:***




**Bits [31:17]**

Reserved, RES0.

**FnV, bit [16]**

FAR not Valid, for a Synchronous external abort other than a Synchronous external abort on a translation table walk.

`0` IFAR is valid.

`1` IFAR is not valid, and holds an UNKNOWN value.

This field is only valid for a Synchronous external abort other than a Synchronous external abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

**Bits [15:13]**

Reserved, RES0.

**ExT, bit [12]**

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

**Bits [11:10]**

Reserved, RES0.

**LPAE, bit [9]**

On taking a Data Abort exception, this bit is set as follows:

`0` Using the Short-descriptor translation table formats.

`1` Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

**Bits [8:6]**

Reserved, RES0.

**STATUS, bits [5:0]**

Fault status bits. All encodings not shown below are reserved:

`000000` Address size fault in TTBR0 or TTBR1

`000001` Address size fault, level 1

`000010` Address size fault, level 2

`000011` Address size fault, level 3

`000101` Translation fault, level 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`000110` Translation fault, level 2

`000111` Translation fault, level 3

`001001` Access flag fault, level 1

`001010` Access flag fault, level 2

`001011` Access flag fault, level 3

`001101` Permission fault, level 1

`001110` Permission fault, level 2

`001111` Permission fault, level 3

`010000` Synchronous external abort, other than synchronous parity or ECC error, not on translation table walk

`010101` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 1

`010110` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 2

`010111` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 3

`011000` Synchronous parity or ECC error on memory access, not on translation table walk

`011101` Synchronous parity or ECC error on memory access on translation table walk, level 1

`011110` Synchronous parity or ECC error on memory access on translation table walk, level 2

`011111` Synchronous parity or ECC error on memory access on translation table walk, level 3

`100001` Alignment fault

`100010` Debug event

`110000` TLB conflict abort

All other values are reserved.

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a fault at level 1.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

### Accessing the IFSR32_EL2:

To access the IFSR32_EL2:

```
MRS <Xt>, IFSR32_EL2 ; Read IFSR32_EL2 into Xt
MSR IFSR32_EL2, <Xt> ; Write Xt to IFSR32_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0101 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.62 ISR_EL1, Interrupt Status Register

The ISR_EL1 characteristics are:

**Purpose**

Shows whether an IRQ, FIQ, or SError interrupt is pending. When accessed from EL3, EL2 or Secure EL1, the indicated interrupt is the physical interrupt. When accessed from Non-secure EL1, if the interrupt is virtualized, the indicated interrupt is virtual, otherwise it is physical.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

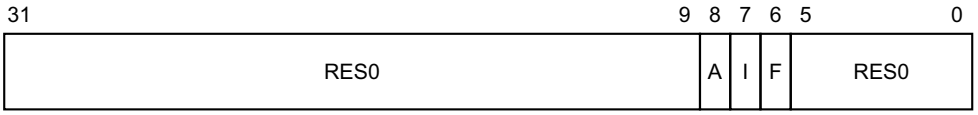
AArch64 System register ISR_EL1 is architecturally mapped to AArch32 System register ISR.

**Attributes**

ISR_EL1 is a 32-bit register.

#### Field descriptions

The ISR_EL1 bit assignments are:




**Bits [31:9]**

Reserved, RES0.

**A, bit [8]**

SError interrupt pending bit:

0 No pending SError.

1 An SError interrupt is pending.

**I, bit [7]**

IRQ pending bit. Indicates whether an IRQ interrupt is pending:

0 No pending IRQ.

1 An IRQ interrupt is pending.

**F, bit [6]**

FIQ pending bit. Indicates whether an FIQ interrupt is pending.

0 No pending FIQ.

1 An FIQ interrupt is pending.

**Bits [5:0]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the ISR_EL1:

To access the ISR_EL1:

```
MRS <Xt>, ISR_EL1 ; Read ISR_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1100 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.63 MAIR_EL1, Memory Attribute Indirection Register (EL1)

The MAIR_EL1 characteristics are:

**Purpose**

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

MAIR_EL1 is permitted to be cached in a TLB.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register MAIR_EL1[31:0] is architecturally mapped to AArch32 System register PRRR when TTBCR.EAE==0.

AArch64 System register MAIR_EL1[31:0] is architecturally mapped to AArch32 System register MAIR0 when TTBCR.EAE==1.

AArch64 System register MAIR_EL1[63:32] is architecturally mapped to AArch32 System register NMRR when TTBCR.EAE==0.

AArch64 System register MAIR_EL1[63:32] is architecturally mapped to AArch32 System register MAIR1 when TTBCR.EAE==1.

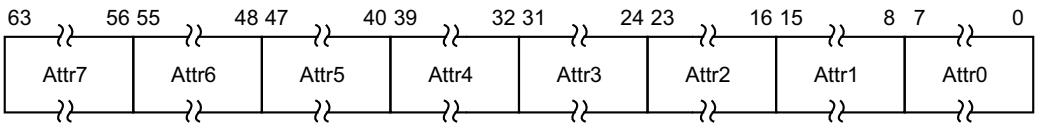
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

MAIR_EL1 is a 64-bit register.

### Field descriptions

The MAIR_EL1 bit assignments are:




**Attr<n>, bits [8n+7:8n], for n = 0 to 7**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where AttrIndx[2:0] gives the value of <n> in Attr<n>.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [7:4] are encoded as follows:

| Attr<n>[7:4] | Meaning |
|---|---|
| 0000 | Device memory. See encoding of Attr<n>[3:0] for the type of Device memory. |
| 00RW, RW not 00 | Normal Memory, Outer Write-through transient |
| 0100 | Normal Memory, Outer Non-cacheable |
| 01RW, RW not 00 | Normal Memory, Outer Write-back transient |
| 10RW | Normal Memory, Outer Write-through non-transient |
| 11RW | Normal Memory, Outer Write-back non-transient |

R = Outer Read Allocate Policy, W = Outer Write Allocate Policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

| Attr<n>[3:0] | Meaning when Attr<n>[7:4] is 0000 | Meaning when Attr<n>[7:4] is not 0000 |
|---|---|---|
| 0000 | Device-nGnRnE memory | UNPREDICTABLE |
| 00RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through transient |
| 0100 | Device-nGnRE memory | Normal memory, Inner Non-cacheable |
| 01RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back transient |
| 1000 | Device-nGRE memory | Normal Memory, Inner Write-through non-transient (RW=00) |
| 10RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through non-transient |
| 1100 | Device-GRE memory | Normal Memory, Inner Write-back non-transient (RW=00) |
| 11RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back non-transient |

R = Inner Read Allocate Policy, W = Inner Write Allocate Policy.

The R and W bits in some Attr<n> fields have the following meanings:

| R or W | Meaning |
|---|---|
| 0 | Do not allocate |
| 1 | Allocate |

### Accessing the MAIR_EL1:

To access the MAIR_EL1:

```
MRS <Xt>, MAIR_EL1 ; Read MAIR_EL1 into Xt
MSR MAIR_EL1, <Xt> ; Write Xt to MAIR_EL1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1010 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.64 MAIR_EL2, Memory Attribute Indirection Register (EL2)

The MAIR_EL2 characteristics are:

**Purpose**

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

MAIR_EL2 is permitted to be cached in a TLB.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register MAIR_EL2[31:0] is architecturally mapped to AArch32 System register HMAIR0.

AArch64 System register MAIR_EL2[63:32] is architecturally mapped to AArch32 System register HMAIR1.

If EL2 is not implemented, this register is RES0 from EL3.

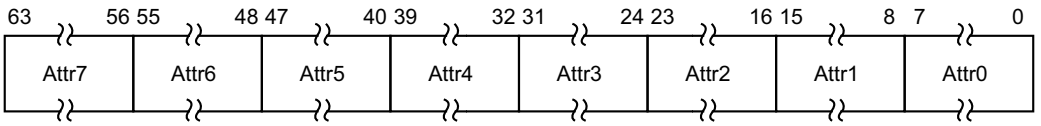
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

MAIR_EL2 is a 64-bit register.

#### Field descriptions

The MAIR_EL2 bit assignments are:




**Attr<n>, bits [8n+7:8n], for n = 0 to 7**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where AttrIndx[2:0] gives the value of <n> in Attr<n>.

Bits [7:4] are encoded as follows:

| Attr<n>[7:4] | Meaning |
|---|---|
| 0000 | Device memory. See encoding of Attr<n>[3:0] for the type of Device memory. |
| 00RW, RW not 00 | Normal Memory, Outer Write-through transient |
| 0100 | Normal Memory, Outer Non-cacheable |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Attr<n>[7:4] | Meaning |
|---|---|
| 01RW, RW not 00 | Normal Memory, Outer Write-back transient |
| 10RW | Normal Memory, Outer Write-through non-transient |
| 11RW | Normal Memory, Outer Write-back non-transient |

R = Outer Read Allocate Policy, W = Outer Write Allocate Policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

| Attr<n>[3:0] | Meaning when Attr<n>[7:4] is 0000 | Meaning when Attr<n>[7:4] is not 0000 |
|---|---|---|
| 0000 | Device-nGnRnE memory | UNPREDICTABLE |
| 00RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through transient |
| 0100 | Device-nGnRE memory | Normal memory, Inner Non-cacheable |
| 01RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back transient |
| 1000 | Device-nGRE memory | Normal Memory, Inner Write-through non-transient (RW=00) |
| 10RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through non-transient |
| 1100 | Device-GRE memory | Normal Memory, Inner Write-back non-transient (RW=00) |
| 11RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back non-transient |

R = Inner Read Allocate Policy, W = Inner Write Allocate Policy.

The R and W bits in some Attr<n> fields have the following meanings:

| R or W | Meaning |
|---|---|
| 0 | Do not allocate |
| 1 | Allocate |

### Accessing the MAIR_EL2:

To access the MAIR_EL2:

```
MRS <Xt>, MAIR_EL2 ; Read MAIR_EL2 into Xt
MSR MAIR_EL2, <Xt> ; Write Xt to MAIR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1010 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.65 MAIR_EL3, Memory Attribute Indirection Register (EL3)

The MAIR_EL3 characteristics are:

**Purpose**

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

MAIR_EL3 is permitted to be cached in a TLB.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

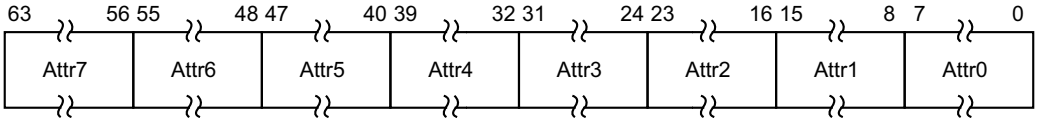
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

MAIR_EL3 is a 64-bit register.

#### Field descriptions

The MAIR_EL3 bit assignments are:




**Attr<n>, bits [8n+7:8n], for n = 0 to 7**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where AttrIndx[2:0] gives the value of <n> in Attr<n>.

Bits [7:4] are encoded as follows:

| Attr<n>[7:4] | Meaning |
|---|---|
| 0000 | Device memory. See encoding of Attr<n>[3:0] for the type of Device memory. |
| 00RW, RW not 00 | Normal Memory, Outer Write-through transient |
| 0100 | Normal Memory, Outer Non-cacheable |
| 01RW, RW not 00 | Normal Memory, Outer Write-back transient |
| 10RW | Normal Memory, Outer Write-through non-transient |
| 11RW | Normal Memory, Outer Write-back non-transient |

R = Outer Read Allocate Policy, W = Outer Write Allocate Policy.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The meaning of bits [3:0] depends on the value of bits [7:4]:

| Attr<n>[3:0] | Meaning when Attr<n>[7:4] is 0000 | Meaning when Attr<n>[7:4] is not 0000 |
|---|---|---|
| 0000 | Device-nGnRnE memory | UNPREDICTABLE |
| 00RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through transient |
| 0100 | Device-nGnRE memory | Normal memory, Inner Non-cacheable |
| 01RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back transient |
| 1000 | Device-nGRE memory | Normal Memory, Inner Write-through non-transient (RW=00) |
| 10RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through non-transient |
| 1100 | Device-GRE memory | Normal Memory, Inner Write-back non-transient (RW=00) |
| 11RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back non-transient |

R = Inner Read Allocate Policy, W = Inner Write Allocate Policy.

The R and W bits in some Attr<n> fields have the following meanings:

| R or W | Meaning |
|---|---|
| 0 | Do not allocate |
| 1 | Allocate |

### Accessing the MAIR_EL3:

To access the MAIR_EL3:

```
MRS <Xt>, MAIR_EL3 ; Read MAIR_EL3 into Xt
MSR MAIR_EL3, <Xt> ; Write Xt to MAIR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 1010 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.66 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

**Purpose**

Provides identification information for the PE, including an implementer code for the device and a device ID number.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register MIDR_EL1 is architecturally mapped to AArch32 System register MIDR.

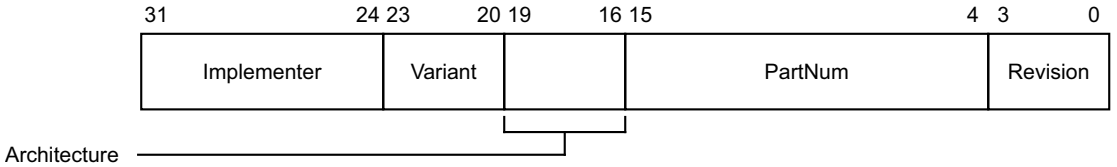
AArch64 System register MIDR_EL1 is architecturally mapped to External register MIDR_EL1.

**Attributes**

MIDR_EL1 is a 32-bit register.

#### Field descriptions

The MIDR_EL1 bit assignments are:




**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by ARM. Assigned codes include the following:

| Hex representation | ASCII representation | Implementer |
|---|---|---|
| 0x41 | A | ARM Limited |
| 0x42 | B | Broadcom Corporation |
| 0x43 | C | Cavium Inc. |
| 0x44 | D | Digital Equipment Corporation |
| 0x49 | I | Infineon Technologies AG |
| 0x4D | M | Motorola or Freescale Semiconductor Inc. |
| 0x4E | N | NVIDIA Corporation |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Hex representation | ASCII representation | Implementer |
|---|---|---|
| 0x50 | P | Applied Micro Circuits Corporation |
| 0x51 | Q | Qualcomm Inc. |
| 0x56 | V | Marvell International Ltd. |
| 0x69 | i | Intel Corporation |

ARM can assign codes that are not published in this manual. All values not assigned by ARM are reserved and must not be used.

**Variant, bits [23:20]**

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

**Architecture, bits [19:16]**

The permitted values of this field are:

0001 ARMv4

0010 ARMv4T

0011 ARMv5 (obsolete)

0100 ARMv5T

0101 ARMv5TE

0110 ARMv5TEJ

0111 ARMv6

1111 Architectural features are individually identified in the ID_* registers, see *Identification registers, functional group* on page G4-4162.

All other values are reserved.

**PartNum, bits [15:4]**

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by ARM, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

**Revision, bits [3:0]**

An IMPLEMENTATION DEFINED revision number for the device.

## Accessing the MIDR_EL1:

To access the MIDR_EL1:

```
MRS <Xt>, MIDR_EL1 ; Read MIDR_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.67 MPIDR_EL1, Multiprocessor Affinity Register

The MPIDR_EL1 characteristics are:

**Purpose**

In a multiprocessor system, provides an additional PE identification mechanism for scheduling purposes.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register MPIDR_EL1 is architecturally mapped to AArch32 System register MPIDR.

The assigned value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

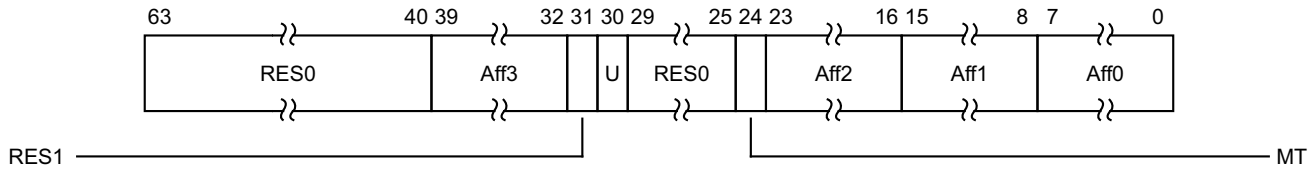
In a uniprocessor system ARM recommends that each Aff<n> field of this register returns a value of 0.

**Attributes**

MPIDR_EL1 is a 64-bit register.

## Field descriptions

The MPIDR_EL1 bit assignments are:




**Bits [63:40]**

Reserved, RES0.

**Aff3, bits [39:32]**

Affinity level 3. Highest level affinity field.

**Bit [31]**

Reserved, RES1.

**U, bit [30]**

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system. The possible values of this bit are:

`0` Processor is part of a multiprocessor system.

`1` Processor is part of a uniprocessor system.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [29:25]**

Reserved, RES0.

**MT, bit [24]**

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach. The possible values of this bit are:

0 Performance of PEs at the lowest affinity level is largely independent.

1 Performance of PEs at the lowest affinity level is very interdependent.

**Aff2, bits [23:16]**

Affinity level 2. Second highest level affinity field.

**Aff1, bits [15:8]**

Affinity level 1. Third highest level affinity field.

**Aff0, bits [7:0]**

Affinity level 0. Lowest level affinity field.

### Accessing the MPIDR_EL1:

To access the MPIDR_EL1:

```
MRS <Xt>, MPIDR_EL1 ; Read MPIDR_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0000 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.68 MVFR0_EL1, AArch32 Media and VFP Feature Register 0

The MVFR0_EL1 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register MVFR0_EL1 is architecturally mapped to AArch32 System register MVFR0.

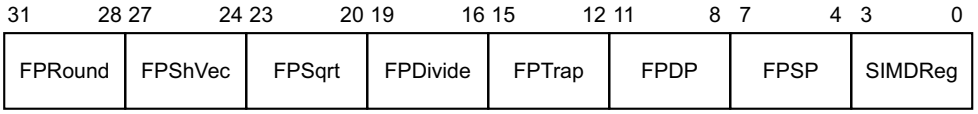
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

MVFR0_EL1 is a 32-bit register.

#### Field descriptions

The MVFR0_EL1 bit assignments are:




**FPRound, bits [31:28]**

Floating-Point Rounding modes. Indicates the rounding modes supported by the VFP floating-point hardware. Defined values are:

`0000` Only Round to Nearest mode supported, except that Round towards Zero mode is supported for VCVT instructions that always use that rounding mode regardless of the FPSCR setting.

`0001` All rounding modes supported.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**FPShVec, bits [27:24]**

Short Vectors. Indicates that the floating-point instructions support the use of short vectors. Defined values are:

`0000` Short vectors not supported.

`0001` Short vector operation supported.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**FPSqrt, bits [23:20]**

Square Root. Indicates the hardware support for the ARMv6 VFP square root operations. Defined values are:

0000 Not supported in hardware.

0001 Supported.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

The VSQRT.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VSQRT.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

**FPDivide, bits [19:16]**

Indicates the hardware support for VFP divide operations. Defined values are:

0000 Not supported in hardware.

0001 Supported.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

The VDIV.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VDIV.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

**FPTrap, bits [15:12]**

Floating Point Exception Trapping. Indicates whether the VFP hardware implementation supports exception trapping. Defined values are:

0000 Not supported.

0001 Supported by the hardware. When exception trapping is supported, support code is needed to handle the trapped exceptions.

All other values are reserved.

A value of 0001 does not indicate that trapped exception handling is available. Because trapped exception handling requires support code, only the support code can provide this information.

**FPDP, bits [11:8]**

Double Precision. Indicates the hardware support for VFP double-precision operations. Defined values are:

0000 Not supported in hardware.

0001 Supported, VFPv2.

0010 Supported, VFPv3, VFPv4, or ARMv8. VFPv3 and ARMv8 add an instruction to load a double-precision floating-point constant, and conversions between double-precision and fixed-point values.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

A value of 0b0001 or 0b0010 indicates support for all VFP double-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F64 is only available if the Square root field is 0001.
- VDIV.F64 is only available if the Divide field is 0001.
- Conversion between double-precision and single-precision is only available if the single-precision field is nonzero.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**FPSP, bits [7:4]**

Single Precision. Indicates the hardware support for VFP single-precision operations. Defined values are:

`0000` Not supported in hardware.

`0001` Supported, VFPv2.

`0010` Supported, VFPv3 or VFPv4. VFPv3 adds an instruction to load a single-precision floating-point constant, and conversions between single-precision and fixed-point values.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

A value of `0b0001` or `0b0010` indicates support for all VFP single-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F32 is only available if the Square root field is `0001`.
- VDIV.F32 is only available if the Divide field is `0001`.
- Conversion between double-precision and single-precision is only available if the double-precision field is nonzero.

**SIMDReg, bits [3:0]**

Advanced SIMD registers. Indicates support for the Advanced SIMD register bank. Defined values are:

`0000` Not supported.

`0001` Supported, 16 x 64-bit registers.

`0010` Supported, 32 x 64-bit registers.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and `0010`.

If this field is nonzero:

- All VFP LDC, STC, MCR, and MRC instructions are supported.
- If the MCRR and MRRC instructions are supported then the corresponding VFP instructions are supported.

## Accessing the MVFR0_EL1:

To access the MVFR0_EL1:

```
MRS <Xt>, MVFR0_EL1 ; Read MVFR0_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.69 MVFR1_EL1, AArch32 Media and VFP Feature Register 1

The MVFR1_EL1 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register MVFR1_EL1 is architecturally mapped to AArch32 System register MVFR1.

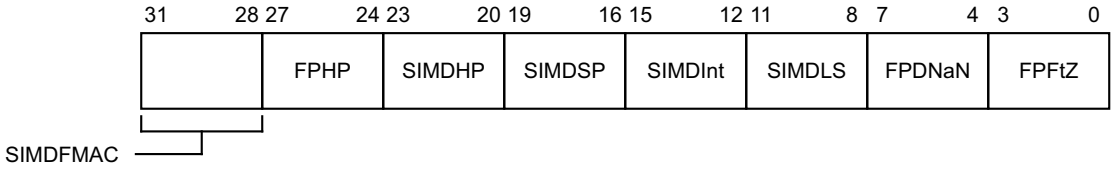
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

MVFR1_EL1 is a 32-bit register.

### Field descriptions

The MVFR1_EL1 bit assignments are:




**SIMDFMAC, bits [31:28]**

Advanced SIMD Fused Multiply-Accumulate. Indicates whether the floating-point and Advanced SIMD implementation includes the fused multiply accumulate instructions. Defined values are:

`0000` Not implemented.

`0001` Implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

The floating-point and Advanced SIMD implementations must provide the same level of support for these instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**FPHP, bits [27:24]**

Floating Point Half Precision. Indicates whether the floating-point implementation provides half-precision floating-point conversion instructions. Defined values are:

`0000` Not implemented.

`0001` Instructions to convert between half-precision and single-precision implemented.

`0010` As for `0b0001`, and also instructions to convert between half-precision and double-precision implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

**SIMDHP, bits [23:20]**

Advanced SIMD Half Precision. Indicates whether the floating-point and Advanced SIMD implementation provides half-precision floating-point conversion instructions. Defined values are:

`0000` Not implemented.

`0001` Implemented. This value is permitted only if the SIMDSP field is `0001`.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**SIMDSP, bits [19:16]**

Advanced SIMD Single Precision. Indicates whether the floating-point and Advanced SIMD implementation provides single-precision floating-point instructions. Defined values are:

`0000` Not implemented.

`0001` Implemented. This value is permitted only if the SIMDInt field is `0001`.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**SIMDInt, bits [15:12]**

Advanced SIMD Integer. Indicates whether the floating-point and Advanced SIMD implementation provides integer instructions. Defined values are:

`0000` Not implemented.

`0001` Implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**SIMDLS, bits [11:8]**

Advanced SIMD Load/Store. Indicates whether the floating-point and Advanced SIMD implementation provides load/store instructions. Defined values are:

`0000` Not implemented.

`0001` Implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**FPDNaN, bits [7:4]**

Default NaN mode. Indicates whether the VFP hardware implementation supports only the Default NaN mode. Defined values are:

`0000` Hardware supports only the Default NaN mode.

`0001` Hardware supports propagation of NaN values.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**FPFtZ, bits [3:0]**

Flush to Zero mode. Indicates whether the VFP hardware implementation supports only the Flush-to-Zero mode of operation. Defined values are:

`0000` Hardware supports only the Flush-to-Zero mode of operation.

`0001` Hardware supports full denormalized number arithmetic.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

## Accessing the MVFR1_EL1:

To access the MVFR1_EL1:

```
MRS <Xt>, MVFR1_EL1 ; Read MVFR1_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.70 MVFR2_EL1, AArch32 Media and VFP Feature Register 2

The MVFR2_EL1 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register MVFR2_EL1 is architecturally mapped to AArch32 System register MVFR2.

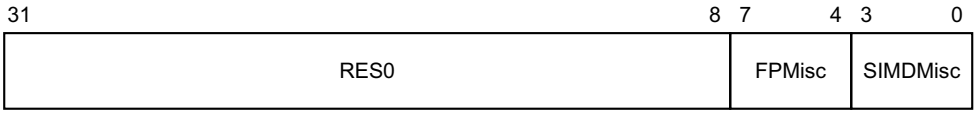
In an AArch64-only implementation, this register is UNKNOWN.

**Attributes**

MVFR2_EL1 is a 32-bit register.

#### Field descriptions

The MVFR2_EL1 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**FPMisc, bits [7:4]**

Indicates support for miscellaneous VFP features.

`0000` No support for miscellaneous features.

`0001` Support for Floating-point selection.

`0010` As `0001`, and Floating-point Conversion to Integer with Directed Rounding modes.

`0011` As `0010`, and Floating-point Round to Integral Floating-point.

`0100` As `0011`, and Floating-point MaxNum and MinNum.

All other values are reserved.

In ARMv8-A the only permitted value is `0100`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SIMDMisc, bits [3:0]**

Indicates support for miscellaneous Advanced SIMD features.

`0000` No support for miscellaneous features.

`0001` Floating-point Conversion to Integer with Directed Rounding modes.

`0010` As `0001`, and Floating-point Round to Integral Floating-point.

`0011` As `0010`, and Floating-point MaxNum and MinNum.

All other values are reserved.

In ARMv8-A the only permitted value is `0011`.

## Accessing the MVFR2_EL1:

To access the MVFR2_EL1:

```
MRS <Xt>, MVFR2_EL1 ; Read MVFR2_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0011 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.71 PAR_EL1, Physical Address Register

The PAR_EL1 characteristics are:

**Purpose**

Returns the output address (OA) from an address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

On a successful conversion, the PAR_EL1 can return a value that indicates the resulting attributes, rather than the values that appear in the translation table descriptors. More precisely:

- The ATTR and SH fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the translation table descriptors.
- See the NS bit description for constraints on the value it returns.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register PAR_EL1 is architecturally mapped to AArch32 System register PAR.

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PAR_EL1 is a 64-bit register.

#### Field descriptions

The PAR_EL1 bit assignments are:

***For all register layouts:***

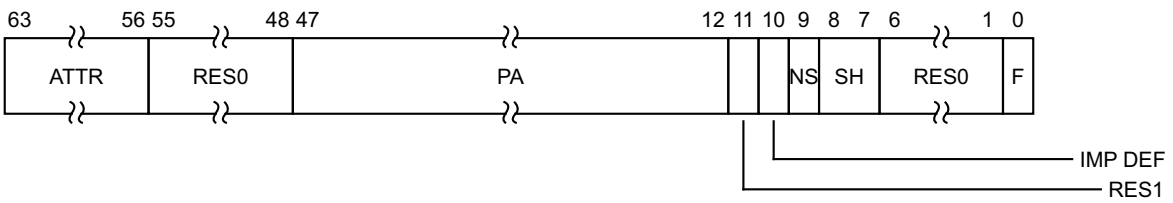
**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

`0` Address translation completed successfully.

`1` Address translation aborted.

***When PAR_EL1.F==0:***





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This section describes the register value returned by the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

**ATTR, bits [63:56]**

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR_EL1, MAIR_EL2, and MAIR_EL3.

See the Usage constraints for more information about the value returned.

**Bits [55:48]**

Reserved, RES0.

**PA, bits [47:12]**

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[47:12].

**Bit [11]**

Reserved, RES1.

**IMP DEF, bit [10]**

IMPLEMENTATION DEFINED.

**NS, bit [9]**

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

**SH, bits [8:7]**

Shareability attribute, for the returned output address. Permitted values are:

`00` Non-shareable.

`10` Outer Shareable.

`11` Inner Shareable.

The value `01` is reserved.

**Note**

This field returns the value `10` for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

See the Usage constraints for more information about the value returned.

**Bits [6:1]**

Reserved, RES0.

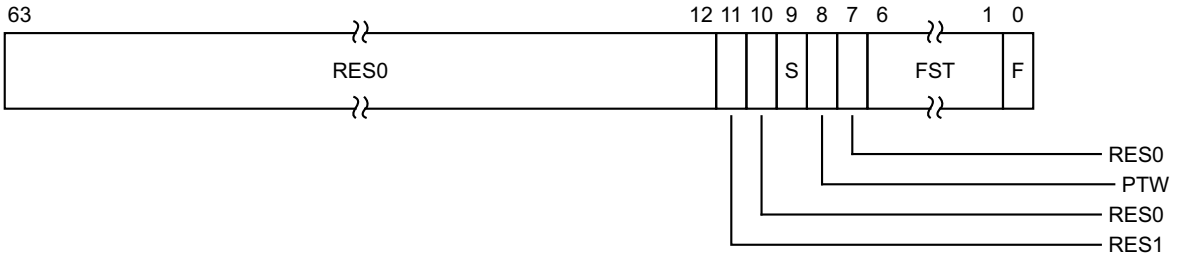
**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

`0` Address translation completed successfully.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

***When PAR_EL1.F==1:***




This section describes the register value returned by the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

**Bits [63:12]**

Reserved, RES0.

**Bit [11]**

Reserved, RES1.

**Bit [10]**

Reserved, RES0.

**S, bit [9]**

Indicates the translation stage at which the translation aborted:

0 Translation aborted because of a fault in the stage 1 translation.

1 Translation aborted because of a fault in the stage 2 translation.

**PTW, bit [8]**

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

**Bit [7]**

Reserved, RES0.

**FST, bits [6:1]**

Fault status code, as shown in the Data Abort ESR encoding.

**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

1 Address translation aborted.

## Accessing the PAR_EL1:

To access the PAR_EL1:

```
MRS <Xt>, PAR_EL1 ; Read PAR_EL1 into Xt
MSR PAR_EL1, <Xt> ; Write Xt to PAR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0111 | 0100 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.72 REVIDR_EL1, Revision ID Register

The REVIDR_EL1 characteristics are:

**Purpose**

Provides implementation-specific minor revision information.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TID1==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register REVIDR_EL1 is architecturally mapped to AArch32 System register REVIDR.

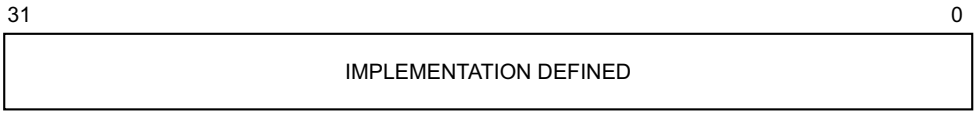
If REVIDR_EL1 has the same value as MIDR_EL1, then its contents have no significance.

**Attributes**

REVIDR_EL1 is a 32-bit register.

#### Field descriptions

The REVIDR_EL1 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the REVIDR_EL1:

To access the REVIDR_EL1:

```
MRS <Xt>, REVIDR_EL1 ; Read REVIDR_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0000 | 0000 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.73 RMR_EL1, Reset Management Register (if EL2 and EL3 not implemented)

The RMR_EL1 characteristics are:

**Purpose**

If EL1 is the highest implemented Exception level:

- A write to the register can request a Warm reset.
- If EL1 can use AArch32 and AArch64, specifies the Execution state that the PE boots into on a Warm reset.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 |
|---|---|
| - | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register RMR_EL1 is architecturally mapped to AArch32 System register RMR (at EL1).

Only implemented if EL1 is the highest implemented Exception level. In this case:

- If EL1 can use AArch32 and AArch64 then this register must be implemented.
- If EL1 cannot use AArch32 then it is IMPLEMENTATION DEFINED whether the register is implemented.

When this register is not implemented its encoding is UNDEFINED.

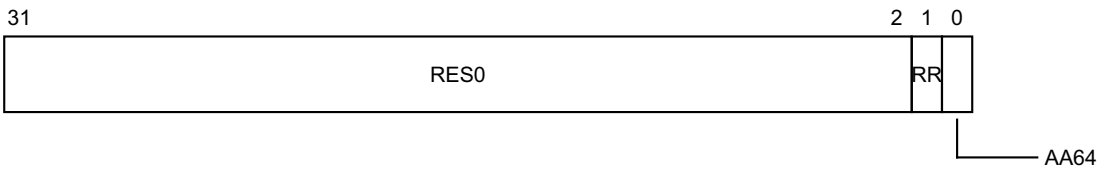
See the field descriptions for the reset values. These apply whenever the register is implemented.

**Attributes**

RMR_EL1 is a 32-bit register.

#### Field descriptions

The RMR_EL1 bit assignments are:




**Bits [31:2]**

Reserved, RES0.

**RR, bit [1]**

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to 0. This reset value applies on a Cold reset and on a Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**AA64, bit [0]**

When EL1 can use AArch32, determines which Execution state the PE boots into after a Warm reset:

`0` AArch32.

`1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If EL1 cannot use AArch32 this bit is RAO/WI.

If this field is implemented as an RW field, this field resets to `1`. This reset value applies on a Cold reset.

### Accessing the RMR_EL1:

To access the RMR_EL1:

```
MRS <Xt>, RMR_EL1 ; Read RMR_EL1 into Xt
MSR RMR_EL1, <Xt> ; Write Xt to RMR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1100 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.74 RMR_EL2, Reset Management Register (if EL3 not implemented)

The RMR_EL2 characteristics are:

**Purpose**

If EL2 is the highest implemented Exception level:

- A write to the register can request a Warm reset.
- If EL2 can use AArch32 and AArch64, specifies the Execution state that the PE boots into on a Warm reset.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register RMR_EL2 is architecturally mapped to AArch32 System register HRMR.

Only implemented if EL2 is the highest implemented Exception level. In this case:

- If EL2 can use AArch32 and AArch64 then this register must be implemented.
- If EL2 cannot use AArch32 then it is IMPLEMENTATION DEFINED whether the register is implemented.

When this register is not implemented its encoding is UNDEFINED.

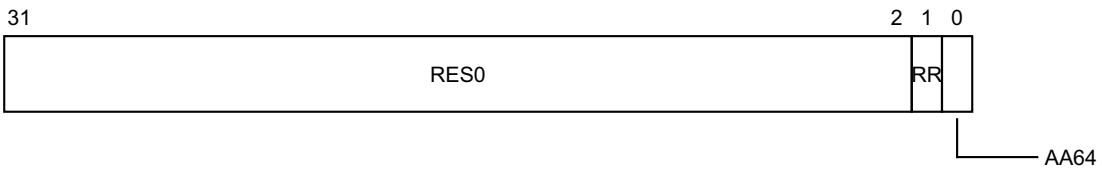
See the field descriptions for the reset values. These apply whenever the register is implemented.

**Attributes**

RMR_EL2 is a 32-bit register.

#### Field descriptions

The RMR_EL2 bit assignments are:




**Bits [31:2]**

Reserved, RES0.

**RR, bit [1]**

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to 0. This reset value applies on a Cold reset and on a Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**AA64, bit [0]**

When EL2 can use AArch32, determines which Execution state the PE boots into after a Warm reset:

`0` AArch32.

`1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If EL2 cannot use AArch32 this bit is RAO/WI.

If this field is implemented as an RW field, this field resets to 1. This reset value applies on a Cold reset.

## Accessing the RMR_EL2:

To access the RMR_EL2:

```
MRS <Xt>, RMR_EL2 ; Read RMR_EL2 into Xt
MSR RMR_EL2, <Xt> ; Write Xt to RMR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1100 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.75 RMR_EL3, Reset Management Register (if EL3 implemented)

The RMR_EL3 characteristics are:

**Purpose**

If EL1 is the highest implemented Exception level:

- A write to the register can request a Warm reset.
- If EL2 can use AArch32 and AArch64, specifies the Execution state that the PE boots into on a Warm reset.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register RMR_EL3 is architecturally mapped to AArch32 System register RMR (at EL3).

When EL3 is implemented:

- If EL3 can use AArch32 and AArch64 then this register must be implemented.
- If EL3 cannot use AArch32 then it is IMPLEMENTATION DEFINED whether the register is implemented.

When this register is not implemented its encoding is UNDEFINED.

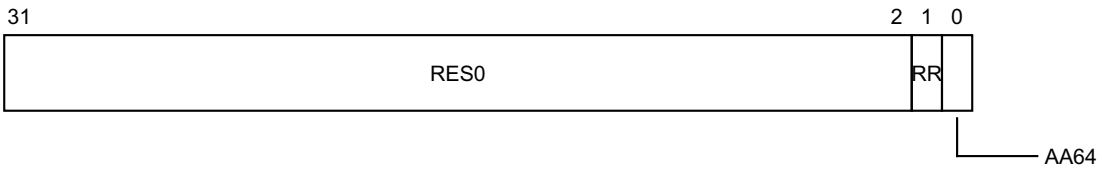
See the field descriptions for the reset values. These apply whenever the register is implemented.

**Attributes**

RMR_EL3 is a 32-bit register.

#### Field descriptions

The RMR_EL3 bit assignments are:




**Bits [31:2]**

Reserved, RES0.

**RR, bit [1]**

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to 0. This reset value applies on a Cold reset and on a Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**AA64, bit [0]**

When EL3 can use AArch32, determines which Execution state the PE boots into after a Warm reset:

`0` AArch32.

`1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If EL3 cannot use AArch32 this bit is RAO/WI.

If this field is implemented as an RW field, this field resets to `1`. This reset value applies on a Cold reset.

### Accessing the RMR_EL3:

To access the RMR_EL3:

```
MRS <Xt>, RMR_EL3 ; Read RMR_EL3 into Xt
MSR RMR_EL3, <Xt> ; Write Xt to RMR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 1100 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.76 RVBAR_EL1, Reset Vector Base Address Register (if EL2 and EL3 not implemented)

The RVBAR_EL1 characteristics are:

**Purpose**

If EL1 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 |
|---|---|
| - | RO |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

Only implemented if the highest Exception level implemented is EL1.

**Attributes**

RVBAR_EL1 is a 64-bit register.

#### Field descriptions

The RVBAR_EL1 bit assignments are:




**Bits [63:0]**

Reset Address. The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

#### Accessing the RVBAR_EL1:

To access the RVBAR_EL1:

```
MRS <Xt>, RVBAR_EL1 ; Read RVBAR_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1100 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.77 RVBAR_EL2, Reset Vector Base Address Register (if EL3 not implemented)

The RVBAR_EL2 characteristics are:

**Purpose**

If EL2 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RO |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

Only implemented if the highest Exception level implemented is EL2.

**Attributes**

RVBAR_EL2 is a 64-bit register.

#### Field descriptions

The RVBAR_EL2 bit assignments are:




**Bits [63:0]**

Reset Address. The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

#### Accessing the RVBAR_EL2:

To access the RVBAR_EL2:

```
MRS <Xt>, RVBAR_EL2 ; Read RVBAR_EL2 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1100 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.78 RVBAR_EL3, Reset Vector Base Address Register (if EL3 implemented)

The RVBAR_EL3 characteristics are:

**Purpose**

If EL3 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RO | RO |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

Only implemented if the highest Exception level implemented is EL3.

**Attributes**

RVBAR_EL3 is a 64-bit register.

#### Field descriptions

The RVBAR_EL3 bit assignments are:




**Bits [63:0]**

Reset Address. The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

#### Accessing the RVBAR_EL3:

To access the RVBAR_EL3:

```
MRS <Xt>, RVBAR_EL3 ; Read RVBAR_EL3 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 1100 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.79 S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED registers

The S3_<op1>_<Cn>_<Cm>_<op2> characteristics are:

**Purpose**

This area of the instruction set space is reserved for IMPLEMENTATION DEFINED registers.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| IMP DEF | IMP DEF | IMP DEF | IMP DEF | IMP DEF | IMP DEF |

The numbers in these register names are encoded in decimal without leading zeroes, and the Cn and Cm fields require a literal C before the number. For example, S3_4_C11_C9_7.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TIDCP==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

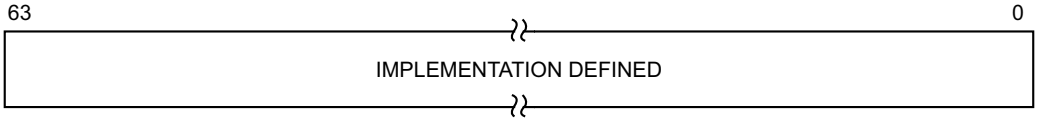
There are no configuration notes.

**Attributes**

S3_<op1>_<Cn>_<Cm>_<op2> is a 64-bit register.

#### Field descriptions

The S3_<op1>_<Cn>_<Cm>_<op2> bit assignments are:




**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

#### Accessing the S3_<op1>_<Cn>_<Cm>_<op2>:

To access the S3_<op1>_<Cn>_<Cm>_<op2>:

```
MRS <Xt>, S3_<op1>_<Cn>_<Cm>_<op2> ; Read S3_<op1>_<Cn>_<Cm>_<op2> into Xt
MSR S3_<op1>_<Cn>_<Cm>_<op2>, <Xt> ; Write Xt to S3_<op1>_<Cn>_<Cm>_<op2>
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | xxx | 1x11 | xxxx | xxx |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.80 SCR_EL3, Secure Configuration Register

The SCR_EL3 characteristics are:

**Purpose**

Defines the configuration of the current Security state. It specifies:

- The Security state of EL0 and EL1, either Secure or Non-secure.
- The Execution state at lower Exception levels.
- Whether IRQ, FIQ, and External Abort interrupts are taken to EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register SCR_EL3 can be mapped to AArch32 System register SCR, but this is not architecturally mandated.

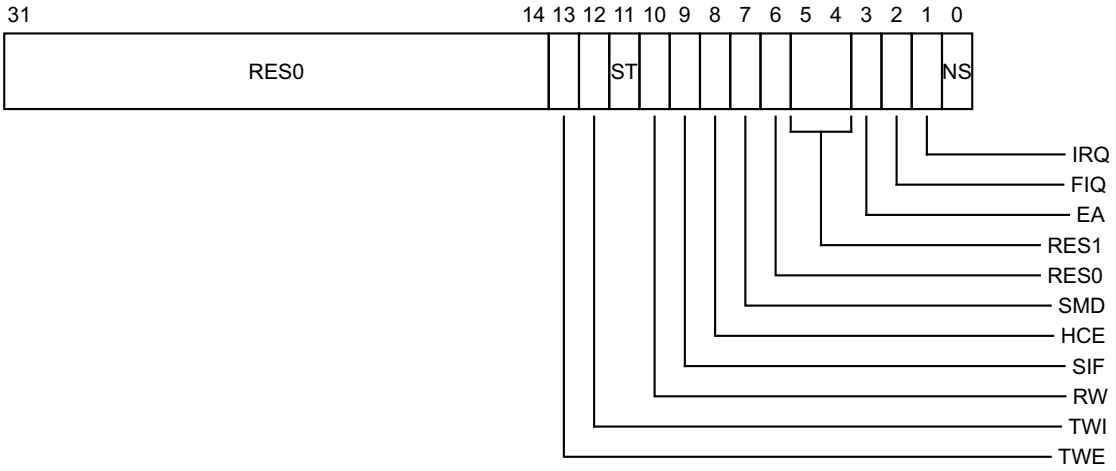
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

SCR_EL3 is a 32-bit register.

#### Field descriptions

The SCR_EL3 bit assignments are:




**Bits [31:14]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TWE, bit [13]**

Traps EL2, EL1, and EL0 execution of WFE instructions to EL3, from both Security states and both Execution states.

0 EL2, EL1, and EL0 execution of WFE instructions is not trapped to EL3.

1 Any attempt to execute a WFE instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

**TWI, bit [12]**

Traps EL2, EL1, and EL0 execution of WFI instructions to EL3, from both Security states and both Execution states.

0 EL2, EL1, and EL0 execution of WFI instructions is not trapped to EL3.

1 Any attempt to execute a WFI instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

**ST, bit [11]**

Traps Secure EL1 accesses to the Counter-timer Physical Secure timer registers to EL3, from AArch64 state only.

0 Secure EL1 using AArch64 accesses to the CNTPS_TVAL_EL1, CNTPS_CTL_EL1, and CNTPS_CVAL_EL1 are trapped to EL3.

1 Secure EL1 using AArch64 accesses to the CNTPS_TVAL_EL1, CNTPS_CTL_EL1, and CNTPS_CVAL_EL1 are not trapped to EL3.

**RW, bit [10]**

Execution state control for lower Exception levels.

0 Lower levels are all AArch32.

1 The next lower level is AArch64.

If EL2 is present:

- EL2 is AArch64.
- EL2 controls EL1 and EL0 behaviors.

If EL2 is not present:

- EL1 is AArch64.
- EL0 is determined by the Execution state described in the current process state when executing at EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If all lower exception levels cannot use AArch32 then this bit is RAO/WI.

This bit is permitted to be cached in a TLB.

**SIF, bit [9]**

Secure instruction fetch. When the PE is in Secure state, this bit disables instruction fetch from Non-secure memory. The possible values for this bit are:

0 Secure state instruction fetches from Non-secure memory are permitted.

1 Secure state instruction fetches from Non-secure memory are not permitted.

This bit is permitted to be cached in a TLB.

**HCE, bit [8]**

Hypervisor Call instruction enable. Enables HVC instructions at EL3, EL2, and Non-secure EL1, in both Execution states.

0 HVC instructions are UNDEFINED at EL3, EL2, and Non-secure EL1. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

1 HVC instructions are enabled at EL1 and above.

**Note**

HVC instructions are always UNDEFINED at EL0.

If EL2 is not implemented, this bit is RES0.

**SMD, bit [7]**

Secure Monitor Call disable. Disables SMC instructions at EL1 and above, from both Security states and both Execution states.

0 SMC instructions are enabled at EL1 and above.

1 SMC instructions are UNDEFINED at EL1 and above.

**Note**

SMC instructions are always UNDEFINED at EL0.

**Bit [6]**

Reserved, RES0.

**Bits [5:4]**

Reserved, RES1.

**EA, bit [3]**

External Abort and SError Interrupt Routing.

0 When executing at Exception levels below EL3, External Aborts and SError Interrupts are not taken to EL3.

In addition, when executing at EL3:

- SError Interrupts are not taken.
- External Aborts are taken to EL3.

1 When executing at any Exception level, External Aborts and SError Interrupts are taken to EL3.

For more information, see *Asynchronous exception routing* on page D1-1569.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**FIQ, bit [2]**

Physical FIQ Routing.

0 When executing at Exception levels below EL3, physical FIQ interrupts are not taken to EL3.

When executing at EL3, physical FIQ interrupts are not taken.

1 When executing at any Exception level, physical FIQ interrupts are taken to EL3.

For more information, see *Asynchronous exception routing* on page D1-1569.

**IRQ, bit [1]**

Physical IRQ Routing.

0 When executing at Exception levels below EL3, physical IRQ interrupts are not taken to EL3.

When executing at EL3, physical IRQ interrupts are not taken.

1 When executing at any Exception level, physical IRQ interrupts are taken to EL3.

For more information, see *Asynchronous exception routing* on page D1-1569.

**NS, bit [0]**

Non-secure bit.

0 Indicates that EL0 and EL1 are in Secure state, and so memory accesses from those Exception levels can access Secure memory.

When executing at EL3:

- The AT S1E2R, AT S1E2W, TLBI VAE2, TLBI VALE2, TLBI VAE2IS, TLBI VALE2IS, TLBI ALLE2, and TLBI ALLE2IS System instructions are UNDEFINED.
- Each AT S12E* System instruction executes as the corresponding AT S1E* instruction. For example, AT S12E0R executes as AT S1E0R.
- Each of the TLBI IPAS2E1, TLBI IPAS2E1IS, TLBI IPAS2LE1, and TLBI IPAS2LE1IS System instructions executes as a NOP.
- A TLBI VMALLS12E1 System instruction executes as TLBI VMALLE1, and a TLBI VMALLS12E1IS System instruction executes as TLBI VMALLE1IS.

1 Indicates that EL0 and EL1 are in Non-secure state, and so memory accesses from those Exception levels cannot access Secure memory.

## Accessing the SCR_EL3:

To access the SCR_EL3:

```
MRS <Xt>, SCR_EL3 ; Read SCR_EL3 into Xt
MSR SCR_EL3, <Xt> ; Write Xt to SCR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0001 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.81 SCTLR_EL1, System Control Register (EL1)

The SCTLR_EL1 characteristics are:

**Purpose**

Provides top level control of the system, including its memory system, at EL1 and EL0.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register SCTLR_EL1 is architecturally mapped to AArch32 System register SCTLR.

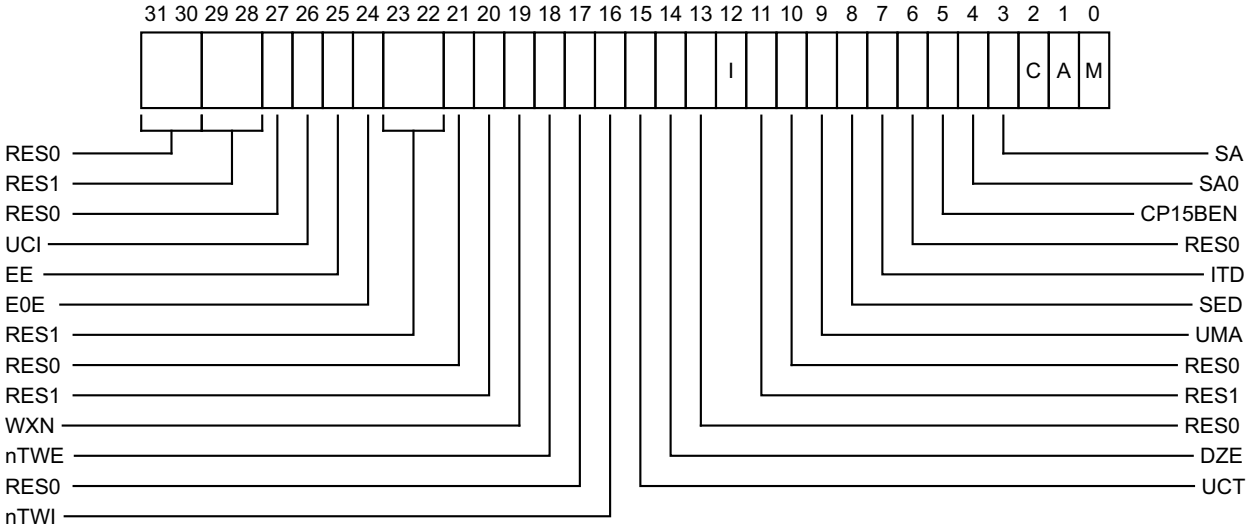
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL1 using AArch64. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

SCTLR_EL1 is a 32-bit register.

#### Field descriptions

The SCTLR_EL1 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:30]**

Reserved, RES0.

**Bits [29:28]**

Reserved, RES1.

**Bit [27]**

Reserved, RES0.

**UCI, bit [26]**

Traps EL0 execution of cache maintenance instructions to EL1, from AArch64 state only.

0 Any attempt to execute a DC CVAU, DC CIVAC, DC CVAC, or IC IVAU instruction at EL0 using AArch64 is trapped to EL1.

1 EL0 execution of cache maintenance instructions is not trapped to EL1.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**EE, bit [25]**

Endianness of data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime.

The possible values of this bit are:

0 Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.

1 Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to an IMPLEMENTATION DEFINED value.

**E0E, bit [24]**

Endianness of data accesses at EL0.

The possible values of this bit are:

0 Explicit data accesses at EL0 are little-endian.

1 Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0 then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at EL0 then this bit is RES1. This option is not permitted when SCTLR_EL1.EE is RES0.

This bit has no effect on the endianness of LDTR, LDTRH, LDTRSH, LDTRSW, STTR and STTRH instructions executed at EL1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**Bits [23:22]**

Reserved, RES1.

**Bit [21]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bit [20]**

Reserved, RES1.

**WXN, bit [19]**

Write permission implies XN (Execute-never). For the EL1&0 translation regime, this bit can force all memory regions that are writeable to be treated as XN. The possible values of this bit are:

0 This control has no effect on memory access permissions.

1 Any region that is writeable in the EL1&0 translation regime is forced to XN for accesses from software executing at EL1 or EL0.

The WXN bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**nTWE, bit [18]**

Traps EL0 execution of WFE instructions to EL1, from both Execution states.

0 Any attempt to execute a WFE instruction at EL0 is trapped to EL1, if the instruction would otherwise have caused the PE to enter a low-power state.

1 EL0 execution of WFE instructions is not trapped to EL1.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bit [17]**

Reserved, RES0.

**nTWI, bit [16]**

Traps EL0 execution of WFI instructions to EL1, from both Execution states.

0 Any attempt to execute a WFI instruction at EL0 is trapped EL1, if the instruction would otherwise have caused the PE to enter a low-power state.

1 EL0 execution of WFI instructions is not trapped to EL1.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**UCT, bit [15]**

Traps EL0 accesses to the CTR_EL0 to EL1, from AArch64 state only.

0 Accesses to the CTR_EL0 from EL0 using AArch64 are trapped to EL1.

1 Accesses to the CTR_EL0 from EL0 using AArch64 are not trapped to EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**DZE, bit [14]**

Traps EL0 execution of DC ZVA instructions to EL1, from AArch64 state only.

`0` Any attempt to execute a DC ZVA instruction at EL0 using AArch64 is trapped to EL1. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that DC ZVA instructions are not supported.

`1` EL0 execution of DC ZVA instructions is not trapped to EL1.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bit [13]**

Reserved, RES0.

**I, bit [12]**

Instruction access Cacheability control, for accesses at EL0 and EL1:

`0` All instruction access to Normal memory from EL0 and EL1 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

`1` This control has no effect on the Cacheability of instruction access to Normal memory from EL0 and EL1.

If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

When the value of the HCR_EL2.DC bit is 1, then instruction access to Normal memory from EL0 and EL1 are Cacheable regardless of the value of the SCTLR_EL1.I bit.

When this register has an architecturally-defined reset value, this field resets to `0`.

**Bit [11]**

Reserved, RES1.

**Bit [10]**

Reserved, RES0.

**UMA, bit [9]**

User Mask Access. Traps EL0 execution of MSR and MRS instructions that access the PSTATE.{D, A, I, F} masks to EL1, from AArch64 state only.

`0` Any attempt at EL0 using AArch64 to execute an `MRS`, `MSR(register)`, or `MSR(immediate)` instruction that accesses the DAIF is trapped to EL1.

`1` EL0 execution of `MRS`, `MSR(register)`, or `MSR(immediate)` instructions that access the DAIF is not trapped to EL1.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**SED, bit [8]**

SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

`0` SETEND instruction execution is enabled at EL0 using AArch32.

`1` SETEND instructions are UNDEFINED at EL0 using AArch32.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If EL0 cannot use AArch32, this bit is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**ITD, bit [7]**

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

`0` All IT instruction functionality is enabled at EL0 using AArch32.

`1` Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED:

- All encodings of the IT instruction with hw1[3:0]!=1000.
- All encodings of the subsequent instruction with the following values for hw1:

  `11xxxxxxxxxxxxx`
  All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.

  `1011xxxxxxxxxxx`
  All instructions in *Miscellaneous 16-bit instructions* on page F3-2471.

  `10100xxxxxxxxxx`
  ADD Rd, PC, #imm

  `01001xxxxxxxxxx`
  LDR Rd, [PC, #imm]

  `0100x1xxx1111xxx`
  ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.

  `010001xx1xxxx111`
  ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn.

  These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

  It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

  - A 16-bit instruction, that can only be followed by another 16-bit instruction.
  - The first half of a 32-bit instruction.

  This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

  An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If EL0 cannot use AArch32, this bit is RES1.

ITD is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**Bit [6]**

Reserved, RES0.

**CP15BEN, bit [5]**

CP15 memory barrier enable. Enables accesses to the CP15 DMB, DSB, and ISB barrier operations from EL0 using AArch32.

`0` EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.

`1` EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If EL0 cannot use AArch32, this bit is RES0.

CP15BEN is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAO/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**SA0, bit [4]**

Stack Alignment Check Enable for EL0. When set, use of the stack pointer as the base address in a load/store instruction at EL0 must be aligned to a 16-byte boundary, or a Stack Alignment Fault exception will be raised.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**SA, bit [3]**

Stack Alignment Check Enable. When set, use of the stack pointer as the base address in a load/store instruction at EL1 must be aligned to a 16-byte boundary, or a Stack Alignment Fault exception will be raised.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**C, bit [2]**

Cacheability control, for data caching.

`0` All data access to Normal memory from EL0 and EL1, and all Normal memory accesses to the EL1&0 stage 1 translation tables, are Non-cacheable for all levels of data and unified cache.

`1` This control has no effect on the Cacheability of:

- Data access to Normal memory from EL0 and EL1.
- Normal memory accesses to the EL1&0 stage 1 translation tables.

When the value of the HCR_EL2.DC bit is 1, the PE ignores SCLTR.C. This means that Non-secure EL0 and Non-secure EL1 data accesses to Normal memory are Cacheable.

When this register has an architecturally-defined reset value, this field resets to `0`.

**A, bit [1]**

Alignment check enable. This is the enable bit for Alignment fault checking at EL1 and EL0:

`0` Alignment fault checking disabled when executing at EL1 or EL0.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

`1` Alignment fault checking enabled when executing at EL1 or EL0.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**M, bit [0]**

MMU enable for EL1 and EL0 stage 1 address translation. Possible values of this bit are:

`0` EL1 and EL0 stage 1 address translation disabled.

See the SCTLR_EL1.I field for the behavior of instruction accesses to Normal memory.

`1` EL1 and EL0 stage 1 address translation enabled.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the value of HCR_EL2.{DC, TGE} is not {0, 0} then in Non-secure state the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of the field.

When this register has an architecturally-defined reset value, this field resets to `0`.

### Accessing the SCTLR_EL1:

To access the SCTLR_EL1:

```
MRS <Xt>, SCTLR_EL1 ; Read SCTLR_EL1 into Xt
MSR SCTLR_EL1, <Xt> ; Write Xt to SCTLR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0001 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.82 SCTLR_EL2, System Control Register (EL2)

The SCTLR_EL2 characteristics are:

**Purpose**

Provides top level control of the system, including its memory system, at EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register SCTLR_EL2 is architecturally mapped to AArch32 System register HSCTLR.

If EL2 is not implemented, this register is RES0 from EL3.

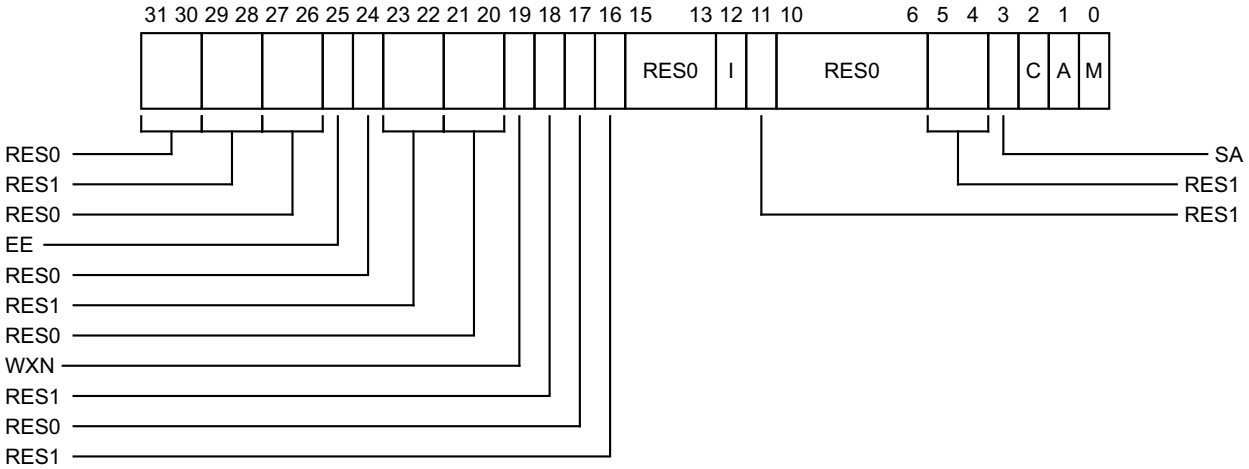
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 using AArch64. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

SCTLR_EL2 is a 32-bit register.

### Field descriptions

The SCTLR_EL2 bit assignments are:




**Bits [31:30]**

Reserved, RES0.

**Bits [29:28]**

Reserved, RES1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [27:26]**

Reserved, RES0.

**EE, bit [25]**

Endianness of data accesses at EL2, stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the EL1&0 translation regime.

The possible values of this bit are:

0 Explicit data accesses at EL2, stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are little-endian.

1 Explicit data accesses at EL2, stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to an IMPLEMENTATION DEFINED value.

**Bit [24]**

Reserved, RES0.

**Bits [23:22]**

Reserved, RES1.

**Bits [21:20]**

Reserved, RES0.

**WXN, bit [19]**

Write permission implies XN (Execute-never). For the EL2 translation regime, this bit can force all memory regions that are writeable to be treated as XN. The possible values of this bit are:

0 This control has no effect on memory access permissions.

1 Any region that is writeable in the EL2 translation regime is forced to XN for accesses from software executing at EL2.

The WXN bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bit [18]**

Reserved, RES1.

**Bit [17]**

Reserved, RES0.

**Bit [16]**

Reserved, RES1.

**Bits [15:13]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**I, bit [12]**

Instruction access Cacheability control, for accesses at EL2:

0 All instruction access to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

1 This control has no effect on the Cacheability of instruction access to Normal memory from EL2.

If the value of SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL1&0 or EL3 translation regimes.

When this register has an architecturally-defined reset value, this field resets to 0.

**Bit [11]**

Reserved, RES1.

**Bits [10:6]**

Reserved, RES0.

**Bits [5:4]**

Reserved, RES1.

**SA, bit [3]**

Stack Alignment Check Enable. When set, use of the stack pointer as the base address in a load/store instruction at EL2 must be aligned to a 16-byte boundary, or a Stack Alignment Fault exception will be raised.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**C, bit [2]**

Cacheability control, for data caching.

0 All data access to Normal memory from EL2, and all Normal memory accesses to the EL2 translation tables, are Non-cacheable for all levels of data and unified cache.

1 This control has no effect on the Cacheability of:

- Data access to Normal memory from EL2.
- Normal memory accesses to the EL2 translation tables.

This bit has no effect on the EL1&0 or EL3 translation regimes.

When this register has an architecturally-defined reset value, this field resets to 0.

**A, bit [1]**

Alignment check enable. This is the enable bit for Alignment fault checking at EL2:

0 Alignment fault checking disabled when executing at EL2.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

1 Alignment fault checking enabled when executing at EL2.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**M, bit [0]**

MMU enable for EL2 stage 1 address translation. Possible values of this bit are:

`0` EL2 stage 1 address translation disabled.
See the SCTLR_EL2.I field for the behavior of instruction accesses to Normal memory.

`1` EL2 stage 1 address translation enabled.

When this register has an architecturally-defined reset value, this field resets to `0`.

### Accessing the SCTLR_EL2:

To access the SCTLR_EL2:

```
MRS <Xt>, SCTLR_EL2 ; Read SCTLR_EL2 into Xt
MSR SCTLR_EL2, <Xt> ; Write Xt to SCTLR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0001 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.83 SCTLR_EL3, System Control Register (EL3)

The SCTLR_EL3 characteristics are:

**Purpose**

Provides top level control of the system, including its memory system, at EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

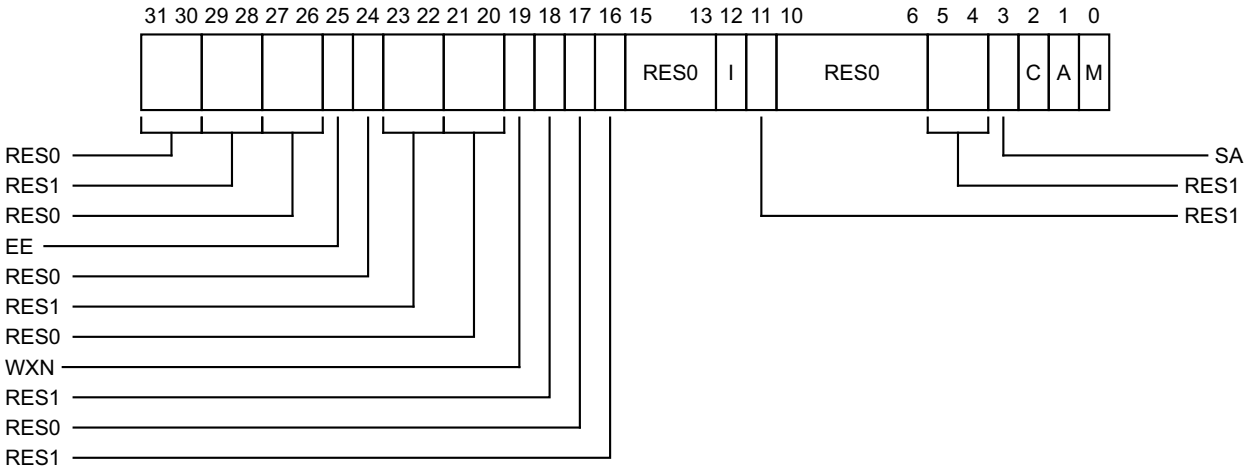
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL3 using AArch64.Other RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

SCTLR_EL3 is a 32-bit register.

#### Field descriptions

The SCTLR_EL3 bit assignments are:




**Bits [31:30]**

Reserved, RES0.

**Bits [29:28]**

Reserved, RES1.

**Bits [27:26]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EE, bit [25]**

Endianness of data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime.

The possible values of this bit are:

0 Explicit data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime are little-endian.

1 Explicit data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to an IMPLEMENTATION DEFINED value.

**Bit [24]**

Reserved, RES0.

**Bits [23:22]**

Reserved, RES1.

**Bits [21:20]**

Reserved, RES0.

**WXN, bit [19]**

Write permission implies XN (Execute-never). For the EL3 translation regime, this bit can force all memory regions that are writeable to be treated as XN. The possible values of this bit are:

0 This control has no effect on memory access permissions.

1 Any region that is writeable in the EL3 translation regime is forced to XN for accesses from software executing at EL3.

The WXN bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bit [18]**

Reserved, RES1.

**Bit [17]**

Reserved, RES0.

**Bit [16]**

Reserved, RES1.

**Bits [15:13]**

Reserved, RES0.

**I, bit [12]**

Instruction access Cacheability control, for accesses at EL3:

0 All instruction access to Normal memory from EL3 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR_EL3.M is 0, instruction accesses from stage 1 of the EL3 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 This control has no effect on the Cacheability of instruction access to Normal memory from EL3.

If the value of SCTLR_EL3.M is 0, instruction accesses from stage 1 of the EL3 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL1&0 or EL2 translation regimes.

When this register has an architecturally-defined reset value, this field resets to 0.

**Bit [11]**

Reserved, RES1.

**Bits [10:6]**

Reserved, RES0.

**Bits [5:4]**

Reserved, RES1.

**SA, bit [3]**

Stack Alignment Check Enable. When set, use of the stack pointer as the base address in a load/store instruction at EL3 must be aligned to a 16-byte boundary, or a Stack Alignment Fault exception will be raised.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**C, bit [2]**

Cacheability control, for data caching.

0 All data access to Normal memory from EL3, and all Normal memory accesses to the EL3 translation tables, are Non-cacheable for all levels of data and unified cache.

1 This control has no effect on the Cacheability of:

- Data access to Normal memory from EL3.
- Normal memory accesses to the EL3 translation tables.

This bit has no effect on the EL1&0 or EL2 translation regimes.

When this register has an architecturally-defined reset value, this field resets to 0.

**A, bit [1]**

Alignment check enable. This is the enable bit for Alignment fault checking at EL3:

0 Alignment fault checking disabled when executing at EL3.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

1 Alignment fault checking enabled when executing at EL3.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**M, bit [0]**

MMU enable for EL3 stage 1 address translation. Possible values of this bit are:

0 EL3 stage 1 address translation disabled.

See the SCTLR_EL3.I field for the behavior of instruction accesses to Normal memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 EL3 stage 1 address translation enabled.

When this register has an architecturally-defined reset value, this field resets to `0`.

### Accessing the SCTLR_EL3:

To access the SCTLR_EL3:

```
MRS <Xt>, SCTLR_EL3 ; Read SCTLR_EL3 into Xt
MSR SCTLR_EL3, <Xt> ; Write Xt to SCTLR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0001 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.84 TCR_EL1, Translation Control Register (EL1)

The TCR_EL1 characteristics are:

**Purpose**

Determines which of the Translation Table Base Registers defined the base address for a translation table walk required for the stage 1 translation of a memory access from EL0 or EL1. Also controls the translation table format and holds cacheability and shareability information.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

Any of the bits in TCR_EL1 are permitted to be cached in a TLB.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register TCR_EL1[31:0] is architecturally mapped to AArch32 System register TTBCR.

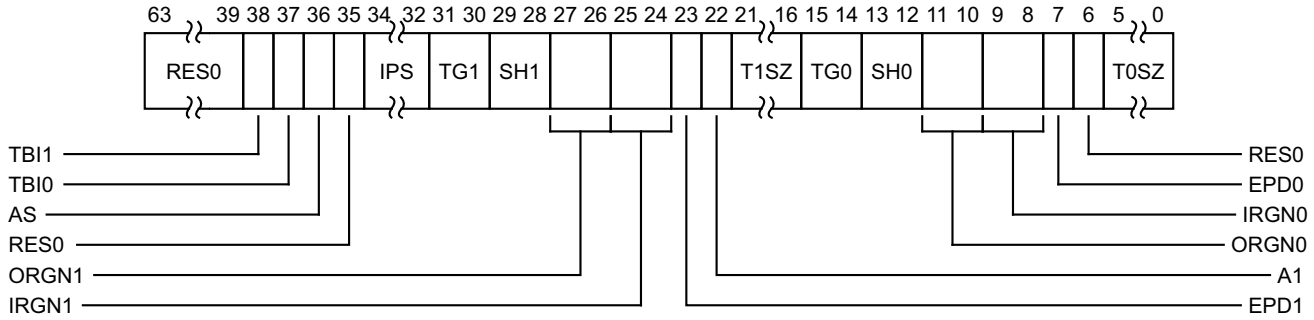
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TCR_EL1 is a 64-bit register.

### Field descriptions

The TCR_EL1 bit assignments are:




**Bits [63:39]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TBI1, bit [38]**

Top Byte ignored - indicates whether the top byte of an address is used for address match for the TTBR1_EL1 region, or ignored and used for tagged addresses.

`0` Top Byte used in the address calculation.

`1` Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR1_EL1. It has an effect whether the EL1&0 translation regime is enabled or not.

Additionally, this affects changes to the program counter, when TBI1 is 1 and bit [55] of the target address is 1, caused by:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

In these cases bits [63:56] of the address are also set to 1 before it is stored in the PC.

**TBI0, bit [37]**

Top Byte ignored - indicates whether the top byte of an address is used for address match for the TTBR0_EL1 region, or ignored and used for tagged addresses.

`0` Top Byte used in the address calculation.

`1` Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL1. It has an effect whether the EL1&0 translation regime is enabled or not.

Additionally, this affects changes to the program counter, when TBI0 is 1 and bit [55] of the target address is 0, caused by:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

In these cases bits [63:56] of the address are also set to 0 before it is stored in the PC.

**AS, bit [36]**

ASID Size.

`0` 8 bit - the upper 8 bits of TTBR0_EL1 and TTBR1_EL1 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB.

`1` 16 bit - the upper 16 bits of TTBR0_EL1 and TTBR1_EL1 are used for allocation and matching in the TLB.

If the implementation has only 8 bits of ASID, this field is RES0.

**Bit [35]**

Reserved, RES0.

**IPS, bits [34:32]**

Intermediate Physical Address Size.

`000` 32 bits, 4GB.

`001` 36 bits, 64GB.

`010` 40 bits, 1TB.

`011` 42 bits, 4TB.

`100` 44 bits, 16TB.

`101` 48 bits, 256TB.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Other values are reserved.

The reserved values behave in the same way as the 101 encoding, but software must not rely on this property as the behavior of the RESERVED values might change in a future revision of the architecture.

**TG1, bits [31:30]**

Granule size for the TTBR1_EL1.

01 16KB

10 4KB

11 64KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

**SH1, bits [29:28]**

Shareability attribute for memory associated with translation table walks using TTBR1_EL1.

00 Non-shareable

10 Outer Shareable

11 Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

**ORGN1, bits [27:26]**

Outer cacheability attribute for memory associated with translation table walks using TTBR1_EL1.

00 Normal memory, Outer Non-cacheable

01 Normal memory, Outer Write-Back Write-Allocate Cacheable

10 Normal memory, Outer Write-Through Cacheable

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable

**IRGN1, bits [25:24]**

Inner cacheability attribute for memory associated with translation table walks using TTBR1_EL1.

00 Normal memory, Inner Non-cacheable

01 Normal memory, Inner Write-Back Write-Allocate Cacheable

10 Normal memory, Inner Write-Through Cacheable

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable

**EPD1, bit [23]**

Translation table walk disable for translations using TTBR1_EL1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1_EL1. The encoding of this bit is:

0 Perform translation table walks using TTBR1_EL1.

1 A TLB miss on an address that is translated using TTBR1_EL1 generates a Translation fault. No translation table walk is performed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**A1, bit [22]**

Selects whether TTBR0_EL1 or TTBR1_EL1 defines the ASID. The encoding of this bit is:

0 TTBR0_EL1.ASID defines the ASID.

1 TTBR1_EL1.ASID defines the ASID.

**T1SZ, bits [21:16]**

The size offset of the memory region addressed by TTBR1_EL1. The region size is $2^{(64-T1SZ)}$ bytes.

The maximum and minimum possible values for T1SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

**TG0, bits [15:14]**

Granule size for the TTBR0_EL1.

00 4KB

01 64KB

10 16KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

**SH0, bits [13:12]**

Shareability attribute for memory associated with translation table walks using TTBR0_EL1.

00 Non-shareable

10 Outer Shareable

11 Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

**ORGN0, bits [11:10]**

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

00 Normal memory, Outer Non-cacheable

01 Normal memory, Outer Write-Back Write-Allocate Cacheable

10 Normal memory, Outer Write-Through Cacheable

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable

**IRGN0, bits [9:8]**

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

00 Normal memory, Inner Non-cacheable

01 Normal memory, Inner Write-Back Write-Allocate Cacheable

10 Normal memory, Inner Write-Through Cacheable

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EPD0, bit [7]**

Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0. The encoding of this bit is:

`0` Perform translation table walks using TTBR0.

`1` A TLB miss on an address that is translated using TTBR0 generates a Translation fault. No translation table walk is performed.

**Bit [6]**

Reserved, RES0.

**T0SZ, bits [5:0]**

The size offset of the memory region addressed by TTBR0_EL1. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

## Accessing the TCR_EL1:

To access the TCR_EL1:

```
MRS <Xt>, TCR_EL1 ; Read TCR_EL1 into Xt
MSR TCR_EL1, <Xt> ; Write Xt to TCR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0010 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.85 TCR_EL2, Translation Control Register (EL2)

The TCR_EL2 characteristics are:

**Purpose**

Controls translation table walks required for the stage 1 translation of memory accesses from EL2, and holds cacheability and shareability information for the accesses.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

Any of the bits in TCR_EL2 are permitted to be cached in a TLB.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register TCR_EL2 is architecturally mapped to AArch32 System register HTCR.

If EL2 is not implemented, this register is RES0 from EL3.

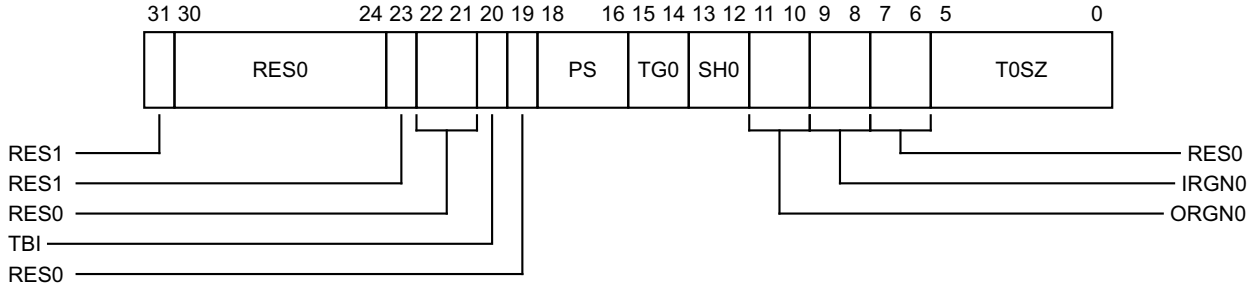
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TCR_EL2 is a 32-bit register.

### Field descriptions

The TCR_EL2 bit assignments are:




**Bit [31]**

Reserved, RES1.

**Bits [30:24]**

Reserved, RES0.

**Bit [23]**

Reserved, RES1.

**Bits [22:21]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TBI, bit [20]**

Top Byte ignored - indicates whether the top byte of an address is used for address match for the TTBR0_EL2 region, or ignored and used for tagged addresses.

`0` Top Byte used in the address calculation.

`1` Top Byte ignored in the address calculation.

This affects addresses generated in EL2 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL2. It has an effect whether the EL2 translation regime is enabled or not.

Additionally, this affects changes to the program counter, when TBI is 1, caused by:

- A branch or procedure return within EL2.
- An exception taken to EL2.
- An exception return to EL2.

In these cases bits [63:56] of the address are set to 0 before it is stored in the PC.

**Bit [19]**

Reserved, RES0.

**PS, bits [18:16]**

Physical Address Size.

`000` 32 bits, 4GB.

`001` 36 bits, 64GB.

`010` 40 bits, 1TB.

`011` 42 bits, 4TB.

`100` 44 bits, 16TB.

`101` 48 bits, 256TB.

Other values are reserved.

The reserved values behave in the same way as the `101` encoding, but software must not rely on this property as the behavior of the RESERVED values might change in a future revision of the architecture.

**TG0, bits [15:14]**

Granule size for the TTBR0_EL2.

`00` 4KB

`01` 64KB

`10` 16KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

**SH0, bits [13:12]**

Shareability attribute for memory associated with translation table walks using TTBR0_EL2.

`00` Non-shareable

`10` Outer Shareable

`11` Inner Shareable


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

**ORGN0, bits [11:10]**

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

00 Normal memory, Outer Non-cacheable

01 Normal memory, Outer Write-Back Write-Allocate Cacheable

10 Normal memory, Outer Write-Through Cacheable

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable

**IRGN0, bits [9:8]**

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

00 Normal memory, Inner Non-cacheable

01 Normal memory, Inner Write-Back Write-Allocate Cacheable

10 Normal memory, Inner Write-Through Cacheable

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable

**Bits [7:6]**

Reserved, RES0.

**T0SZ, bits [5:0]**

The size offset of the memory region addressed by TTBR0_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

## Accessing the TCR_EL2:

To access the TCR_EL2:

```
MRS <Xt>, TCR_EL2 ; Read TCR_EL2 into Xt
MSR TCR_EL2, <Xt> ; Write Xt to TCR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0010 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.86 TCR_EL3, Translation Control Register (EL3)

The TCR_EL3 characteristics are:

**Purpose**

Controls translation table walks required for the stage 1 translation of memory accesses from EL3, and holds cacheability and shareability information for the accesses.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

Any of the bits in TCR_EL3 are permitted to be cached in a TLB.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

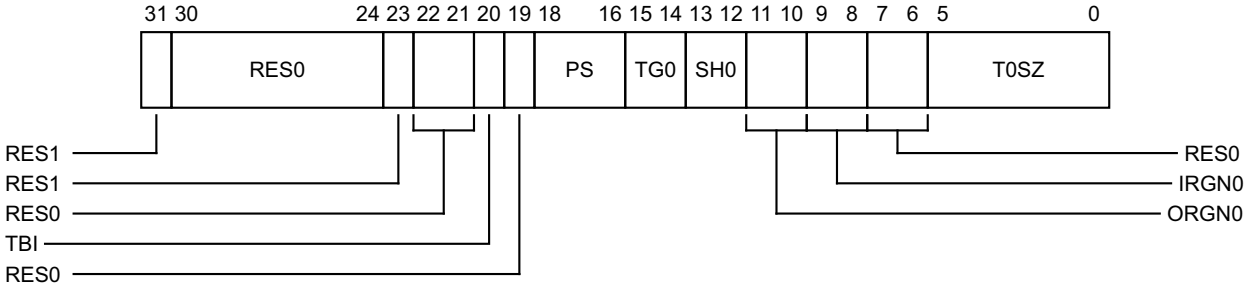
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TCR_EL3 is a 32-bit register.

### Field descriptions

The TCR_EL3 bit assignments are:




**Bit [31]**

Reserved, RES1.

**Bits [30:24]**

Reserved, RES0.

**Bit [23]**

Reserved, RES1.

**Bits [22:21]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TBI, bit [20]**

Top Byte ignored - indicates whether the top byte of an address is used for address match for the TTBR0_EL3 region, or ignored and used for tagged addresses.

`0` Top Byte used in the address calculation.

`1` Top Byte ignored in the address calculation.

This affects addresses generated in EL3 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL3. It has an effect whether the EL3 translation regime is enabled or not.

Additionally, this affects changes to the program counter, when TBI is 1, caused by:

- A branch or procedure return within EL3.
- A exception taken to EL3.
- An exception return to EL3.

In these cases bits [63:56] of the address are set to 0 before it is stored in the PC.

**Bit [19]**

Reserved, RES0.

**PS, bits [18:16]**

Physical Address Size.

`000` 32 bits, 4GB.

`001` 36 bits, 64GB.

`010` 40 bits, 1TB.

`011` 42 bits, 4TB.

`100` 44 bits, 16TB.

`101` 48 bits, 256TB.

Other values are reserved.

The reserved values behave in the same way as the `101` encoding, but software must not rely on this property as the behavior of the RESERVED values might change in a future revision of the architecture.

**TG0, bits [15:14]**

Granule size for the TTBR0_EL3.

`00` 4KB

`01` 64KB

`10` 16KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

**SH0, bits [13:12]**

Shareability attribute for memory associated with translation table walks using TTBR0_EL3.

`00` Non-shareable

`10` Outer Shareable

`11` Inner Shareable


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

**ORGN0, bits [11:10]**

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL3.

00 Normal memory, Outer Non-cacheable

01 Normal memory, Outer Write-Back Write-Allocate Cacheable

10 Normal memory, Outer Write-Through Cacheable

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable

**IRGN0, bits [9:8]**

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL3.

00 Normal memory, Inner Non-cacheable

01 Normal memory, Inner Write-Back Write-Allocate Cacheable

10 Normal memory, Inner Write-Through Cacheable

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable

**Bits [7:6]**

Reserved, RES0.

**T0SZ, bits [5:0]**

The size offset of the memory region addressed by TTBR0_EL3. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

## Accessing the TCR_EL3:

To access the TCR_EL3:

```
MRS <Xt>, TCR_EL3 ; Read TCR_EL3 into Xt
MSR TCR_EL3, <Xt> ; Write Xt to TCR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0010 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.87 TPIDR_EL0, EL0 Read/Write Software Thread ID Register

The TPIDR_EL0 characteristics are:

**Purpose**

Provides a location where software executing at EL0 can store thread identifying information, for OS management purposes.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | RW | RW | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register TPIDR_EL0[31:0] is architecturally mapped to AArch32 System register TPIDRURW.

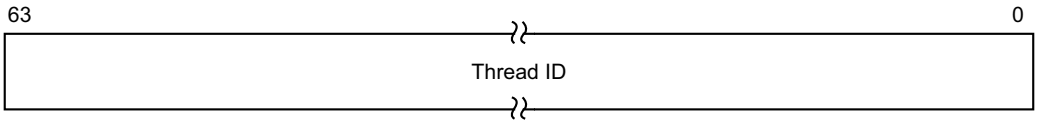
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TPIDR_EL0 is a 64-bit register.

### Field descriptions

The TPIDR_EL0 bit assignments are:




**Bits [63:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

### Accessing the TPIDR_EL0:

To access the TPIDR_EL0:

```
MRS <Xt>, TPIDR_EL0 ; Read TPIDR_EL0 into Xt
MSR TPIDR_EL0, <Xt> ; Write Xt to TPIDR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1101 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.88 TPIDR_EL1, EL1 Software Thread ID Register

The TPIDR_EL1 characteristics are:

**Purpose**

Provides a location where software executing at EL1 can store thread identifying information, for OS management purposes.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register TPIDR_EL1[31:0] is architecturally mapped to AArch32 System register TPIDRPRW.

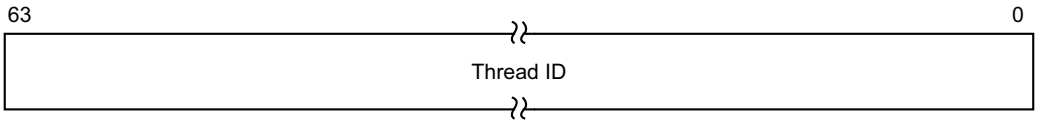
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TPIDR_EL1 is a 64-bit register.

### Field descriptions

The TPIDR_EL1 bit assignments are:




**Bits [63:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

### Accessing the TPIDR_EL1:

To access the TPIDR_EL1:

```
MRS <Xt>, TPIDR_EL1 ; Read TPIDR_EL1 into Xt
MSR TPIDR_EL1, <Xt> ; Write Xt to TPIDR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1101 | 0000 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.89 TPIDR_EL2, EL2 Software Thread ID Register

The TPIDR_EL2 characteristics are:

**Purpose**

Provides a location where software executing at EL2 can store thread identifying information, for OS management purposes.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register TPIDR_EL2[31:0] is architecturally mapped to AArch32 System register HTPIDR.

If EL2 is not implemented, this register is RES0 from EL3.

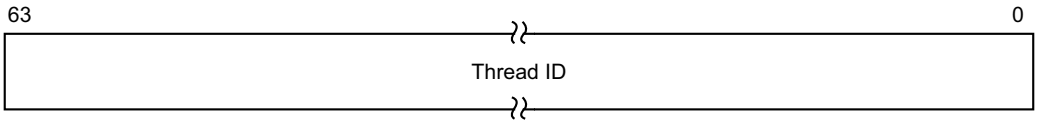
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TPIDR_EL2 is a 64-bit register.

#### Field descriptions

The TPIDR_EL2 bit assignments are:




**Bits [63:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

#### Accessing the TPIDR_EL2:

To access the TPIDR_EL2:

```
MRS <Xt>, TPIDR_EL2 ; Read TPIDR_EL2 into Xt
MSR TPIDR_EL2, <Xt> ; Write Xt to TPIDR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1101 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.90 TPIDR_EL3, EL3 Software Thread ID Register

The TPIDR_EL3 characteristics are:

**Purpose**

Provides a location where software executing at EL3 can store thread identifying information, for OS management purposes.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

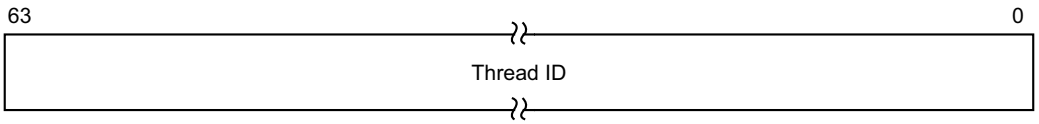
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TPIDR_EL3 is a 64-bit register.

### Field descriptions

The TPIDR_EL3 bit assignments are:




**Bits [63:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

### Accessing the TPIDR_EL3:

To access the TPIDR_EL3:

```
MRS <Xt>, TPIDR_EL3 ; Read TPIDR_EL3 into Xt
MSR TPIDR_EL3, <Xt> ; Write Xt to TPIDR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 1101 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.91 TPIDRRO_EL0, EL0 Read-Only Software Thread ID Register

The TPIDRRO_EL0 characteristics are:

**Purpose**

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RO | RW | RW | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register TPIDRRO_EL0[31:0] is architecturally mapped to AArch32 System register TPIDRURO.

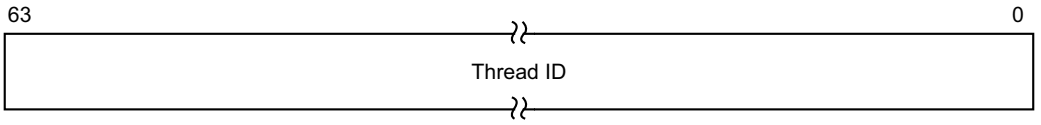
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TPIDRRO_EL0 is a 64-bit register.

#### Field descriptions

The TPIDRRO_EL0 bit assignments are:




**Bits [63:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

#### Accessing the TPIDRRO_EL0:

To access the TPIDRRO_EL0:

```
MRS <Xt>, TPIDRRO_EL0 ; Read TPIDRRO_EL0 into Xt
MSR TPIDRRO_EL0, <Xt> ; Write Xt to TPIDRRO_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1101 | 0000 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.92 TTBR0_EL1, Translation Table Base Register 0 (EL1)

The TTBR0_EL1 characteristics are:

**Purpose**

Holds the base address of translation table 0, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses at EL0 and EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

Any of the fields in this register are permitted to be cached in a TLB.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register TTBR0_EL1 is architecturally mapped to AArch32 System register TTBR0.

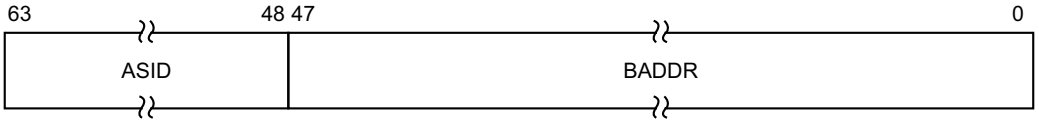
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TTBR0_EL1 is a 64-bit register.

### Field descriptions

The TTBR0_EL1 bit assignments are:




**ASID, bits [63:48]**

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

**BADDR, bits [47:0]**

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0, with the additional requirement that if they are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be on of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T0SZ, the stage of translation, and the translation granule size.

### Accessing the TTBR0_EL1:

To access the TTBR0_EL1:

```
MRS <Xt>, TTBR0_EL1 ; Read TTBR0_EL1 into Xt
MSR TTBR0_EL1, <Xt> ; Write Xt to TTBR0_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0010 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.93 TTBR0_EL2, Translation Table Base Register 0 (EL2)

The TTBR0_EL2 characteristics are:

**Purpose**

Holds the base address of the translation table for the stage 1 translation of memory accesses from EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

Any of the fields in this register are permitted to be cached in a TLB.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register TTBR0_EL2 is architecturally mapped to AArch32 System register HTTBR.

If EL2 is not implemented, this register is RES0 from EL3.

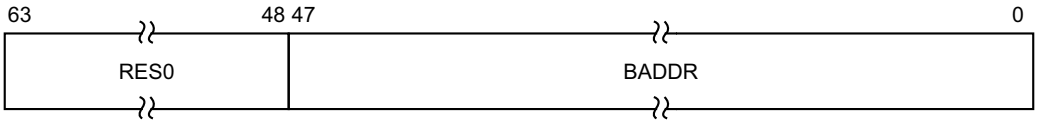
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TTBR0_EL2 is a 64-bit register.

### Field descriptions

The TTBR0_EL2 bit assignments are:




**Bits [63:48]**

Reserved, RES0.

**BADDR, bits [47:0]**

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0, with the additional requirement that if they are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be on of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL2.T0SZ, the stage of translation, and the translation granule size.

### Accessing the TTBR0_EL2:

To access the TTBR0_EL2:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRS <Xt>, TTBR0_EL2 ; Read TTBR0_EL2 into Xt
MSR TTBR0_EL2, <Xt> ; Write Xt to TTBR0_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0010 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.94 TTBR0_EL3, Translation Table Base Register 0 (EL3)

The TTBR0_EL3 characteristics are:

**Purpose**

Holds the base address of the translation table for the stage 1 translation of memory accesses from EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

Any of the fields in this register are permitted to be cached in a TLB.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

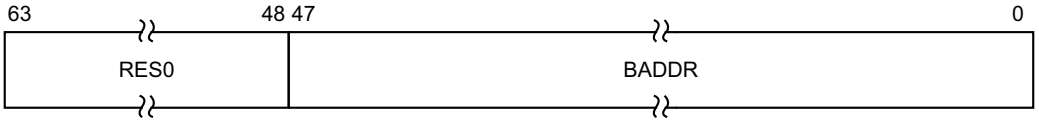
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TTBR0_EL3 is a 64-bit register.

### Field descriptions

The TTBR0_EL3 bit assignments are:




**Bits [63:48]**

Reserved, RES0.

**BADDR, bits [47:0]**

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0, with the additional requirement that if they are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be on of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL3.T0SZ, the stage of translation, and the translation granule size.

### Accessing the TTBR0_EL3:

To access the TTBR0_EL3:

```
MRS <Xt>, TTBR0_EL3 ; Read TTBR0_EL3 into Xt
MSR TTBR0_EL3, <Xt> ; Write Xt to TTBR0_EL3
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0010 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.95 TTBR1_EL1, Translation Table Base Register 1

The TTBR1_EL1 characteristics are:

**Purpose**

Holds the base address of translation table 1, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses at EL0 and EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

Any of the fields in this register are permitted to be cached in a TLB.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

**Configurations**

AArch64 System register TTBR1_EL1 is architecturally mapped to AArch32 System register TTBR1.

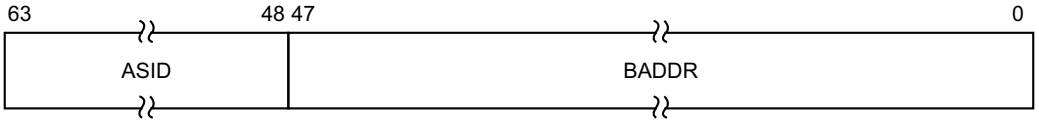
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

TTBR1_EL1 is a 64-bit register.

### Field descriptions

The TTBR1_EL1 bit assignments are:




**ASID, bits [63:48]**

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

**BADDR, bits [47:0]**

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0, with the additional requirement that if they are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be on of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T1SZ, the stage of translation, and the translation granule size.

### Accessing the TTBR1_EL1:

To access the TTBR1_EL1:

```
MRS <Xt>, TTBR1_EL1 ; Read TTBR1_EL1 into Xt
MSR TTBR1_EL1, <Xt> ; Write Xt to TTBR1_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 0010 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.96 VBAR_EL1, Vector Base Address Register (EL1)

The VBAR_EL1 characteristics are:

**Purpose**

Holds the vector base address for any exception that is taken to EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register VBAR_EL1[31:0] is architecturally mapped to AArch32 System register VBAR.

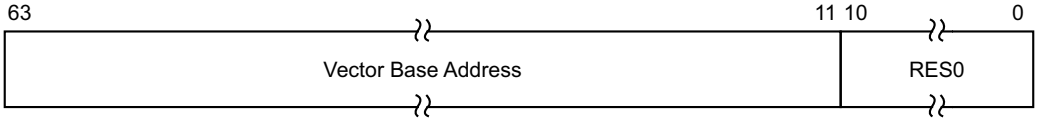
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

VBAR_EL1 is a 64-bit register.

### Field descriptions

The VBAR_EL1 bit assignments are:




**Bits [63:11]**

Vector Base Address. Base address of the exception vectors for exceptions taken to EL1.

If tagged addresses are being used, bits [55:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

If tagged addresses are not being used, bits [63:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

**Bits [10:0]**

Reserved, RES0.

### Accessing the VBAR_EL1:

To access the VBAR_EL1:

```
MRS <Xt>, VBAR_EL1 ; Read VBAR_EL1 into Xt
MSR VBAR_EL1, <Xt> ; Write Xt to VBAR_EL1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1100 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.97 VBAR_EL2, Vector Base Address Register (EL2)

The VBAR_EL2 characteristics are:

**Purpose**

Holds the vector base address for any exception that is taken to EL2.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register VBAR_EL2[31:0] is architecturally mapped to AArch32 System register HVBAR.

If EL2 is not implemented, this register is RES0 from EL3.

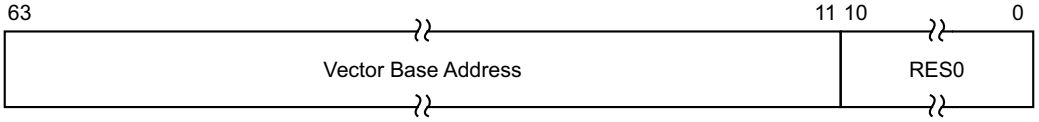
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

VBAR_EL2 is a 64-bit register.

### Field descriptions

The VBAR_EL2 bit assignments are:




**Bits [63:11]**

Vector Base Address. Base address of the exception vectors for exceptions taken to EL2.

If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.

If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.

**Bits [10:0]**

Reserved, RES0.

### Accessing the VBAR_EL2:

To access the VBAR_EL2:

```
MRS <Xt>, VBAR_EL2 ; Read VBAR_EL2 into Xt
MSR VBAR_EL2, <Xt> ; Write Xt to VBAR_EL2
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1100 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.98 VBAR_EL3, Vector Base Address Register (EL3)

The VBAR_EL3 characteristics are:

**Purpose**

Holds the vector base address for any exception that is taken to EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

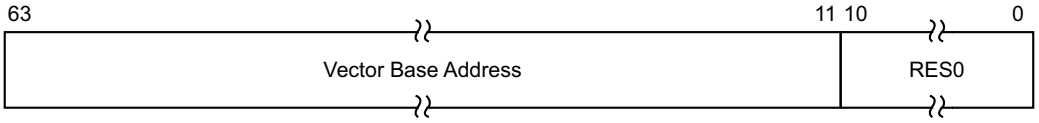
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

VBAR_EL3 is a 64-bit register.

### Field descriptions

The VBAR_EL3 bit assignments are:




**Bits [63:11]**

Vector Base Address. Base address of the exception vectors for exceptions taken to EL3.

If tagged addresses are being used, bits [55:48] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.

If tagged addresses are not being used, bits [63:48] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.

**Bits [10:0]**

Reserved, RES0.

### Accessing the VBAR_EL3:

To access the VBAR_EL3:

```
MRS <Xt>, VBAR_EL3 ; Read VBAR_EL3 into Xt
MSR VBAR_EL3, <Xt> ; Write Xt to VBAR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 1100 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.99 VMPIDR_EL2, Virtualization Multiprocessor ID Register

The VMPIDR_EL2 characteristics are:

**Purpose**

Holds the value of the Virtualization Multiprocessor ID. This is the value returned by Non-secure EL1 reads of MPIDR_EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register VMPIDR_EL2[31:0] is architecturally mapped to AArch32 System register VMPIDR.

If EL2 is not implemented, reads of this register return the value of the MPIDR_EL1, and writes to the register are ignored.

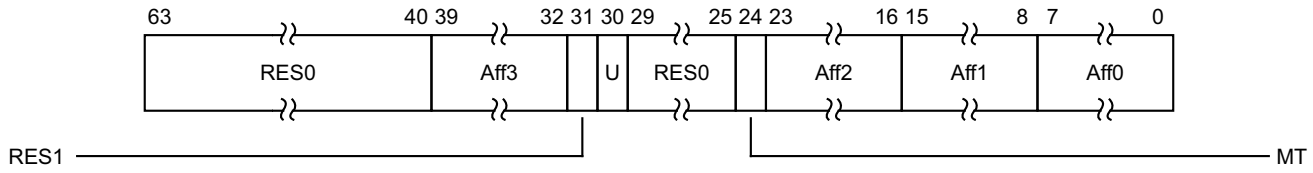
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

VMPIDR_EL2 is a 64-bit register.

#### Field descriptions

The VMPIDR_EL2 bit assignments are:




**Bits [63:40]**

Reserved, RES0.

**Aff3, bits [39:32]**

Affinity level 3. Highest level affinity field.

**Bit [31]**

Reserved, RES1.

**U, bit [30]**

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system. The possible values of this bit are:

0 Processor is part of a multiprocessor system.

1 Processor is part of a uniprocessor system.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [29:25]**

Reserved, RES0.

**MT, bit [24]**

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach. The possible values of this bit are:

0 Performance of PEs at the lowest affinity level is largely independent.

1 Performance of PEs at the lowest affinity level is very interdependent.

**Aff2, bits [23:16]**

Affinity level 2. Second highest level affinity field.

**Aff1, bits [15:8]**

Affinity level 1. Third highest level affinity field.

**Aff0, bits [7:0]**

Affinity level 0. Lowest level affinity field.

## Accessing the VMPIDR_EL2:

To access the VMPIDR_EL2:

```
MRS <Xt>, VMPIDR_EL2 ; Read VMPIDR_EL2 into Xt
MSR VMPIDR_EL2, <Xt> ; Write Xt to VMPIDR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0000 | 0000 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.100 VPIDR_EL2, Virtualization Processor ID Register

The VPIDR_EL2 characteristics are:

**Purpose**

Holds the value of the Virtualization Processor ID. This is the value returned by Non-secure EL1 reads of MIDR_EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register VPIDR_EL2 is architecturally mapped to AArch32 System register VPIDR.

If EL2 is not implemented, reads of this register return the value of the MIDR_EL1, and writes to the register are ignored.

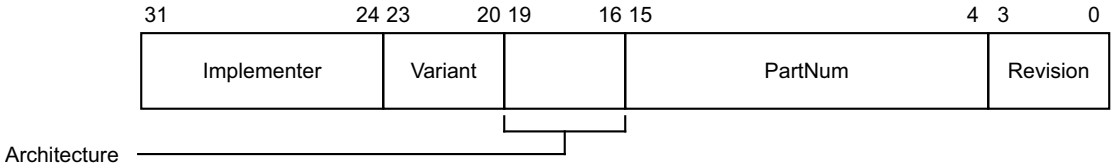
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

VPIDR_EL2 is a 32-bit register.

#### Field descriptions

The VPIDR_EL2 bit assignments are:




**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by ARM. Assigned codes include the following:

| Hex representation | ASCII representation | Implementer |
|---|---|---|
| 0x41 | A | ARM Limited |
| 0x42 | B | Broadcom Corporation |
| 0x43 | C | Cavium Inc. |
| 0x44 | D | Digital Equipment Corporation |
| 0x49 | I | Infineon Technologies AG |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Hex representation | ASCII representation | Implementer |
|---|---|---|
| 0x4D | M | Motorola or Freescale Semiconductor Inc. |
| 0x4E | N | NVIDIA Corporation |
| 0x50 | P | Applied Micro Circuits Corporation |
| 0x51 | Q | Qualcomm Inc. |
| 0x56 | V | Marvell International Ltd. |
| 0x69 | i | Intel Corporation |

ARM can assign codes that are not published in this manual. All values not assigned by ARM are reserved and must not be used.

**Variant, bits [23:20]**

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

**Architecture, bits [19:16]**

The permitted values of this field are:

0001 ARMv4

0010 ARMv4T

0011 ARMv5 (obsolete)

0100 ARMv5T

0101 ARMv5TE

0110 ARMv5TEJ

0111 ARMv6

1111 Architectural features are individually identified in the ID_* registers, see *Identification registers, functional group* on page G4-4162.

All other values are reserved.

**PartNum, bits [15:4]**

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by ARM, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

**Revision, bits [3:0]**

An IMPLEMENTATION DEFINED revision number for the device.

## Accessing the VPIDR_EL2:

To access the VPIDR_EL2:

```
MRS <Xt>, VPIDR_EL2 ; Read VPIDR_EL2 into Xt
MSR VPIDR_EL2, <Xt> ; Write Xt to VPIDR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0000 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.2.101 VTCR_EL2, Virtualization Translation Control Register

The VTCR_EL2 characteristics are:

**Purpose**

Controls the translation table walks required for the stage 2 translation of memory accesses from Non-secure EL0 and EL1, and holds cacheability and shareability information for the accesses.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

Any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register VTCR_EL2 is architecturally mapped to AArch32 System register VTCR.

If EL2 is not implemented, this register is RES0 from EL3.

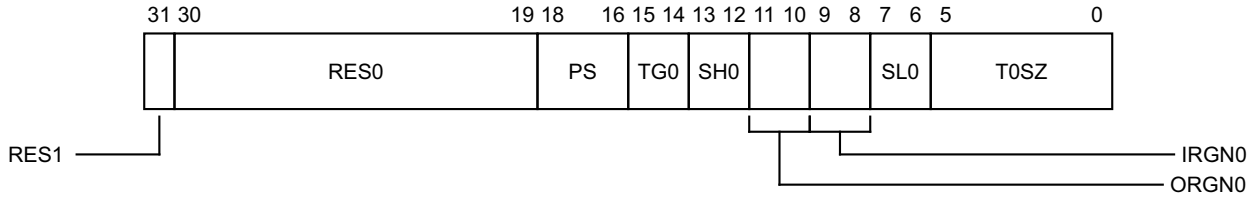
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

VTCR_EL2 is a 32-bit register.

#### Field descriptions

The VTCR_EL2 bit assignments are:




**Bit [31]**

Reserved, RES1.

**Bits [30:19]**

Reserved, RES0.

**PS, bits [18:16]**

Physical Address Size.

000 32 bits, 4GB.

001 36 bits, 64GB.

010 40 bits, 1TB.

011 42 bits, 4TB.

100 44 bits, 16TB.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

101 48 bits, 256TB.

Other values are reserved.

The reserved values behave in the same way as the 101 encoding, but software must not rely on this property as the behavior of the RESERVED values might change in a future revision of the architecture.

**TG0, bits [15:14]**

Granule size for the VTTBR_EL2.

00 4KB

01 64KB

10 16KB

Other values are reserved.

If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

**SH0, bits [13:12]**

Shareability attribute for memory associated with translation table walks using VTTBR_EL2.

00 Non-shareable

10 Outer Shareable

11 Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

**ORGN0, bits [11:10]**

Outer cacheability attribute for memory associated with translation table walks using VTTBR_EL2.

00 Normal memory, Outer Non-cacheable

01 Normal memory, Outer Write-Back Write-Allocate Cacheable

10 Normal memory, Outer Write-Through Cacheable

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable

**IRGN0, bits [9:8]**

Inner cacheability attribute for memory associated with translation table walks using VTTBR_EL2.

00 Normal memory, Inner Non-cacheable

01 Normal memory, Inner Write-Back Write-Allocate Cacheable

10 Normal memory, Inner Write-Through Cacheable

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable

**SL0, bits [7:6]**

Starting level of the VTCR_EL2 addressed region. The meaning of this field depends on the value of VTCR_EL2.TG0 (the granule size).

00 If TG0 is 00 (4KB granule), start at level 2. If TG0 is 10 (16KB granule) or 01 (64KB granule), start at level 3.

01 If TG0 is 00 (4KB granule), start at level 1. If TG0 is 10 (16KB granule) or 01 (64KB granule), start at level 2.

10 If TG0 is 00 (4KB granule), start at level 0. If TG0 is 10 (16KB granule) or 01 (64KB granule), start at level 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

**T0SZ, bits [5:0]**

The size offset of the memory region addressed by VTTBR_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

### Accessing the VTCR_EL2:

To access the VTCR_EL2:

```
MRS <Xt>, VTCR_EL2 ; Read VTCR_EL2 into Xt
MSR VTCR_EL2, <Xt> ; Write Xt to VTCR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0010 | 0001 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.2.102 VTTBR_EL2, Virtualization Translation Table Base Register

The VTTBR_EL2 characteristics are:

**Purpose**

Holds the base address of the translation table for the stage 2 translation of memory accesses from Non-secure EL0 and EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

Any of the fields in this register are permitted to be cached in a TLB.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register VTTBR_EL2 is architecturally mapped to AArch32 System register VTTBR.

If EL2 is not implemented, this register is RES0 from EL3.

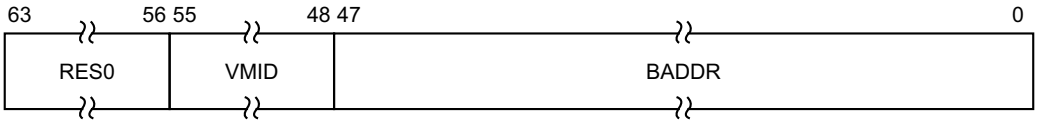
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

VTTBR_EL2 is a 64-bit register.

### Field descriptions

The VTTBR_EL2 bit assignments are:




**Bits [63:56]**

Reserved, RES0.

**VMID, bits [55:48]**

The VMID for the translation table.

**BADDR, bits [47:0]**

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0, with the additional requirement that if they are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be on of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of VTCR_EL2.T0SZ, the stage of translation, and the translation granule size.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the VTTBR_EL2:

To access the VTTBR_EL2:

```
MRS <Xt>, VTTBR_EL2 ; Read VTTBR_EL2 into Xt
MSR VTTBR_EL2, <Xt> ; Write Xt to VTTBR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0010 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.3 Debug registers

This section lists the Debug System registers in AArch64 state, in alphabetic order:

- The principal encoding space for debug registers is `Op0==0b10`, `Op1==` {0, 3, 4}. *Instructions for accessing debug System registers* on page C5-276 summarizes the registers in this encoding space and lists them in order of their encodings.
- In addition, the following registers in the `Op0==0b11` encoding space are classified as Debug registers:
  — MDCR_EL2.
  — MDCR_EL3.
  — SDER32_EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.1 DBGAUTHSTATUS_EL1, Debug Authentication Status register

The DBGAUTHSTATUS_EL1 characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 read accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register DBGAUTHSTATUS_EL1 is architecturally mapped to AArch32 System register DBGAUTHSTATUS.

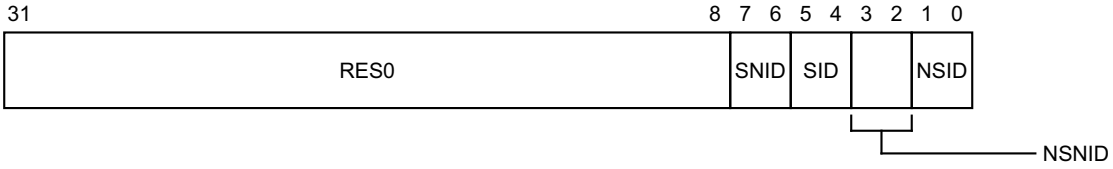
AArch64 System register DBGAUTHSTATUS_EL1 is architecturally mapped to External register DBGAUTHSTATUS_EL1.

**Attributes**

DBGAUTHSTATUS_EL1 is a 32-bit register.

#### Field descriptions

The DBGAUTHSTATUS_EL1 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**SNID, bits [7:6]**

Secure non-invasive debug. Possible values of this field are:

`00` Not implemented. EL3 is not implemented and the implemented Security state is Non-Secure state.

`10` Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.

`11` Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

Other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SID, bits [5:4]**

Secure invasive debug. Possible values of this field are:

00 Not implemented. EL3 is not implemented and the implemented Security state is Non-Secure state.

10 Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.

11 Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.

Other values are reserved.

**NSNID, bits [3:2]**

Non-secure non-invasive debug. Possible values of this field are:

00 Not implemented. EL3 is not implemented and the implemented Security state is Secure state.

10 Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.

11 Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

Other values are reserved.

**NSID, bits [1:0]**

Non-secure invasive debug. Possible values of this field are:

00 Not implemented. EL3 is not implemented and the implemented Security state is Secure state.

10 Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.

11 Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.

Other values are reserved.

## Accessing the DBGAUTHSTATUS_EL1:

To access the DBGAUTHSTATUS_EL1:

```
MRS <Xt>, DBGAUTHSTATUS_EL1 ; Read DBGAUTHSTATUS_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0111 | 1110 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.2 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n>_EL1 characteristics are:

**Purpose**

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

When the E field is zero, all the other fields in the register are ignored.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 and *Software Access debug event* on page H3-4881.

Subject to the prioritization rules:

If EDSCR.TDA==1, OSLSR.OSLK==0, and halting is allowed, EL1, EL2, and EL3 accesses to this register generate a Software Access debug event.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register DBGBCR<n>_EL1 is architecturally mapped to AArch32 System register DBGBCR<n>.

AArch64 System register DBGBCR<n>_EL1 is architecturally mapped to External register DBGBCR<n>_EL1.

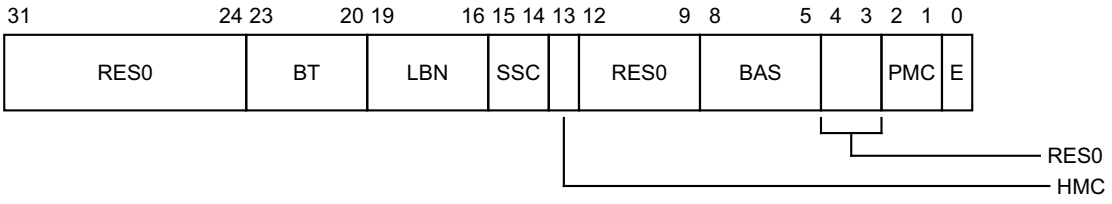
If breakpoint n is not implemented then this register is unallocated.

**Attributes**

DBGBCR<n>_EL1 is a 32-bit register.

## Field descriptions

The DBGBCR<n>_EL1 bit assignments are:




**Bits [31:24]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**BT, bits [23:20]**

Breakpoint Type. Possible values are:

`0000` Unlinked address match.

`0001` Linked address match.

`0010` Unlinked context ID match.

`0011` Linked context ID match

`0100` Unlinked address mismatch.

`0101` Linked address mismatch.

`1000` Unlinked VMID match.

`1001` Linked VMID match.

`1010` Unlinked VMID and context ID match.

`1011` Linked VMID and context ID match.

The field breaks down as follows:

- BT[3:1]: Base type.

  `000` Match address. DBGBVR<n>_EL1 is the address of an instruction.

  `010` Mismatch address. Behaves as type `000` if in an AArch64 translation, or if Halting debug is enabled and halting is allowed. Otherwise, DBGBVR<n>_EL1 is the address of an instruction to be stepped.

  `001` Match context ID. DBGBVR<n>_EL1.ContextID is a context ID.

  `100` Match VMID. DBGBVR<n>_EL1.VMID is a VMID.

  `101` Match VMID and context ID. DBGBVR<n>_EL1.ContextID is a context ID, and DBGBVR<n>_EL1.VMID is a VMID.

- BT[0]: Enable linking.

Values `0b011x` and `0b11xx` are always reserved. Constraints on breakpoint programming mean other values are reserved under some conditions. For more information, including the effect of programming this field to a reserved value, see *Reserved DBGBCR<n>_EL1.BT values* on page D2-1663.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**LBN, bits [19:16]**

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of DBGBCR<n>_EL1.E is 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**SSC, bits [15:14]**

Security state control. Determines the Security states under which a breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-1664.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to its defined reset value on Cold reset.

**HMC, bit [13]**

Higher mode control. Determines the debug perspective for deciding when a breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the SSC, bits [15:14] description.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [12:9]**

Reserved, RES0.

**BAS, bits [8:5]**

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state. In an AArch64-only implementation, this field is reserved, RES1. Otherwise:

- BAS[2] and BAS[0] are read/write.
- BAS[3] and BAS[1] are read-only copies of BAS[2] and BAS[0] respectively.

The values `0011` and `1100` are only supported if AArch32 is supported at any Exception level.

The permitted values depend on the breakpoint type.

For Address match breakpoints in either AArch32 or AArch64 state, the permitted values are:

| BAS | Match instruction at | Constraint for debuggers |
|---|---|---|
| `0011` | DBGBVR<n>_EL1 | Use for T32 instructions. |
| `1100` | DBGBVR<n>_EL1+2 | Use for T32 instructions. |
| `1111` | DBGBVR<n>_EL1 | Use for A64 and A32 instructions. |

`0000` is reserved and must behave as if the breakpoint is disabled or map to a permitted value.

For more information, see *Using the BAS field in Address Match breakpoints* on page G2-3909.

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

| BAS | Step instruction at | Constraint for debuggers |
|---|---|---|
| `0000` | - | Use for a match anywhere breakpoint. |
| `0011` | DBGBVR<n>_EL1 | Use for stepping T32 instructions. |
| `1100` | DBGBVR<n>_EL1+2 | Use for stepping T32 instructions. |
| `1111` | DBGBVR<n>_EL1 | Use for stepping A32 instructions. |

For more information, see *Using the BAS field in Address Match breakpoints* on page G2-3909.

For Context matching breakpoints, this field is RES1 and ignored.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [4:3]**

Reserved, RES0.

**PMC, bits [2:1]**

Privilege mode control. Determines the Exception level or levels at which a breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>_EL1.SSC description.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**E, bit [0]**

Enable breakpoint DBGBVR<n>_EL1. Possible values are:

`0` Breakpoint disabled.

`1` Breakpoint enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

## Accessing the DBGBCR<n>_EL1:

To access the DBGBCR<n>_EL1:

```
MRS <Xt>, DBGBCR<n>_EL1 ; Read DBGBCR<n>_EL1 into Xt, where n is in the range 0 to 15
MSR DBGBCR<n>_EL1, <Xt> ; Write Xt to DBGBCR<n>_EL1, where n is in the range 0 to 15
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0000 | n<3:0> | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n>_EL1 characteristics are:

**Purpose**

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register DBGBCR<n>_EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 and *Software Access debug event* on page H3-4881.

Subject to the prioritization rules:

If EDSCR.TDA==1, OSLSR.OSLK==0, and halting is allowed, EL1, EL2, and EL3 accesses to this register generate a Software Access debug event.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register DBGBVR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGBVR<n>.

AArch64 System register DBGBVR<n>_EL1[63:32] is architecturally mapped to AArch32 System register DBGBXVR<n>.

AArch64 System register DBGBVR<n>_EL1 is architecturally mapped to External register DBGBVR<n>_EL1.

If breakpoint n is not implemented then this register is unallocated.

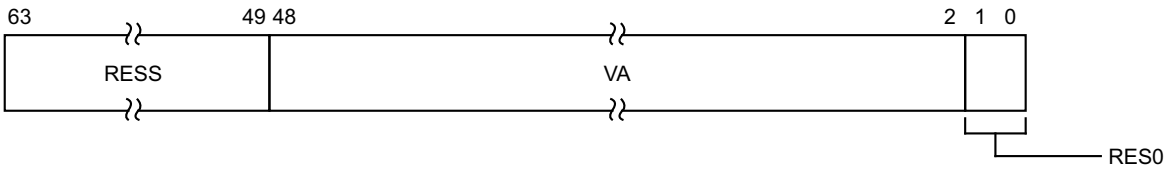
**Attributes**

DBGBVR<n>_EL1 is a 64-bit register.

#### Field descriptions

The DBGBVR<n>_EL1 bit assignments are:

***When DBGBCR<n>_EL1.BT==0b0x0x:***




**RESS, bits [63:49]**

Reserved, Sign extended. Hardware and software must treat this field as RES0 if bit[48] is 0 or RES0, and as RES1 if bit[48] is 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of bit [48], meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

**VA, bits [48:2]**

Bits[48:2] of the address value for comparison.

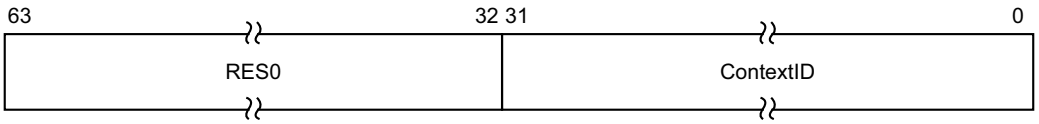
When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [1:0]**

Reserved, RES0.

***When DBGBCR<n>_EL1.BT==0b0x1x:***




**Bits [63:32]**

Reserved, RES0.

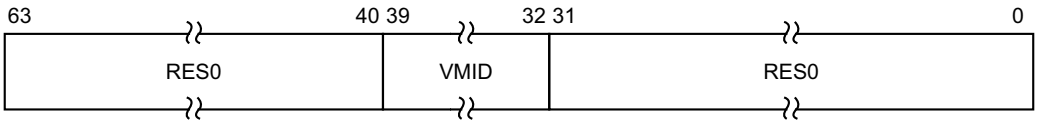
**ContextID, bits [31:0]**

Context ID value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

***When DBGBCR<n>_EL1.BT==0b1x0x and EL2 implemented:***




**Bits [63:40]**

Reserved, RES0.

**VMID, bits [39:32]**

VMID value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

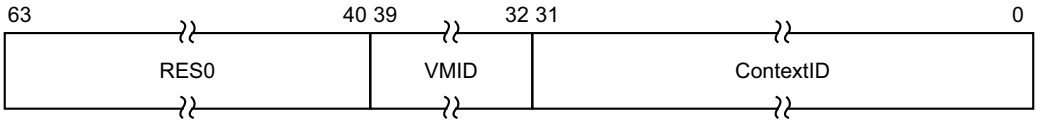
This field resets to its defined reset value on Cold reset.

**Bits [31:0]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

***When DBGBCR<n>_EL1.BT==0x1x1x and EL2 implemented:***




**Bits [63:40]**

Reserved, RES0.

**VMID, bits [39:32]**

VMID value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**ContextID, bits [31:0]**

Context ID value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

## Accessing the DBGBVR<n>_EL1:

To access the DBGBVR<n>_EL1:

```
MRS <Xt>, DBGBVR<n>_EL1 ; Read DBGBVR<n>_EL1 into Xt, where n is in the range 0 to 15
MSR DBGBVR<n>_EL1, <Xt> ; Write Xt to DBGBVR<n>_EL1, where n is in the range 0 to 15
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0000 | n<3:0> | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.4 DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register

The DBGCLAIMCLR_EL1 characteristics are:

**Purpose**

Used by software to read the values of the CLAIM tag bits, and to clear these bits to 0.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

The architecture does not define any functionality for the CLAIM tag bits.

Used in conjunction with the DBGCLAIMSET_EL1 register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register DBGCLAIMCLR_EL1 is architecturally mapped to AArch32 System register DBGCLAIMCLR.

AArch64 System register DBGCLAIMCLR_EL1 is architecturally mapped to External register DBGCLAIMCLR_EL1.

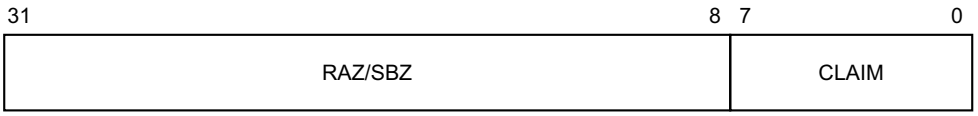
An implementation must include 8 CLAIM tag bits.

**Attributes**

DBGCLAIMCLR_EL1 is a 32-bit register.

#### Field descriptions

The DBGCLAIMCLR_EL1 bit assignments are:




**Bits [31:8]**

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

**CLAIM, bits [7:0]**

Read or clear CLAIM tag bits. Reading this field returns the current value of the CLAIM tag bits.

Writing a 1 to one of these bits clears the corresponding CLAIM tag bit to 0. This is an indirect write to the CLAIM tag bits. A single write operation can clear multiple CLAIM tag bits to 0.

Writing 0 to one of these bits has no effect.

A cold reset clears the CLAIM tag bits to 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the DBGCLAIMCLR_EL1:

To access the DBGCLAIMCLR_EL1:

```
MRS <Xt>, DBGCLAIMCLR_EL1 ; Read DBGCLAIMCLR_EL1 into Xt
MSR DBGCLAIMCLR_EL1, <Xt> ; Write Xt to DBGCLAIMCLR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0111 | 1001 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.5 DBGCLAIMSET_EL1, Debug CLAIM Tag Set register

The DBGCLAIMSET_EL1 characteristics are:

**Purpose**

Used by software to set the CLAIM tag bits to 1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

The architecture does not define any functionality for the CLAIM tag bits.

Used in conjunction with the DBGCLAIMCLR_EL1 register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register DBGCLAIMSET_EL1 is architecturally mapped to AArch32 System register DBGCLAIMSET.

AArch64 System register DBGCLAIMSET_EL1 is architecturally mapped to External register DBGCLAIMSET_EL1.

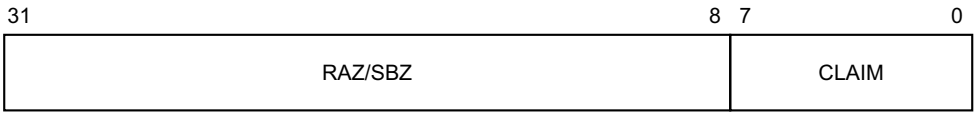
An implementation must include 8 CLAIM tag bits.

**Attributes**

DBGCLAIMSET_EL1 is a 32-bit register.

#### Field descriptions

The DBGCLAIMSET_EL1 bit assignments are:




**Bits [31:8]**

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

**CLAIM, bits [7:0]**

Set CLAIM tag bits. RAO.

Writing a 1 to one of these bits sets the corresponding CLAIM tag bit to 1. This is an indirect write to the CLAIM tag bits. A single write operation can set multiple CLAIM tag bits to 1.

Writing 0 to one of these bits has no effect.

A cold reset clears the CLAIM tag bits to 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the DBGCLAIMSET_EL1:

To access the DBGCLAIMSET_EL1:

```
MRS <Xt>, DBGCLAIMSET_EL1 ; Read DBGCLAIMSET_EL1 into Xt
MSR DBGCLAIMSET_EL1, <Xt> ; Write Xt to DBGCLAIMSET_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0111 | 1000 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.6 DBGDTR_EL0, Debug Data Transfer Register, half-duplex

The DBGDTR_EL0 characteristics are:

**Purpose**

Transfers 64 bits of data between the PE and an external debugger. Can transfer both ways using only a single register.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If MDSCR_EL1.TDCC==1, EL0 accesses to this register are trapped to EL1.

**Configurations**

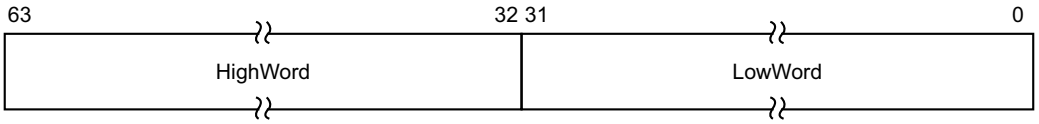
There are no configuration notes.

**Attributes**

DBGDTR_EL0 is a 64-bit register.

#### Field descriptions

The DBGDTR_EL0 bit assignments are:




**HighWord, bits [63:32]**

Writes to this register set DTRRX to the value in this field and do not change RXfull.

Reads from this register return the value of DTRTX and do not change TXfull.

**LowWord, bits [31:0]**

Writes to this register set DTRTX to the value in this field and set TXfull to 1.

Reads from this register return the value of DTRRX and clear RXfull to 0.

#### Accessing the DBGDTR_EL0:

To access the DBGDTR_EL0:

```
MRS <Xt>, DBGDTR_EL0 ; Read DBGDTR_EL0 into Xt
MSR DBGDTR_EL0, <Xt> ; Write Xt to DBGDTR_EL0
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 011 | 0000 | 0100 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.7 DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

**Purpose**

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. It is a component of the Debug Communications Channel.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL0, EL1, and EL2 read accesses to this register are trapped to EL3.

If MDSCR_EL1.TDCC==1, EL0 read accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register DBGDTRRX_EL0 is architecturally mapped to AArch32 System register DBGDTRRXint.

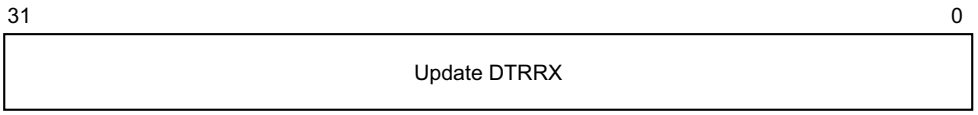
AArch64 System register DBGDTRRX_EL0 is architecturally mapped to External register DBGDTRRX_EL0.

**Attributes**

DBGDTRRX_EL0 is a 32-bit register.

#### Field descriptions

The DBGDTRRX_EL0 bit assignments are:




**Bits [31:0]**

Update DTRRX.

If RXfull is set to 1, then reads of this register return the last value written to DTRRX and clear RXfull to 0.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the DBGDTRRX_EL0:

To access the DBGDTRRX_EL0:

```
MRS <Xt>, DBGDTRRX_EL0 ; Read DBGDTRRX_EL0 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 011 | 0000 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.8 DBGDTRTX_EL0, Debug Data Transfer Register, Transmit

The DBGDTRTX_EL0 characteristics are:

**Purpose**

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. It is a component of the Debug Communication Channel.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-WO | WO | WO | WO | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL0 and EL1 write accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL0, EL1, and EL2 write accesses to this register are trapped to EL3.

If MDSCR_EL1.TDCC==1, EL0 write accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register DBGDTRTX_EL0 is architecturally mapped to AArch32 System register DBGDTRTXint.

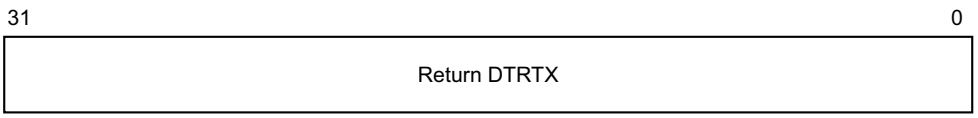
AArch64 System register DBGDTRTX_EL0 is architecturally mapped to External register DBGDTRTX_EL0.

**Attributes**

DBGDTRTX_EL0 is a 32-bit register.

#### Field descriptions

The DBGDTRTX_EL0 bit assignments are:




**Bits [31:0]**

Return DTRTX.

If TXfull is set to 0, then writes of this register update the value in DTRTX and set TXfull to 1.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the DBGDTRTX_EL0:

To access the DBGDTRTX_EL0:

```
MSR DBGDTRTX_EL0, <Xt> ; Write Xt to DBGDTRTX_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 011 | 0000 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.9 DBGPRCR_EL1, Debug Power Control Register

The DBGPRCR_EL1 characteristics are:

**Purpose**

Controls behavior of the PE on powerdown request.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDOSA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDOSA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register DBGPRCR_EL1 is architecturally mapped to AArch32 System register DBGPRCR.

Bit [0] of this register is mapped to EDPRCR.CORENPDRQ, bit [0] of the external view of this register.

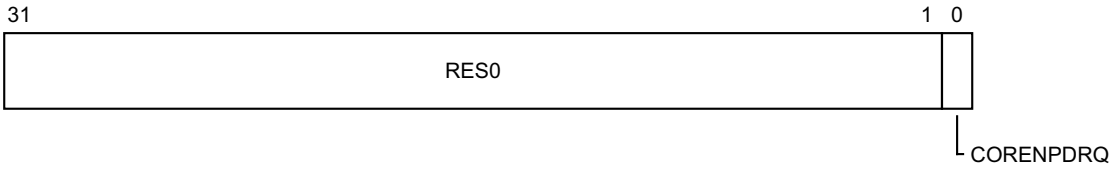
The other bits in these registers are not mapped to each other.

**Attributes**

DBGPRCR_EL1 is a 32-bit register.

#### Field descriptions

The DBGPRCR_EL1 bit assignments are:




**Bits [31:1]**

Reserved, RES0.

**CORENPDRQ, bit [0]**

Core no powerdown request. Requests emulation of powerdown. Possible values of this bit are:

`0` On a powerdown request, the system powers down the Core power domain.

`1` On a powerdown request, the system emulates powerdown of the Core power domain. In this emulation mode the Core power domain is not actually powered down.

Writes to this bit are permitted regardless of the state of the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request Core no powerdown regardless of whether invasive debug is permitted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to the value of EDPRCR.COREPURQ.

This field resets to its defined reset value on Cold reset.

### Accessing the DBGPRCR_EL1:

To access the DBGPRCR_EL1:

```
MRS <Xt>, DBGPRCR_EL1 ; Read DBGPRCR_EL1 into Xt
MSR DBGPRCR_EL1, <Xt> ; Write Xt to DBGPRCR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0001 | 0100 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.10 DBGVCR32_EL2, Debug Vector Catch Register

The DBGVCR32_EL2 characteristics are:

**Purpose**

Allows access to the AArch32 register DBGVCR from AArch64 state only. Its value has no effect on execution in AArch64 state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register DBGVCR32_EL2 is architecturally mapped to AArch32 System register DBGVCR.

If EL1 does not support AArch32, this register is UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

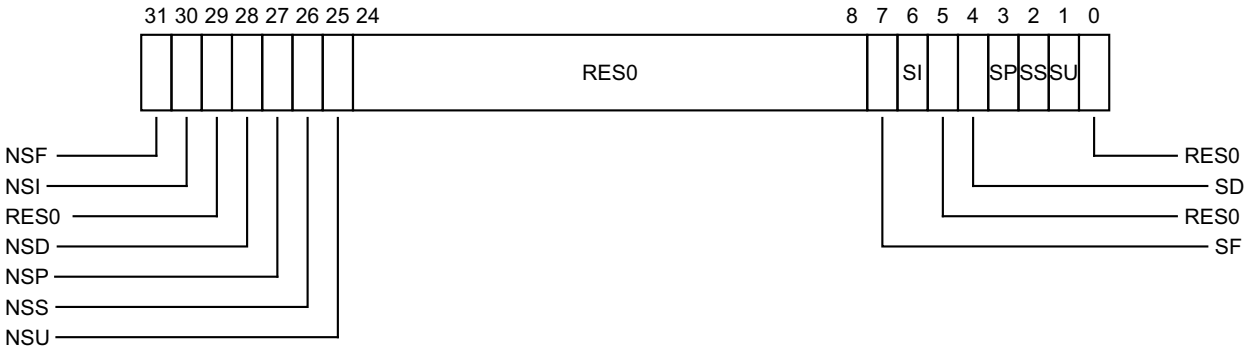
**Attributes**

DBGVCR32_EL2 is a 32-bit register.

#### Field descriptions

The DBGVCR32_EL2 bit assignments are:

***When EL3 implemented and using AArch64:***




**NSF, bit [31]**

FIQ vector catch enable in Non-secure state.

The exception vector offset is `0x1C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**NSI, bit [30]**

IRQ vector catch enable in Non-secure state.

The exception vector offset is `0x18`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [29]**

Reserved, RES0.

**NSD, bit [28]**

Data Abort vector catch enable in Non-secure state.

The exception vector offset is `0x10`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**NSP, bit [27]**

Prefetch Abort vector catch enable in Non-secure state.

The exception vector offset is `0x0C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**NSS, bit [26]**

Supervisor Call (SVC) vector catch enable in Non-secure state.

The exception vector offset is `0x08`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**NSU, bit [25]**

Undefined Instruction vector catch enable in Non-secure state.

The exception vector offset is `0x04`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bits [24:8]**

Reserved, RES0.

**SF, bit [7]**

FIQ vector catch enable in Secure state.

The exception vector offset is `0x1C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SI, bit [6]**

IRQ vector catch enable in Secure state.

The exception vector offset is `0x18`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [5]**

Reserved, RES0.

**SD, bit [4]**

Data Abort vector catch enable in Secure state.

The exception vector offset is `0x10`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SP, bit [3]**

Prefetch Abort vector catch enable in Secure state.

The exception vector offset is `0x0C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SS, bit [2]**

Supervisor Call (SVC) vector catch enable in Secure state.

The exception vector offset is `0x08`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SU, bit [1]**

Undefined Instruction vector catch enable in Secure state.

The exception vector offset is `0x04`.

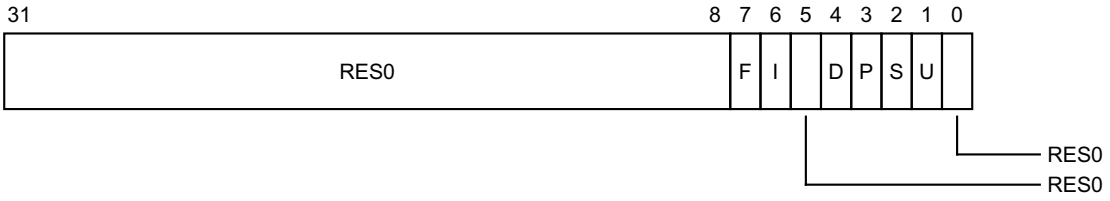
When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [0]**

Reserved, RES0.

***When EL3 not implemented:***




**Bits [31:8]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**F, bit [7]**

FIQ vector catch enable.

The exception vector offset is `0x1C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**I, bit [6]**

IRQ vector catch enable.

The exception vector offset is `0x18`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [5]**

Reserved, RES0.

**D, bit [4]**

Data Abort vector catch enable.

The exception vector offset is `0x10`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**P, bit [3]**

Prefetch Abort vector catch enable.

The exception vector offset `0x0C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**S, bit [2]**

Supervisor Call (SVC) vector catch enable.

The exception vector offset is `0x08`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**U, bit [1]**

Undefined Instruction vector catch enable.

The exception vector offset is `0x04`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [0]**

Reserved, RES0.

### Accessing the DBGVCR32_EL2:

To access the DBGVCR32_EL2:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRS <Xt>, DBGVCR32_EL2 ; Read DBGVCR32_EL2 into Xt
MSR DBGVCR32_EL2, <Xt> ; Write Xt to DBGVCR32_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 100 | 0000 | 0111 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.11 DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15

The DBGWCR<n>_EL1 characteristics are:

**Purpose**

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

When the E field is zero, all the other fields in the register are ignored.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 and *Software Access debug event* on page H3-4881.

Subject to the prioritization rules:

If EDSCR.TDA==1, OSLSR.OSLK==0, and halting is allowed, EL1, EL2, and EL3 accesses to this register generate a Software Access debug event.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register DBGWCR<n>_EL1 is architecturally mapped to AArch32 System register DBGWCR<n>.

AArch64 System register DBGWCR<n>_EL1 is architecturally mapped to External register DBGWCR<n>_EL1.

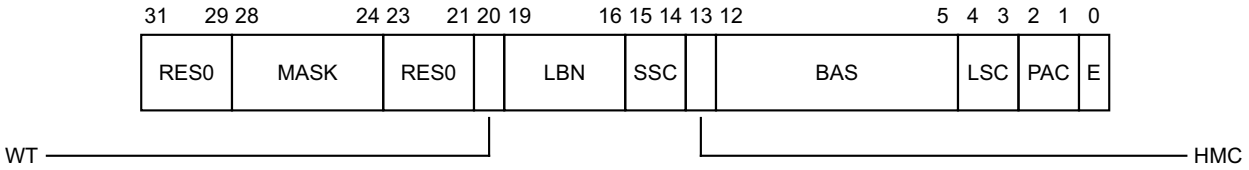
If breakpoint n is not implemented then this register is unallocated.

**Attributes**

DBGWCR<n>_EL1 is a 32-bit register.

#### Field descriptions

The DBGWCR<n>_EL1 bit assignments are:




**Bits [31:29]**

Reserved, RES0.

**MASK, bits [28:24]**

Address mask. Only objects up to 2GB can be watched using a single mask.

`00000` No mask.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`00001` Reserved.

`00010` Reserved.

If programmed with a reserved value, a watchpoint must behave as if either:

- MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCRn_EL1.
- The watchpoint is disabled.

Software must not rely on this property because the behavior of reserved values might change in a future revision of the architecture.

Other values mask the corresponding number of address bits, from `0b00011` masking 3 address bits (`0x00000007` mask for address) to `0b11111` masking 31 address bits (`0x7FFFFFFF` mask for address).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [23:21]**

Reserved, RES0.

**WT, bit [20]**

Watchpoint type. Possible values are:

`0` Unlinked data address match.

`1` Linked data address match.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**LBN, bits [19:16]**

Linked breakpoint number. For Linked data address watchpoints, this specifies the index of the Context-matching breakpoint linked to.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**SSC, bits [15:14]**

Security state control. Determines the Security states under which a watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**HMC, bit [13]**

Higher mode control. Determines the debug perspective for deciding when a watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**BAS, bits [12:5]**

Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n>_EL1 is being watched.

| BAS | Description |
|---|---|
| xxxxxxx1 | Match byte at DBGWVR<n>_EL1 |
| xxxxxx1x | Match byte at DBGWVR<n>_EL1+1 |
| xxxxx1xx | Match byte at DBGWVR<n>_EL1+2 |
| xxxx1xxx | Match byte at DBGWVR<n>_EL1+3 |

In cases where DBGWVR<n>_EL1 addresses a double-word:

| BAS | Description, if DBGWVR<n>_EL1[2] == 0 |
|---|---|
| xxx1xxxx | Match byte at DBGWVR<n>_EL1+4 |
| xx1xxxxx | Match byte at DBGWVR<n>_EL1+5 |
| x1xxxxxx | Match byte at DBGWVR<n>_EL1+6 |
| 1xxxxxxx | Match byte at DBGWVR<n>_EL1+7 |

If DBGWVR<n>_EL1[2] == 1, only BAS[3:0] are used and BAS[7:4] are ignored. ARM deprecates setting DBGWVR<n>_EL1[2] == 1.

The valid values for BAS are `0b0000000`, or a binary number all of whose set bits are contiguous. All other values are reserved and must not be used by software.

If BAS is zero, no bytes are watched by this watchpoint.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**LSC, bits [4:3]**

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

`01` Match instructions that load from a watchpointed address.

`10` Match instructions that store to a watchpointed address.

`11` Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**PAC, bits [2:1]**

Privilege of access control. Determines the Exception level or levels at which a watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to its defined reset value on Cold reset.

**E, bit [0]**

Enable watchpoint n. Possible values are:

`0` Watchpoint disabled.

`1` Watchpoint enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

### Accessing the DBGWCR<n>_EL1:

To access the DBGWCR<n>_EL1:

```
MRS <Xt>, DBGWCR<n>_EL1 ; Read DBGWCR<n>_EL1 into Xt, where n is in the range 0 to 15
MSR DBGWCR<n>_EL1, <Xt> ; Write Xt to DBGWCR<n>_EL1, where n is in the range 0 to 15
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0000 | n<3:0> | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.12 DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15

The DBGWVR<n>_EL1 characteristics are:

**Purpose**

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>_EL1.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 and *Software Access debug event* on page H3-4881.

Subject to the prioritization rules:

If EDSCR.TDA==1, OSLSR.OSLK==0, and halting is allowed, EL1, EL2, and EL3 accesses to this register generate a Software Access debug event.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register DBGWVR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGWVR<n>.

AArch64 System register DBGWVR<n>_EL1 is architecturally mapped to External register DBGWVR<n>_EL1.

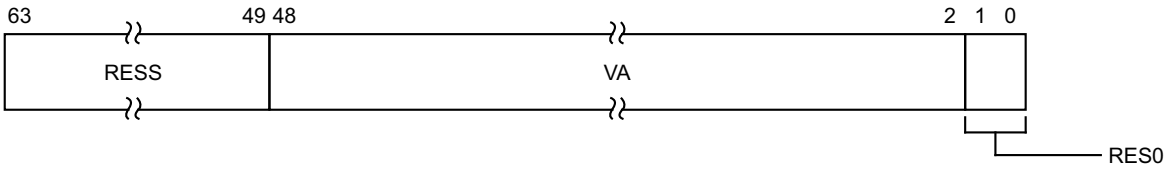
If breakpoint n is not implemented then this register is unallocated.

**Attributes**

DBGWVR<n>_EL1 is a 64-bit register.

#### Field descriptions

The DBGWVR<n>_EL1 bit assignments are:




**RESS, bits [63:49]**

Reserved, Sign extended. Hardware and software must treat this field as RES0 if bit[48] is 0 or RES0, and as RES1 if bit[48] is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of bit [48], meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

**VA, bits [48:2]**

Bits[48:2] of the address value for comparison.

ARM deprecates setting DBGWVR<n>_EL1[2] == 1.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [1:0]**

Reserved, RES0.

## Accessing the DBGWVR<n>_EL1:

To access the DBGWVR<n>_EL1:

```
MRS <Xt>, DBGWVR<n>_EL1 ; Read DBGWVR<n>_EL1 into Xt, where n is in the range 0 to 15
MSR DBGWVR<n>_EL1, <Xt> ; Write Xt to DBGWVR<n>_EL1, where n is in the range 0 to 15
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0000 | n<3:0> | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.13 DLR_EL0, Debug Link Register

The DLR_EL0 characteristics are:

**Purpose**

In Debug state, holds the address to restart from.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | RW | RW | RW | RW | RW |

Access to this register is from Debug state only. During normal execution this register is unallocated.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

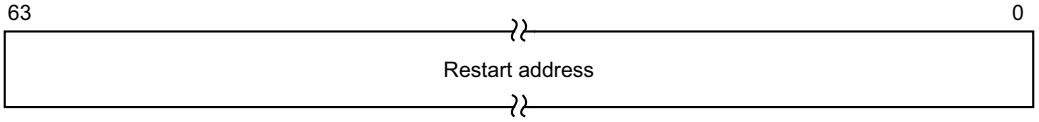
AArch64 System register DLR_EL0[31:0] is architecturally mapped to AArch32 System register DLR.

**Attributes**

DLR_EL0 is a 64-bit register.

#### Field descriptions

The DLR_EL0 bit assignments are:




**Bits [63:0]**

Restart address.

#### Accessing the DLR_EL0:

To access the DLR_EL0:

```
MRS <Xt>, DLR_EL0 ; Read DLR_EL0 into Xt
MSR DLR_EL0, <Xt> ; Write Xt to DLR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 0100 | 0101 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.14 DSPSR_EL0, Debug Saved Program Status Register

The DSPSR_EL0 characteristics are:

**Purpose**

Holds the saved process state on entry to Debug state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | RW | RW | RW | RW | RW |

Access to this register is from Debug state only. During normal execution this register is unallocated.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register DSPSR_EL0 is architecturally mapped to AArch32 System register DSPSR.

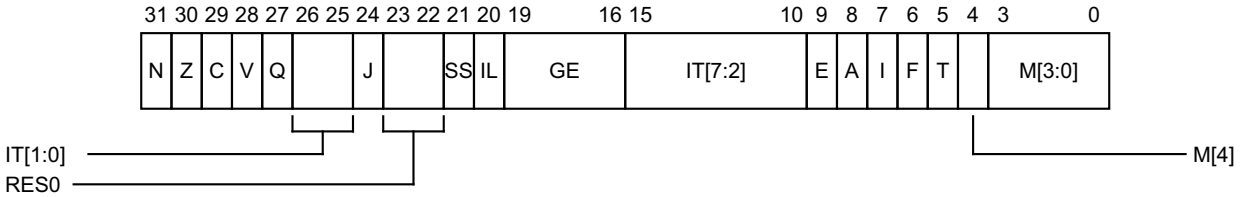
**Attributes**

DSPSR_EL0 is a 32-bit register.

#### Field descriptions

The DSPSR_EL0 bit assignments are:

***When exiting Debug state to AArch32:***




**N, bit [31]**

Copied to CPSR.N on exiting Debug state.

**Z, bit [30]**

Copied to CPSR.Z on exiting Debug state.

**C, bit [29]**

Copied to CPSR.C on exiting Debug state.

**V, bit [28]**

Copied to CPSR.V on exiting Debug state.

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:22]**

Reserved, RES0.

**SS, bit [21]**

Software step. Shows the value of PSTATE.SS immediately before Debug state was entered.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before Debug state was entered.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the Debug state entry was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that Debug state was entered from. Possible values of this bit are:

1 Exception taken from AArch32.

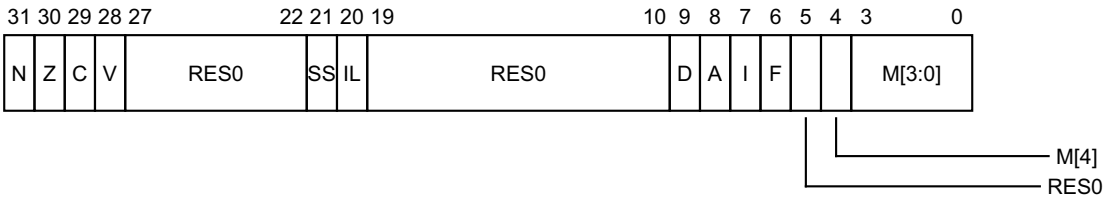
**M[3:0], bits [3:0]**

AArch32 mode that Debug state was entered from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor |
| 0b0111 | Abort |
| 0b1010 | Hyp |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

***When entering Debug state from AArch64 and exiting Debug state to AArch64:***




**N, bit [31]**

Set to the value of the N condition flag on entering Debug state, and copied to the N condition flag on exiting Debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Z, bit [30]**

Set to the value of the Z condition flag on entering Debug state, and copied to the Z condition flag on exiting Debug state.

**C, bit [29]**

Set to the value of the C condition flag on entering Debug state, and copied to the C condition flag on exiting Debug state.

**V, bit [28]**

Set to the value of the V condition flag on entering Debug state, and copied to the V condition flag on exiting Debug state.

**Bits [27:22]**

Reserved, RES0.

**SS, bit [21]**

Software step. Shows the value of PSTATE.SS immediately before Debug state was entered.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before Debug state was entered.

**Bits [19:10]**

Reserved, RES0.

**D, bit [9]**

Process state D mask. The possible values of this bit are:

`0` Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current Exception level are not masked.

`1` Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

**A, bit [8]**

SError (System Error) mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**Bit [5]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**M[4], bit [4]**

Execution state that Debug state was entered from. Possible values of this bit are:

`0` Exception taken from AArch64.

**M[3:0], bits [3:0]**

AArch64 mode that Debug state was entered from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | EL0t |
| 0b0100 | EL1t |
| 0b0101 | EL1h |
| 0b1000 | EL2t |
| 0b1001 | EL2h |
| 0b1100 | EL3t |
| 0b1101 | EL3h |

Other values are reserved, and returning to an Exception level that is using AArch64 with a reserved value in this field is treated as an illegal exception return.

The bits in this field are interpreted as follows:

- M[3:2] holds the Exception Level.
- M[1] is unused and is RES0 for all non-reserved values.
- M[0] is used to select the SP:
  - — 0 means the SP is always SP0.
  - — 1 means the exception SP is determined by the EL.

### Accessing the DSPSR_EL0:

To access the DSPSR_EL0:

```
MRS <Xt>, DSPSR_EL0 ; Read DSPSR_EL0 into Xt
MSR DSPSR_EL0, <Xt> ; Write Xt to DSPSR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 0100 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.15 MDCCINT_EL1, Monitor DCC Interrupt Enable Register

The MDCCINT_EL1 characteristics are:

**Purpose**

Enables interrupt requests to be signaled based on the DCC status flags.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

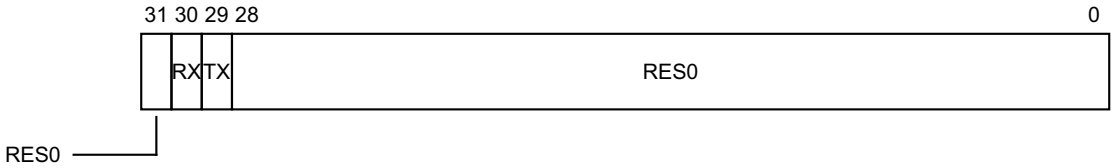
AArch64 System register MDCCINT_EL1 is architecturally mapped to AArch32 System register DBGDCCINT.

**Attributes**

MDCCINT_EL1 is a 32-bit register.

#### Field descriptions

The MDCCINT_EL1 bit assignments are:




**Bit [31]**

Reserved, RES0.

**RX, bit [30]**

DCC interrupt request enable control for DTRRX. Enables a common COMMIRQ interrupt request to be signaled based on the DCC status flags.

`0` No interrupt request generated by DTRRX.

`1` Interrupt request will be generated on RXfull == 1.

If legacy COMMRX and COMMTX signals are implemented, then these are not affected by the value of this bit.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TX, bit [29]**

DCC interrupt request enable control for DTRTX. Enables a common COMMIRQ interrupt request to be signaled based on the DCC status flags.

`0` No interrupt request generated by DTRTX.

`1` Interrupt request will be generated on TXfull == 0.

If legacy COMMRX and COMMTX signals are implemented, then these are not affected by the value of this bit.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

**Bits [28:0]**

Reserved, RES0.

## Accessing the MDCCINT_EL1:

To access the MDCCINT_EL1:

```
MRS <Xt>, MDCCINT_EL1 ; Read MDCCINT_EL1 into Xt
MSR MDCCINT_EL1, <Xt> ; Write Xt to MDCCINT_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0000 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.16 MDCCSR_EL0, Monitor DCC Status Register

The MDCCSR_EL0 characteristics are:

**Purpose**

Main control register for the debug implementation, containing flow-control flags for the DCC. This is an internal, read-only view.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL0, EL1, and EL2 read accesses to this register are trapped to EL3.

If MDSCR_EL1.TDCC==1, EL0 read accesses to this register are trapped to EL1.

**Configurations**

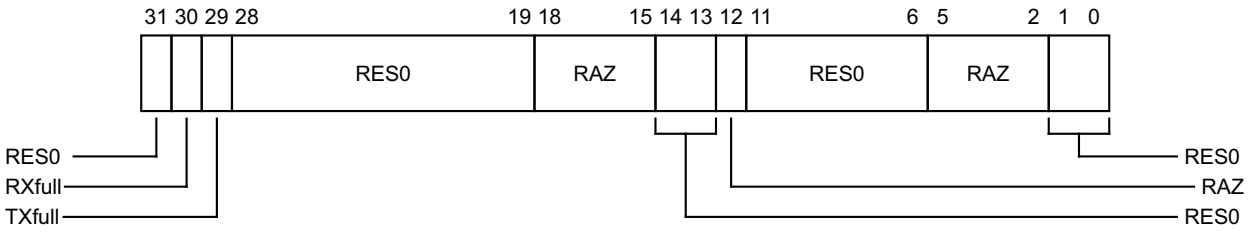
AArch64 System register MDCCSR_EL0 is architecturally mapped to AArch32 System register DBGDSCRint.

**Attributes**

MDCCSR_EL0 is a 32-bit register.

#### Field descriptions

The MDCCSR_EL0 bit assignments are:




**Bit [31]**

Reserved, RES0.

**RXfull, bit [30]**

DTRRX full. Read-only view of the equivalent bit in the EDSCR.

**TXfull, bit [29]**

DTRTX full. Read-only view of the equivalent bit in the EDSCR.

**Bits [28:19]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [18:15]**

Reserved, RAZ. Hardware must implement this field as RAZ. Software must not rely on this field being RAZ.

**Bits [14:13]**

Reserved, RES0.

**Bit [12]**

Reserved, RAZ. Hardware must implement this field as RAZ. Software must not rely on this field being RAZ.

**Bits [11:6]**

Reserved, RES0.

**Bits [5:2]**

Reserved, RAZ. Hardware must implement this field as RAZ. Software must not rely on this field being RAZ.

**Bits [1:0]**

Reserved, RES0.

### Accessing the MDCCSR_EL0:

To access the MDCCSR_EL0:

```
MRS <Xt>, MDCCSR_EL0 ; Read MDCCSR_EL0 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 011 | 0000 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.17 MDCR_EL2, Monitor Debug Configuration Register (EL2)

The MDCR_EL2 characteristics are:

**Purpose**

Provides EL2 configuration options for self-hosted debug and the Performance Monitors extension.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL3.TDA==1, EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register MDCR_EL2 is architecturally mapped to AArch32 System register HDCR.

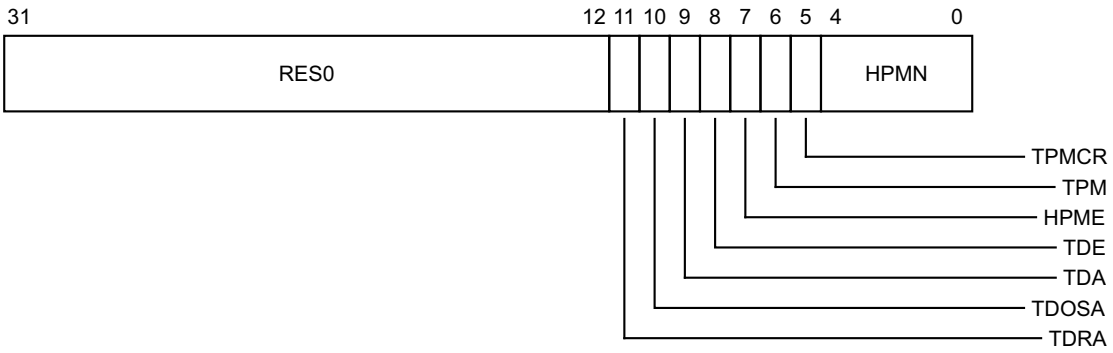
If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

MDCR_EL2 is a 32-bit register.

#### Field descriptions

The MDCR_EL2 bit assignments are:




**Bits [31:12]**

Reserved, RES0.

**TDRA, bit [11]**

Trap Debug ROM Address register access. Traps Non-secure System register accesses to the Debug ROM registers to EL2. This trap is from:

- Non-secure EL0 using AArch32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Non-secure EL1, regardless of which Execution state it is using.

0 Non-secure EL0 and EL1 System register accesses to the Debug ROM registers are not trapped to EL2.

1 Non-secure EL0 and EL1 System register accesses to the Debug ROM registers are trapped to EL2.

The registers for which accesses are trapped are as follows:

AArch64: MDRAR_EL1.

AArch32: DBGDRAR, DBGDSAR.

If MDCR_EL2.TDE == 1 or HCR_EL2.TGE == 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**TDOSA, bit [10]**

Trap debug OS-related register access. Traps Non-secure EL1 System register accesses to the powerdown debug registers to EL2, from both Execution states:

0 Non-secure EL1 System register accesses to the powerdown debug registers are not trapped to EL2.

1 Non-secure EL1 System register accesses to the powerdown debug registers are trapped to EL2.

The registers for which accesses are trapped are as follows:

AArch64: OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, and the DBGPRCR_EL1.

AArch32: DBGOSLSR, DBGOSLAR, DBGOSDLR, and the DBGPRCR.

AArch64 and AArch32:

- Any IMPLEMENTATION DEFINED integration registers.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

**Note**

These registers are not accessible at EL0.

If MDCR_EL2.TDE == 1 or HCR_EL2.TGE == 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**TDA, bit [9]**

Trap Debug Access. Traps Non-secure EL0 and EL1 System register accesses to those debug System registers that are not trapped by either of the following:

- MDCR_EL2.TDRA.
- MDCR_EL2.TDOSA.

0 Has no effect on System register accesses to the debug registers.

1 Non-secure EL0 or EL1 System register accesses to the debug registers, other than the registers trapped by MDCR_EL2.TDRA and MDCR_EL2.TDOSA, are trapped to EL2, from both Execution states.

MDCR_EL2.TDA does not trap accesses to the DBGDTRRX_EL0, DBGDTRTX_EL0, or DBGDTR_EL0 when the PE is in Debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If MDCR_EL2.TDE == 1 or HCR_EL2.TGE == 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**TDE, bit [8]**

Route Software debug exceptions from Non-secure EL1 and EL0 to EL2.

0 Has no effect on Debug exceptions.

1 Route Non-secure Debug exceptions from Non-secure EL1 and EL0 to EL2.

If HCR_EL2.TGE == 1, then this bit is ignored and treated as though it is 1 other than for the value read back from MDCR_EL2.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**HPME, bit [7]**

Hypervisor Performance Monitors Enable. The possible values of this bit are:

0 EL2 Performance Monitors disabled.

1 EL2 Performance Monitors enabled.

When the value of this bit is 1, the Performance Monitors counters that are reserved for use from EL2 or Secure state are enabled. For more information see the description of the HPMN field.

If the Performance Monitors extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**TPM, bit [6]**

Trap Performance Monitors accesses. Traps Non-secure EL0 and EL1 accesses to all Performance Monitors registers to EL2, from both Execution states:

0 Non-secure EL0 and EL1 accesses to all Performance Monitors registers are not trapped to EL2.

1 Non-secure EL0 and EL1 accesses to all Performance Monitors registers are trapped to EL2.

**Note**

- EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

If the Performance Monitors extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**TPMCR, bit [5]**

Trap PMCR_EL0 or PMCR accesses. Traps Non-secure EL0 and EL1 accesses to the PMCR_EL0 or PMCR to EL2.

0 Non-secure EL0 and EL1 accesses to the PMCR_EL0 or PMCR are not trapped to EL2.

1 Non-secure EL0 and EL1 accesses to the PMCR_EL0 or PMCR are trapped to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

If the Performance Monitors extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**HPMN, bits [4:0]**

Defines the number of Performance Monitors counters that are accessible from Non-secure EL0 and EL1 modes.

If the Performance Monitors extension is not implemented, this field is RES0.

In Non-secure state, HPMN divides the Performance Monitors counters as follows. For counter n in Non-secure state:

- If n is in the range 0<=n<HPMN, the counter is accessible from EL1 and EL2, and from EL0 if permitted by PMUSERENR_EL0. PMCR_EL0.E enables the operation of counters in this range.
- If n is in the range HPMN<=n<PMCR_EL0.N, the counter is accessible only from EL2. MDCR_EL2.HPME enables the operation of counters in this range.

If this field is set to 0, or to a value larger than PMCR_EL0.N, then the following CONSTRAINED UNPREDICTABLE behavior applies:

- The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if MDCR_EL2.HPMN is set to an UNKNOWN non-zero value less than PMCR_EL0.N.
  - All counters are reserved for EL2 use, meaning no counters are accessible from Non-secure EL1 and Non-secure EL0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to the value of PMCR_EL0.N.

This field resets to its defined reset value on Warm reset.

## Accessing the MDCR_EL2:

To access the MDCR_EL2:

```
MRS <Xt>, MDCR_EL2 ; Read MDCR_EL2 into Xt
MSR MDCR_EL2, <Xt> ; Write Xt to MDCR_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 0001 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.18 MDCR_EL3, Monitor Debug Configuration Register (EL3)

The MDCR_EL3 characteristics are:

**Purpose**

Provides EL3 configuration options for self-hosted debug and the Performance Monitors extension.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

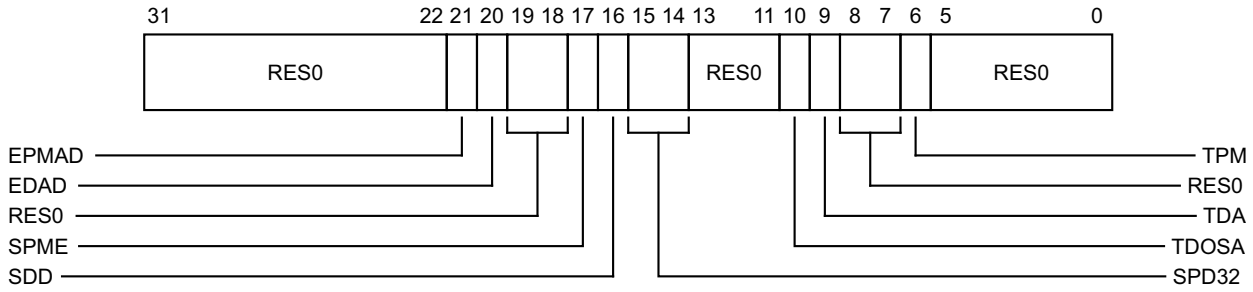
AArch64 System register MDCR_EL3 can be mapped to AArch32 System register SDCR, but this is not architecturally mandated.

**Attributes**

MDCR_EL3 is a 32-bit register.

## Field descriptions

The MDCR_EL3 bit assignments are:




**Bits [31:22]**

Reserved, RES0.

**EPMAD, bit [21]**

External debug interface Performance Monitors registers disable. This disables access to these registers by an external debugger:

0 Access to Performance Monitors registers from external debugger is permitted.

1 Access to Performance Monitors registers from external debugger is disabled, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

If the Performance Monitors extension is not implemented or does not support external debug interface accesses this bit is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EDAD, bit [20]**

External debug interface breakpoint and watchpoint register access disable. This disables access to these registers by an external debugger:

0 Access to breakpoint and watchpoint registers from external debugger is permitted.

1 Access to breakpoint and watchpoint registers from external debugger is disabled, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

**Bits [19:18]**

Reserved, RES0.

**SPME, bit [17]**

Secure Performance Monitors enable. This allows event counting in Secure state:

0 Event counting prohibited in Secure state, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

1 Event counting allowed in Secure state, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

**SDD, bit [16]**

AArch64 Secure self-hosted invasive debug disable. Disables Software debug exceptions in Secure state, other than Software breakpoint instruction.

0 Debug exceptions from Secure EL0 are enabled, and debug exceptions from Secure EL1 are enabled if the value of MDSCR_EL1.KDE is 1 and the value of PSTATE.D is 0.

1 Software debug events, other than software breakpoint instruction debug events, are disabled from all Exception levels in Secure state.

The SDD bit is ignored unless both of the following are true:

- The PE is in Secure state.
- Secure EL1 is using AArch64.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SPD32, bits [15:14]**

AArch32 Secure self-hosted privileged invasive debug control. Enables or disables debug exceptions from Secure state if Secure EL1 is using AArch32, other than Software Breakpoint instructions. Valid values for this field are:

00 Legacy mode. Debug exceptions from Secure EL1 are enabled by the IMPLEMENTATION DEFINED authentication interface.

10 Secure privileged debug disabled. Debug exceptions from Secure EL1 are disabled.

11 Secure privileged debug enabled. Debug exceptions from Secure EL1 are enabled.

Other values are reserved, and have the CONSTRAINED UNPREDICTABLE behavior that they must have the same behavior as 0b00. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

This field has no effect on Software Breakpoint instructions. These are always enabled.

This field is:

- Ignored if either:
  - The PE is in Secure state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Secure EL1 is using AArch64.

- RES0 if the implementation does not support EL1 using AArch32.

If Secure EL1 is using AArch32 then:

- If debug exceptions from Secure EL1 are enabled, then debug exceptions from Secure EL0 are also enabled.
- Otherwise, debug exceptions from Secure EL0 are enabled only if the value of SDER32_EL3.SUIDEN is 1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bits [13:11]**

Reserved, RES0.

**TDOSA, bit [10]**

Trap debug OS-related register access. Traps EL2 and EL1 System register accesses to the powerdown debug registers to EL3, from both Security states and both Execution states:

0 EL2 and EL1 System register accesses to the powerdown debug registers are not trapped to EL3.

1 EL2 and EL1 System register accesses to the powerdown debug registers are trapped to EL3.

The registers for which accesses are trapped are as follows:

AArch64: OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, DBGPRCR_EL1.

AArch32: DBGOSLAR, DBGOSLSR, DBGOSDLR, DBGPRCR.

AArch64 and AArch32:

- Any IMPLEMENTATION DEFINED integration registers.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**TDA, bit [9]**

Trap Debug Access. Traps EL2, EL1, and EL0 System register accesses to those debug System registers that cannot be trapped using the MDCR_EL3.TDOSA field. When MDCR_EL3.TDA is:

0 EL0, EL1, and EL2 accesses to the debug registers, other than the registers that can be trapped by MDCR_EL3.TDOSA, are not trapped to EL3.

1 EL0, EL1, and EL2 accesses to the debug registers, other than the registers that can be trapped by MDCR_EL3.TDOSA, are trapped to EL3, from both Security states and both Execution states.

MDCR_EL3.TDA does not trap accesses to the DBGDTRRX_EL0, DBGDTRTX_EL0, or DBGDTR_EL0 when the PE is in Debug state.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bits [8:7]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TPM, bit [6]**

Trap Performance Monitors accesses. Traps EL2, EL1, and EL0 accesses to all Performance Monitors registers to EL3, from both Security states and both Execution states.

`0` EL2, EL1, and EL0 System register accesses to all Performance Monitors registers are not trapped to EL3.

`1` EL2, EL1, and EL0 System register accesses to all Performance Monitors registers are trapped to EL3.

If the Performance Monitors extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bits [5:0]**

Reserved, RES0.

## Accessing the MDCR_EL3:

To access the MDCR_EL3:

```
MRS <Xt>, MDCR_EL3 ; Read MDCR_EL3 into Xt
MSR MDCR_EL3, <Xt> ; Write Xt to MDCR_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0001 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.19 MDRAR_EL1, Monitor Debug ROM Address Register

The MDRAR_EL1 characteristics are:

**Purpose**

Defines the base physical address of a 4KB-aligned memory-mapped debug component, usually a ROM table that locates and describes the memory-mapped debug components in the system. ARMv8 deprecates any use of this register.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

ARMv8 deprecates any use of this register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDRA==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 read accesses to this register are trapped to EL3.

**Configurations**

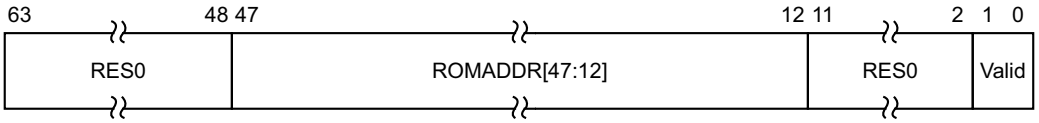
AArch64 System register MDRAR_EL1 is architecturally mapped to AArch32 System register DBGDRAR.

**Attributes**

MDRAR_EL1 is a 64-bit register.

#### Field descriptions

The MDRAR_EL1 bit assignments are:




**Bits [63:48]**

Reserved, RES0.

**ROMADDR[47:12], bits [47:12]**

Bits[47:12] of the ROM table physical address.

If the physical address size in bits (PAsize) is less than 48 then the register bits corresponding to ROMADDR [47:PAsize] are RES0.

Bits [11:0] of the ROM table physical address are zero.

ARM strongly recommends that bits ROMADDR[(PAsize-1):32] are zero in any system that supports AArch32 at the highest implemented Exception level.

**Bits [11:2]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Valid, bits [1:0]**

This field indicates whether the ROM Table address is valid. The permitted values of this field are:

`00` ROM Table address is not valid

`11` ROM Table address is valid.

Other values are reserved.

If no memory-mapped debug components are implemented, this field is RES0.

### Accessing the MDRAR_EL1:

To access the MDRAR_EL1:

```
MRS <Xt>, MDRAR_EL1 ; Read MDRAR_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0001 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.20 MDSCR_EL1, Monitor Debug System Control Register

The MDSCR_EL1 characteristics are:

**Purpose**

Main control register for the debug implementation.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

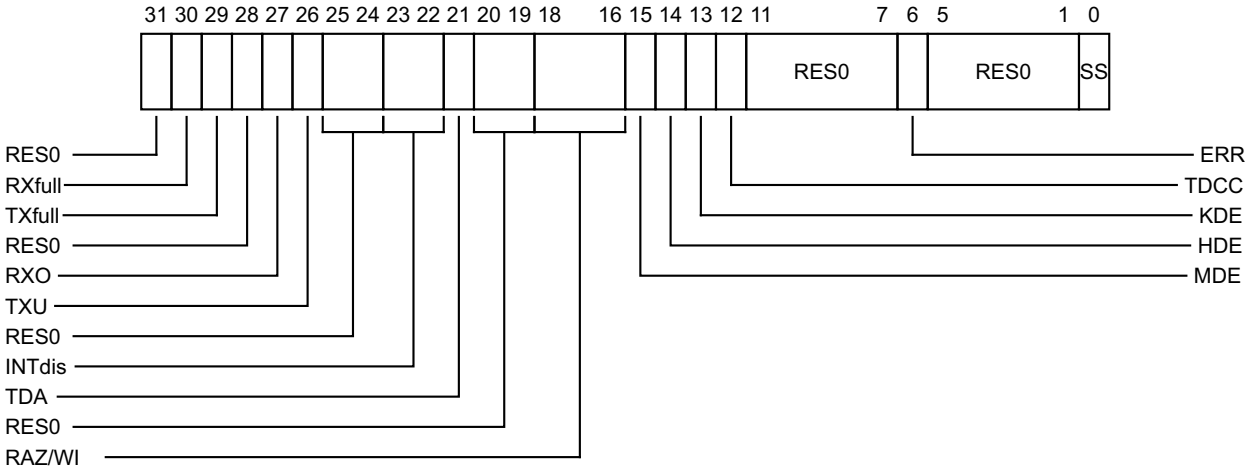
AArch64 System register MDSCR_EL1 is architecturally mapped to AArch32 System register DBGDSCRext.

**Attributes**

MDSCR_EL1 is a 32-bit register.

#### Field descriptions

The MDSCR_EL1 bit assignments are:




**Bit [31]**

Reserved, RES0.

**RXfull, bit [30]**

Used for save/restore of EDSCR.RXfull.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this bit is RW.

**TXfull, bit [29]**

Used for save/restore of EDSCR.TXfull.

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO, and software must treat it as UNK/SBZP.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this bit is RW.

**Bit [28]**

Reserved, RES0.

**RXO, bit [27]**

Used for save/restore of EDSCR.RXO.

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this bit is RW.

**TXU, bit [26]**

Used for save/restore of EDSCR.TXU.

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this bit is RW.

**Bits [25:24]**

Reserved, RES0.

**INTdis, bits [23:22]**

Used for save/restore of EDSCR.INTdis.

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this field is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this field is RW.

**TDA, bit [21]**

Used for save/restore of EDSCR.TDA.

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this bit is RW.

**Bits [20:19]**

Reserved, RES0.

**Bits [18:16]**

Reserved, RAZ/WI. Hardware must implement this as RAZ/WI. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

**MDE, bit [15]**

Monitor debug events. Enable Breakpoint, Watchpoint, and Vector catch debug exceptions.

`0` Breakpoint, Watchpoint, and Vector catch debug exceptions disabled.

`1` Breakpoint, Watchpoint, and Vector catch debug exceptions enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to its defined reset value on Warm reset.

**HDE, bit [14]**

Used for save/restore of EDSCR.HDE.

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this bit is RW.

**KDE, bit [13]**

Local (kernel) debug enable. If $EL_D$ is using AArch64, enable Software debug events within $EL_D$. Permitted values are:

`0` Software debug events, other than Software breakpoint instructions, disabled within $EL_D$.

`1` Software debug events enabled within $EL_D$.

RES0 if $EL_D$ is using AArch32.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**TDCC, bit [12]**

Traps EL0 accesses to the DCC registers to EL1, from both Execution states:

`0` EL0 using AArch64: EL0 accesses to the MDCCSR_EL0, DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0 registers are not trapped to EL1.
EL0 using AArch32: EL0 accesses to the DBGDSCRint, DBGDTRRXint, DBGDTRTXint, DBGDIDR, DBGDSAR, and DBGDRAR registers are not trapped to EL1.

`1` EL0 using AArch64: EL0 accesses to the MDCCSR_EL0, DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0 registers are trapped to EL1.
EL0 using AArch32: EL0 accesses to the DBGDSCRint, DBGDTRRXint, DBGDTRTXint, DBGDIDR, DBGDSAR, and DBGDRAR registers are trapped to EL1.

**Note**

All accesses to these AArch32 registers are trapped, including LDC and STC accesses to DBGDTRTXint and DBGDTRRXint, and MRRC accesses to DBGDSAR and DBGDRAR.

Traps of AArch32 PL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bits [11:7]**

Reserved, RES0.

**ERR, bit [6]**

Used for save/restore of EDSCR.ERR.

When OSLSR_EL1.OSLK == 0 (the OS lock is unlocked), this bit is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When OSLSR_EL1.OSLK == 1 (the OS lock is locked), this bit is RW.

**Bits [5:1]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SS, bit [0]**

Software step control bit. If $EL_D$ is using AArch64, enable Software step. Permitted values are:

`0` Software step disabled

`1` Software step enabled.

RES0 if $EL_D$ is using AArch32.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

### Accessing the MDSCR_EL1:

To access the MDSCR_EL1:

```
MRS <Xt>, MDSCR_EL1 ; Read MDSCR_EL1 into Xt
MSR MDSCR_EL1, <Xt> ; Write Xt to MDSCR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0000 | 0010 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.21 OSDLR_EL1, OS Double Lock Register

The OSDLR_EL1 characteristics are:

**Purpose**

Used to control the OS Double Lock.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDOSA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDOSA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

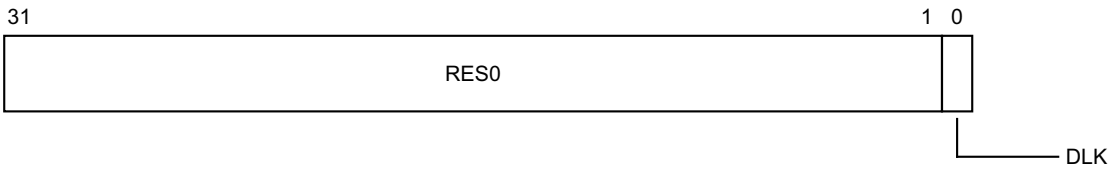
AArch64 System register OSDLR_EL1 is architecturally mapped to AArch32 System register DBGOSDLR.

**Attributes**

OSDLR_EL1 is a 32-bit register.

#### Field descriptions

The OSDLR_EL1 bit assignments are:




**Bits [31:1]**

Reserved, RES0.

**DLK, bit [0]**

OS Double Lock control bit. Possible values are:

`0` OS Double Lock unlocked.

`1` OS Double Lock locked, if DBGPRCR_EL1.CORENPDRQ (Core no powerdown request) bit is set to 0 and the PE is in Non-debug state.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

#### Accessing the OSDLR_EL1:

To access the OSDLR_EL1:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRS <Xt>, OSDLR_EL1 ; Read OSDLR_EL1 into Xt
MSR OSDLR_EL1, <Xt> ; Write Xt to OSDLR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0001 | 0011 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.22 OSDTRRX_EL1, OS Lock Data Transfer Register, Receive

The OSDTRRX_EL1 characteristics are:

**Purpose**

Used for save/restore of DBGDTRRX_EL0. It is a component of the Debug Communications Channel.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

ARM deprecates reads and writes of OSDTRRX_EL1 when the OS lock is unlocked.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

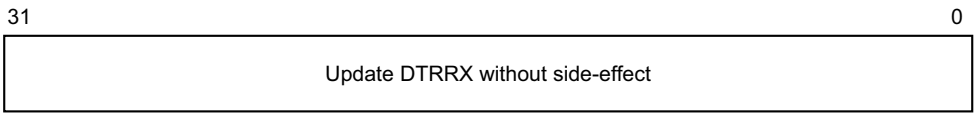
AArch64 System register OSDTRRX_EL1 is architecturally mapped to AArch32 System register DBGDTRRXext.

**Attributes**

OSDTRRX_EL1 is a 32-bit register.

#### Field descriptions

The OSDTRRX_EL1 bit assignments are:




**Bits [31:0]**

Update DTRRX without side-effect.

Writes to this register update the value in DTRRX and do not change RXfull.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

#### Accessing the OSDTRRX_EL1:

To access the OSDTRRX_EL1:

```
MRS <Xt>, OSDTRRX_EL1 ; Read OSDTRRX_EL1 into Xt
MSR OSDTRRX_EL1, <Xt> ; Write Xt to OSDTRRX_EL1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0000 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.23 OSDTRTX_EL1, OS Lock Data Transfer Register, Transmit

The OSDTRTX_EL1 characteristics are:

**Purpose**

Used for save/restore of DBGDTRTX_EL0. It is a component of the Debug Communications Channel.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

ARM deprecates reads and writes of OSDTRTX_EL1 when the OS lock is unlocked.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

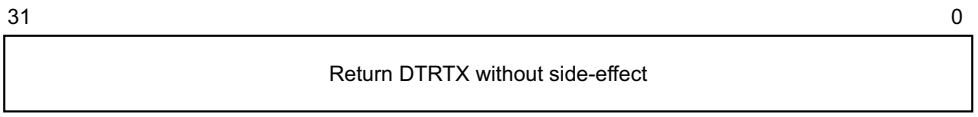
AArch64 System register OSDTRTX_EL1 is architecturally mapped to AArch32 System register DBGDTRTXext.

**Attributes**

OSDTRTX_EL1 is a 32-bit register.

#### Field descriptions

The OSDTRTX_EL1 bit assignments are:




**Bits [31:0]**

Return DTRTX without side-effect.

Reads of this register return the value in DTRTX and do not change TXfull.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

#### Accessing the OSDTRTX_EL1:

To access the OSDTRTX_EL1:

```
MRS <Xt>, OSDTRTX_EL1 ; Read OSDTRTX_EL1 into Xt
MSR OSDTRTX_EL1, <Xt> ; Write Xt to OSDTRTX_EL1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0000 | 0011 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.24 OSECCR_EL1, OS Lock Exception Catch Control Register

The OSECCR_EL1 characteristics are:

**Purpose**

Provides a mechanism for an operating system to access the contents of EDECCR that are otherwise invisible to software, so it can save/restore the contents of EDECCR over powerdown on behalf of the external debugger.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register OSECCR_EL1 is architecturally mapped to AArch32 System register DBGOSECCR.

AArch64 System register OSECCR_EL1 is architecturally mapped to External register EDECCR.

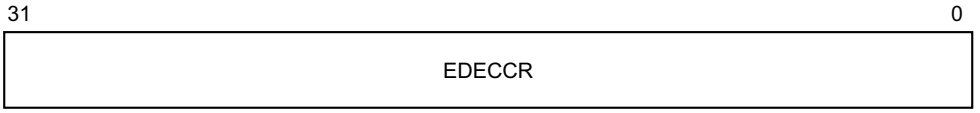
**Attributes**

OSECCR_EL1 is a 32-bit register.

#### Field descriptions

The OSECCR_EL1 bit assignments are:

***When OSLSR.OSLK==1:***




**EDECCR, bits [31:0]**

Used for save/restore to EDECCR over powerdown.

#### Accessing the OSECCR_EL1:

To access the OSECCR_EL1:

```
MRS <Xt>, OSECCR_EL1 ; Read OSECCR_EL1 into Xt
MSR OSECCR_EL1, <Xt> ; Write Xt to OSECCR_EL1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0000 | 0110 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.25 OSLAR_EL1, OS Lock Access Register

The OSLAR_EL1 characteristics are:

**Purpose**

Used to lock or unlock the OS lock.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | WO | WO | WO | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDOSA==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If MDCR_EL3.TDOSA==1, EL1 and EL2 write accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register OSLAR_EL1 is architecturally mapped to AArch32 System register DBGOSLAR.

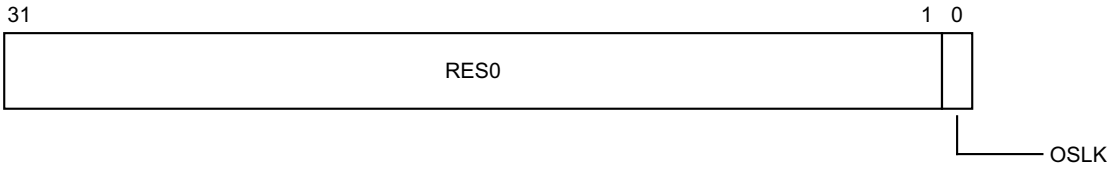
AArch64 System register OSLAR_EL1 is architecturally mapped to External register OSLAR_EL1.

**Attributes**

OSLAR_EL1 is a 32-bit register.

#### Field descriptions

The OSLAR_EL1 bit assignments are:




**Bits [31:1]**

Reserved, RES0.

**OSLK, bit [0]**

On writes to OSLAR_EL1, bit[0] is copied to the OS lock.

Use OSLSR_EL1.OSLK to check the current status of the lock.

#### Accessing the OSLAR_EL1:

To access the OSLAR_EL1:

```
MSR OSLAR_EL1, <Xt> ; Write Xt to OSLAR_EL1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0001 | 0000 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.26 OSLSR_EL1, OS Lock Status Register

The OSLSR_EL1 characteristics are:

**Purpose**

Provides the status of the OS lock.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TDOSA==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDOSA==1, EL1 and EL2 read accesses to this register are trapped to EL3.

**Configurations**

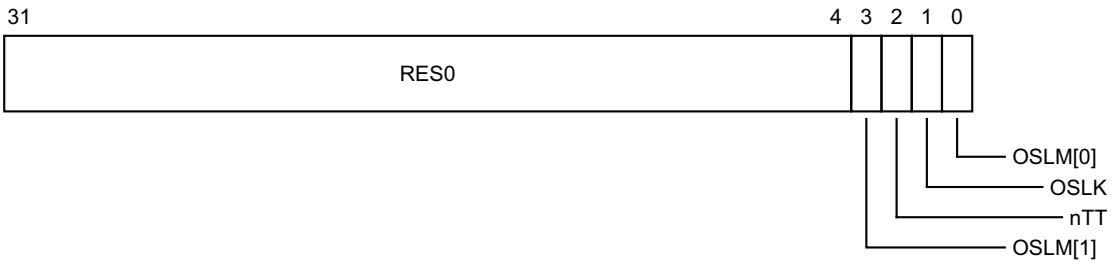
AArch64 System register OSLSR_EL1 is architecturally mapped to AArch32 System register DBGOSLSR.

**Attributes**

OSLSR_EL1 is a 32-bit register.

#### Field descriptions

The OSLSR_EL1 bit assignments are:




**Bits [31:4]**

Reserved, RES0.

**OSLM[1], bit [3]**

See below for description of the OSLM field.

**nTT, bit [2]**

Not 32-bit access. This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

**OSLK, bit [1]**

OS Lock Status. The possible values are:

0 OS lock unlocked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 OS lock locked.

The OS lock is locked and unlocked by writing to the OS Lock Access Register.

When this register has an architecturally-defined reset value, this field resets to 1.

This field resets to its defined reset value on Cold reset.

**OSLM[0], bit [0]**

OS lock model implemented. Identifies the form of OS save and restore mechanism implemented. In ARMv8 these bits are as follows:

10 OS lock implemented. DBGOSSRR not implemented.

All other values are reserved.

### Accessing the OSLSR_EL1:

To access the OSLSR_EL1:

```
MRS <Xt>, OSLSR_EL1 ; Read OSLSR_EL1 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 10 | 000 | 0001 | 0001 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.3.27 SDER32_EL3, AArch32 Secure Debug Enable Register

The SDER32_EL3 characteristics are:

**Purpose**

Allows access to the AArch32 register SDER from AArch64 state only. Its value has no effect on execution in AArch64 state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register SDER32_EL3 is architecturally mapped to AArch32 System register SDER.

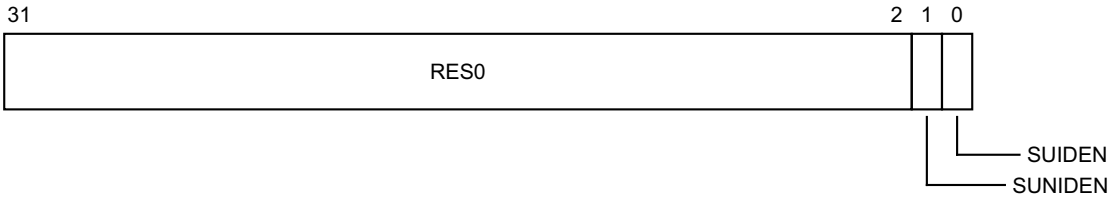
If EL1 is AArch64 only, this register is UNDEFINED.

**Attributes**

SDER32_EL3 is a 32-bit register.

#### Field descriptions

The SDER32_EL3 bit assignments are:




**Bits [31:2]**

Reserved, RES0.

**SUNIDEN, bit [1]**

Secure User Non-Invasive Debug Enable:

0 Performance Monitors event counting prohibited in Secure EL0 unless allowed by MDCR_EL3.SPME or the IMPLEMENTATION DEFINED authentication interface ExternalSecureNoninvasiveDebugEnabled().

1 Performance Monitors event counting allowed in Secure EL0.

**SUIDEN, bit [0]**

Secure User Invasive Debug Enable:

0 Debug exceptions other than Software Breakpoint Instruction exceptions from Secure EL0 are disabled, unless enabled by MDCR_EL3.SPD32.

1 Debug exceptions from Secure EL0 are enabled.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the SDER32_EL3:

To access the SDER32_EL3:

```
MRS <Xt>, SDER32_EL3 ; Read SDER32_EL3 into Xt
MSR SDER32_EL3, <Xt> ; Write Xt to SDER32_EL3
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 110 | 0001 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.4 Performance Monitors registers

This section lists the Performance Monitoring registers in AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.1 PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register

The PMCCFILTR_EL0 characteristics are:

**Purpose**

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

PMCCFILTR_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to `0b11111`.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMCCFILTR_EL0 is architecturally mapped to AArch32 System register PMCCFILTR.

AArch64 System register PMCCFILTR_EL0 is architecturally mapped to External register PMCCFILTR_EL0.

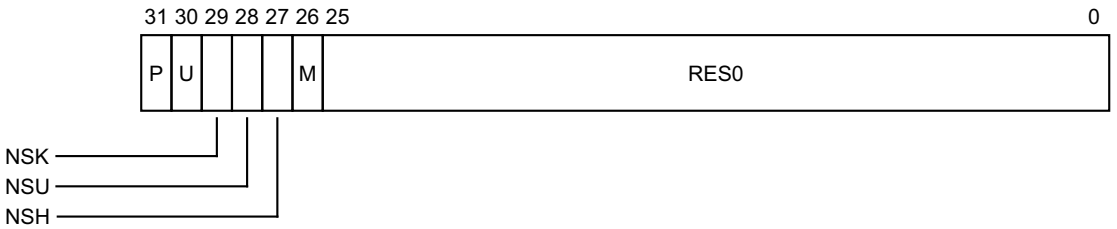
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMCCFILTR_EL0 is a 32-bit register.

#### Field descriptions

The PMCCFILTR_EL0 bit assignments are:




**P, bit [31]**

EL1 modes filtering bit. Controls counting in EL1. If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the NSK bit. The possible values of this bit are:

`0` Count cycles in EL1.

`1` Do not count cycles in EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**U, bit [30]**

EL0 filtering bit. Controls counting in EL0. If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the NSU bit. The possible values of this bit are:

0 Count cycles in EL0.

1 Do not count cycles in EL0.

**NSK, bit [29]**

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, cycles in Non-secure EL1 are counted.

Otherwise, cycles in Non-secure EL1 are not counted.

**NSU, bit [28]**

Non-secure User mode filtering bit. Controls counting in Non-secure EL0. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of U, cycles in Non-secure EL0 are counted.

Otherwise, cycles in Non-secure EL0 are not counted.

**NSH, bit [27]**

Non-secure Hyp modes filtering bit. Controls counting in Non-secure EL2. If EL2 is not implemented, this bit is RES0.

0 Do not count cycles in EL2.

1 Count cycles in EL2.

**M, bit [26]**

Secure EL3 filtering bit. Most applications can ignore this bit and set the value to zero. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, cycles in Secure EL3 are counted.

Otherwise, cycles in Secure EL3 are not counted.

**Bits [25:0]**

Reserved, RES0.

### Accessing the PMCCFILTR_EL0:

To access the PMCCFILTR_EL0:

```
MRS <Xt>, PMCCFILTR_EL0 ; Read PMCCFILTR_EL0 into Xt
MSR PMCCFILTR_EL0, <Xt> ; Write Xt to PMCCFILTR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 1111 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.2 PMCCNTR_EL0, Performance Monitors Cycle Count Register

The PMCCNTR_EL0 characteristics are:

**Purpose**

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See *Time as measured by the Performance Monitors cycle counter* on page D5-1858 for more information.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.CR==0, and PMUSERENR_EL0.EN==0, EL0 read accesses to this register are trapped to EL1.

If PMUSERENR_EL0.EN==0, EL0 write accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMCCNTR_EL0 is architecturally mapped to AArch32 System register PMCCNTR.

AArch64 System register PMCCNTR_EL0 is architecturally mapped to External register PMCCNTR_EL0.

All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions. This means that it is CONSTRAINED UNPREDICTABLE whether or not PMCCNTR_EL0 continues to increment when clocks are stopped by WFI and WFE instructions.

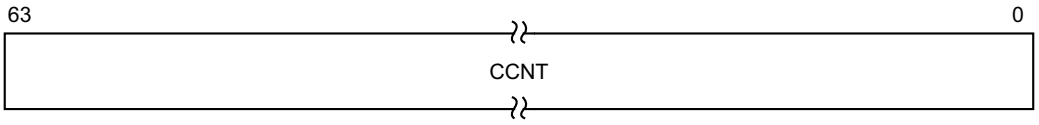
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMCCNTR_EL0 is a 64-bit register.

#### Field descriptions

The PMCCNTR_EL0 bit assignments are:




**CCNT, bits [63:0]**

Cycle count. Depending on the values of PMCR_EL0.{LC,D}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Writing 1 to PMCR_EL0.C sets this field to 0.

### Accessing the PMCCNTR_EL0:

To access the PMCCNTR_EL0:

```
MRS <Xt>, PMCCNTR_EL0 ; Read PMCCNTR_EL0 into Xt
MSR PMCCNTR_EL0, <Xt> ; Write Xt to PMCCNTR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.3 PMCEID0_EL0, Performance Monitors Common Event Identification register 0

The PMCEID0_EL0 characteristics are:

**Purpose**

Defines which common architectural and common microarchitectural feature events are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 read accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, EL0 read accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMCEID0_EL0 is architecturally mapped to AArch32 System register PMCEID0.

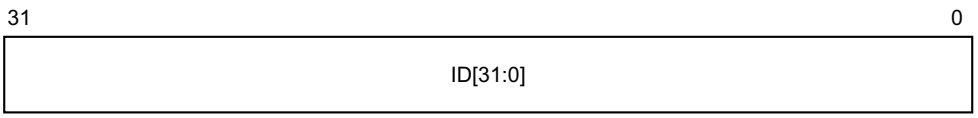
AArch64 System register PMCEID0_EL0 is architecturally mapped to External register PMCEID0_EL0.

**Attributes**

PMCEID0_EL0 is a 32-bit register.

#### Field descriptions

The PMCEID0_EL0 bit assignments are:




**ID[31:0], bits [31:0]**

PMCEID0_EL0[n] maps to event n. For a list of event numbers and descriptions, see *Events, Event numbers and mnemonics* on page D5-1872.

For each bit:

`0` The common event is not implemented.

`1` The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMCEID0_EL0:

To access the PMCEID0_EL0:

```
MRS <Xt>, PMCEID0_EL0 ; Read PMCEID0_EL0 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1100 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.4 PMCEID1_EL0, Performance Monitors Common Event Identification register 1

The PMCEID1_EL0 characteristics are:

**Purpose**

Defines which common architectural and common microarchitectural feature events are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 read accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, EL0 read accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMCEID1_EL0 is architecturally mapped to AArch32 System register PMCEID1.

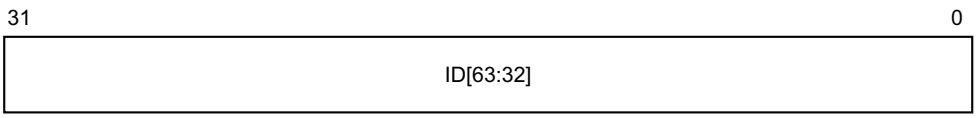
AArch64 System register PMCEID1_EL0 is architecturally mapped to External register PMCEID1_EL0.

**Attributes**

PMCEID1_EL0 is a 32-bit register.

#### Field descriptions

The PMCEID1_EL0 bit assignments are:




**ID[63:32], bits [31:0]**

PMCEID1_EL0[n] maps to event (n + 32). For a list of event numbers and descriptions, see *Events, Event numbers and mnemonics* on page D5-1872.

For each bit:

`0` The common event is not implemented.

`1` The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMCEID1_EL0:

To access the PMCEID1_EL0:

```
MRS <Xt>, PMCEID1_EL0 ; Read PMCEID1_EL0 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1100 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.5 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register

The PMCNTENCLR_EL0 characteristics are:

**Purpose**

Disables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<x>. Reading this register shows which counters are enabled.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMCNTENCLR_EL0 is architecturally mapped to AArch32 System register PMCNTENCLR.

AArch64 System register PMCNTENCLR_EL0 is architecturally mapped to External register PMCNTENCLR_EL0.

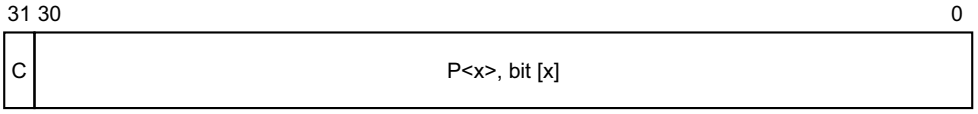
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMCNTENCLR_EL0 is a 32-bit register.

#### Field descriptions

The PMCNTENCLR_EL0 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 disable bit. Disables the cycle counter register. Possible values are:

`0` When read, means the cycle counter is disabled. When written, has no effect.

`1` When read, means the cycle counter is enabled. When written, disables the cycle counter.

**P<x>, bit [x], for x = 0 to 30**

Event counter disable bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Possible values of each bit are:

0 When read, means that PMEVCNTR<x> is disabled. When written, has no effect.

1 When read, means that PMEVCNTR<x> is enabled. When written, disables PMEVCNTR<x>.

### Accessing the PMCNTENCLR_EL0:

To access the PMCNTENCLR_EL0:

```
MRS <Xt>, PMCNTENCLR_EL0 ; Read PMCNTENCLR_EL0 into Xt
MSR PMCNTENCLR_EL0, <Xt> ; Write Xt to PMCNTENCLR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1100 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.6 PMCNTENSET_EL0, Performance Monitors Count Enable Set register

The PMCNTENSET_EL0 characteristics are:

**Purpose**

Enables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<x>. Reading this register shows which counters are enabled.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMCNTENSET_EL0 is architecturally mapped to AArch32 System register PMCNTENSET.

AArch64 System register PMCNTENSET_EL0 is architecturally mapped to External register PMCNTENSET_EL0.

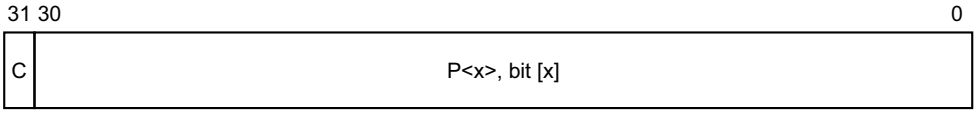
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMCNTENSET_EL0 is a 32-bit register.

#### Field descriptions

The PMCNTENSET_EL0 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 enable bit. Enables the cycle counter register. Possible values are:

`0` When read, means the cycle counter is disabled. When written, has no effect.

`1` When read, means the cycle counter is enabled. When written, enables the cycle counter.

**P<x>, bit [x], for x = 0 to 30**

Event counter enable bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Possible values of each bit are:

0 When read, means that PMEVCNTR<x> is disabled. When written, has no effect.

1 When read, means that PMEVCNTR<x> event counter is enabled. When written, enables PMEVCNTR<x>.

### Accessing the PMCNTENSET_EL0:

To access the PMCNTENSET_EL0:

```
MRS <Xt>, PMCNTENSET_EL0 ; Read PMCNTENSET_EL0 into Xt
MSR PMCNTENSET_EL0, <Xt> ; Write Xt to PMCNTENSET_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1100 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.7 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

**Purpose**

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPMCR==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMCR_EL0 is architecturally mapped to AArch32 System register PMCR.

AArch64 System register PMCR_EL0[6:0] is architecturally mapped to External register PMCR_EL0[6:0].

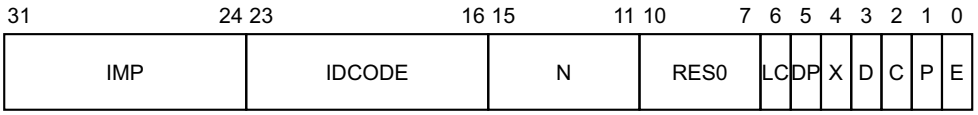
Some or all RW fields of this register have defined reset values. These apply on Warm reset, and only if the PE resets into an Exception level that is using AArch64. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMCR_EL0 is a 32-bit register.

#### Field descriptions

The PMCR_EL0 bit assignments are:




**IMP, bits [31:24]**

Implementer code. This field is RO with an IMPLEMENTATION DEFINED value.

The implementer codes are allocated by ARM. Values have the same interpretation as bits [31:24] of the MIDR.

**IDCODE, bits [23:16]**

Identification code. This field is RO with an IMPLEMENTATION DEFINED value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Each implementer must maintain a list of identification codes that is specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

**N, bits [15:11]**

Number of event counters. A RO field that indicates the number counters implemented. The value of this field is the number of counters implemented, from `0b00000` for no counters to `0b11111` for 31 counters.

In an implementation that includes EL2, reads of this field from Non-secure EL1 and Non-secure EL0 return the value of MDCR_EL2.HPMN.

An implementation can implement only the Cycle Count Register, PMCCNTR_EL0. This is indicated by a value of `0b00000` for the N field.

**Bits [10:7]**

Reserved, RES0.

**LC, bit [6]**

Long cycle counter enable. Determines which PMCCNTR_EL0 bit generates an overflow recorded by PMOVSR[31].

`0` Cycle counter overflow on increment that changes PMCCNTR_EL0[31] from 1 to 0.

`1` Cycle counter overflow on increment that changes PMCCNTR_EL0[63] from 1 to 0.

ARM deprecates use of PMCR_EL0.LC = 0.

In an AArch64-only implementation, this field is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**DP, bit [5]**

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

`0` PMCCNTR_EL0, if enabled, counts when event counting is prohibited.

`1` PMCCNTR_EL0 does not count when event counting is prohibited.

Event counting is prohibited when ProfilingProhibited(IsSecure(),PSTATE.EL) == TRUE.

This bit is RW.

If EL3 is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**X, bit [4]**

Enable export of events in an IMPLEMENTATION DEFINED event stream. The possible values of this bit are:

`0` Do not export events.

`1` Export events where not prohibited.

This bit is used to permit events to be exported to another debug device, such as an OPTIONAL trace macrocell, over an event bus. If the implementation does not include such an event bus, this bit is RAZ/WI.

This bit does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If the implementation does not include an exported event stream, this bit is RAZ/WI. Otherwise this bit is RW.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**D, bit [3]**

Clock divider. The possible values of this bit are:

`0` When enabled, PMCCNTR_EL0 counts every clock cycle.

`1` When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

This bit is RW.

If PMCR_EL0.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

ARM deprecates use of PMCR.D = 1.

In an AArch64-only implementation, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**C, bit [2]**

Cycle counter reset. This bit is WO. The effects of writing to this bit are:

`0` No action.

`1` Reset PMCCNTR_EL0 to zero.

This bit is always RAZ.

Resetting PMCCNTR_EL0 does not clear the PMCCNTR_EL0 overflow bit to 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**P, bit [1]**

Event counter reset. This bit is WO. The effects of writing to this bit are:

`0` No action.

`1` Reset all event counters accessible in the current EL, not including PMCCNTR_EL0, to zero.

This bit is always RAZ.

In Non-secure EL0 and EL1, if EL2 is implemented, a write of 1 to this bit does not reset event counters that MDCR_EL2.HPMN reserves for EL2 use.

In EL2 and EL3, a write of 1 to this bit resets all the event counters.

Resetting the event counters does not clear any overflow bits to 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**E, bit [0]**

Enable. The possible values of this bit are:

`0` All counters, including PMCCNTR_EL0, are disabled.

`1` All counters are enabled by PMCNTENSET_EL0.

This bit is RW.

If EL2 is implemented, this bit does not affect the operation of event counters that MDCR_EL2.HPMN reserves for EL2 use.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMCR_EL0:

To access the PMCR_EL0:

```
MRS <Xt>, PMCR_EL0 ; Read PMCR_EL0 into Xt
MSR PMCR_EL0, <Xt> ; Write Xt to PMCR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1100 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.8 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

**Purpose**

Holds event counter n, which counts events, where n is 0 to 30.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

PMEVCNTR<n>_EL0 can also be accessed by using PMXEVCNTR_EL0 with PMSELR_EL0.SEL set to n.

If <n> is greater than or equal to the number of accessible counters, reads and writes of PMEVCNTR<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- For an access from Non-secure EL1, or an access from Non-secure EL0 when the value of PMUSERENR_EL0.EN is 1, if PMSELR_EL0.SEL is greater than or equal to the number of accessible counters but is less than the number of implemented counters, the register access is trapped to EL2.

**Note**

In an implementation that includes EL2, in Non-secure state at EL0 and EL1, MDCR_EL2.HPMN identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, and PMUSERENR_EL0.ER==0, EL0 read accesses to this register are trapped to EL1.

If PMUSERENR_EL0.EN==0, EL0 write accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMEVCNTR<n>_EL0 is architecturally mapped to AArch32 System register PMEVCNTR<n>.

AArch64 System register PMEVCNTR<n>_EL0 is architecturally mapped to External register PMEVCNTR<n>_EL0.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

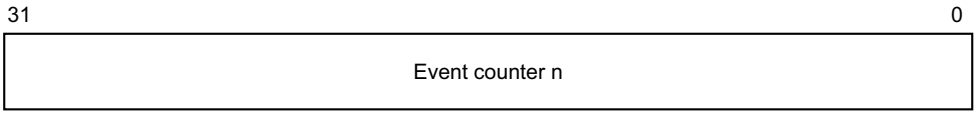
**Attributes**

PMEVCNTR<n>_EL0 is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The PMEVCNTR<n>_EL0 bit assignments are:




**Bits [31:0]**

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

### Accessing the PMEVCNTR<n>_EL0:

To access the PMEVCNTR<n>_EL0:

```
MRS <Xt>, PMEVCNTR<n>_EL0 ; Read PMEVCNTR<n>_EL0 into Xt, where n is in the range 0 to 30
MSR PMEVCNTR<n>_EL0, <Xt> ; Write Xt to PMEVCNTR<n>_EL0, where n is in the range 0 to 30
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 10:n<4:3> | n<2:0> |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.9 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

**Purpose**

Configures event counter n, where n is 0 to 30.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

PMEVTYPER<n>_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to n.

If <n> is greater than or equal to the number of accessible counters, reads and writes of PMEVTYPER<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- For an access from Non-secure EL1, or an access from Non-secure EL0 when the value of PMUSERENR_EL0.EN is 1, if PMSELR_EL0.SEL is greater than or equal to the number of accessible counters but is less than the number of implemented counters, the register access is trapped to EL2.

**Note**

In an implementation that includes EL2, in Non-secure state at EL0 and EL1, MDCR_EL2.HPMN identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMEVTYPER<n>_EL0 is architecturally mapped to AArch32 System register PMEVTYPER<n>.

AArch64 System register PMEVTYPER<n>_EL0 is architecturally mapped to External register PMEVTYPER<n>_EL0.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

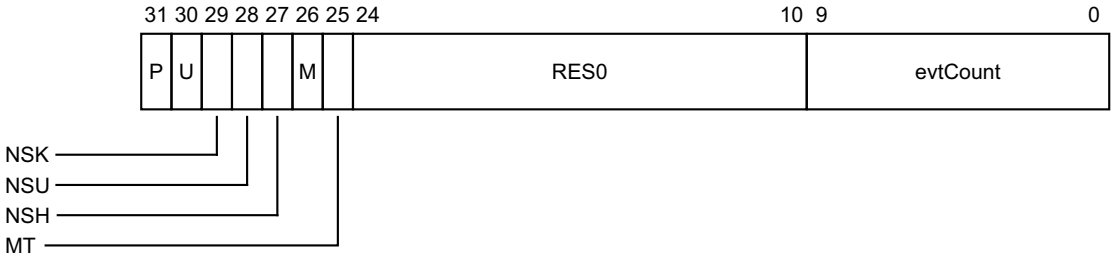
**Attributes**

PMEVTYPER<n>_EL0 is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The PMEVTYPER<n>_EL0 bit assignments are:




**P, bit [31]**

EL1 modes filtering bit. Controls counting in EL1. If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the NSK bit. The possible values of this bit are:

`0` Count events in EL1.

`1` Do not count events in EL1.

**U, bit [30]**

EL0 filtering bit. Controls counting in EL0. If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the NSU bit. The possible values of this bit are:

`0` Count events in EL0.

`1` Do not count events in EL0.

**NSK, bit [29]**

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.

**NSU, bit [28]**

Non-secure User mode filtering bit. Controls counting in Non-secure EL0. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of U, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.

**NSH, bit [27]**

Non-secure Hyp modes filtering bit. Controls counting in Non-secure EL2. If EL2 is not implemented, this bit is RES0.

`0` Do not count events in EL2.

`1` Count events in EL2.

**M, bit [26]**

Secure EL3 filtering bit. Most applications can ignore this bit and set the value to zero. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, events in Secure EL3 are counted.

Otherwise, events in Secure EL3 are not counted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**MT, bit [25]**

Multi-threading. If MPIDR_EL1.MT is set to 0, this bit is RES0. Otherwise valid values for this bit are:

0 Count events only on controlling PE.

1 Count events from any PE with the same affinity at level 1 and above as this PE.

**Note**

Events from a different thread of a multi-threaded implementation are not Attributable to the thread counting the event.

**Bits [24:10]**

Reserved, RES0.

**evtCount, bits [9:0]**

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>_EL0.

Software must program this field with an event defined by the processor or a common event defined by the architecture.

If evtCount is programmed to an event that is reserved or not implemented, the behavior depends on the event type.

For common architectural and microarchitectural events:

- No events are counted.
- The value read back on evtCount is the value written.

For IMPLEMENTATION DEFINED events:

- It is UNPREDICTABLE what event, if any, is counted. UNPREDICTABLE in this case means the event must not expose privileged information.
- The value read back on evtCount is an UNKNOWN value with the same effect.

ARM recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

## Accessing the PMEVTYPER<n>_EL0:

To access the PMEVTYPER<n>_EL0:

```
MRS <Xt>, PMEVTYPER<n>_EL0 ; Read PMEVTYPER<n>_EL0 into Xt, where n is in the range 0 to 30
MSR PMEVTYPER<n>_EL0, <Xt> ; Write Xt to PMEVTYPER<n>_EL0, where n is in the range 0 to 30
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 11:n<4:3> | n<2:0> |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.10 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR_EL1 characteristics are:

**Purpose**

Disables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<n>_EL0. Reading the register shows which overflow interrupt requests are enabled.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register PMINTENCLR_EL1 is architecturally mapped to AArch32 System register PMINTENCLR.

AArch64 System register PMINTENCLR_EL1 is architecturally mapped to External register PMINTENCLR_EL1.

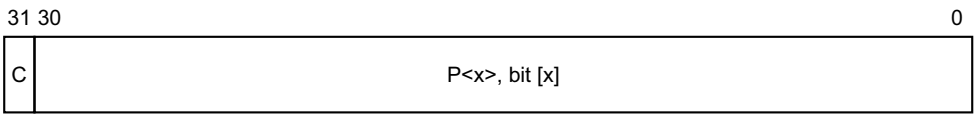
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMINTENCLR_EL1 is a 32-bit register.

#### Field descriptions

The PMINTENCLR_EL1 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 overflow interrupt request disable bit. Possible values are:

`0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`1` When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow interrupt request disable bit for PMEVCNTR<x>_EL0.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [30:N] are RAZ/WI.

Possible values are:

`0` When read, means that the PMEVCNTR<x>_EL0 event counter interrupt request is disabled. When written, has no effect.

`1` When read, means that the PMEVCNTR<x>_EL0 event counter interrupt request is enabled. When written, disables the PMEVCNTR<x>_EL0 interrupt request.

### Accessing the PMINTENCLR_EL1:

To access the PMINTENCLR_EL1:

```
MRS <Xt>, PMINTENCLR_EL1 ; Read PMINTENCLR_EL1 into Xt
MSR PMINTENCLR_EL1, <Xt> ; Write Xt to PMINTENCLR_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1001 | 1110 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.11 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register

The PMINTENSET_EL1 characteristics are:

**Purpose**

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<n>_EL0. Reading the register shows which overflow interrupt requests are enabled.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL1 and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register PMINTENSET_EL1 is architecturally mapped to AArch32 System register PMINTENSET.

AArch64 System register PMINTENSET_EL1 is architecturally mapped to External register PMINTENSET_EL1.

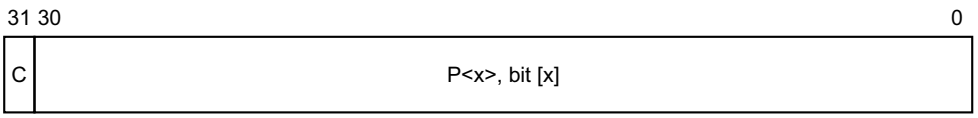
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMINTENSET_EL1 is a 32-bit register.

#### Field descriptions

The PMINTENSET_EL1 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 overflow interrupt request enable bit. Possible values are:

`0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`1` When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow interrupt request enable bit for PMEVCNTR<x>_EL0.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bits [30:N] are RAZ/WI.

Possible values are:

`0` When read, means that the PMEVCNTR<x>_EL0 event counter interrupt request is disabled. When written, has no effect.

`1` When read, means that the PMEVCNTR<x>_EL0 event counter interrupt request is enabled. When written, enables the PMEVCNTR<x>_EL0 interrupt request.

### Accessing the PMINTENSET_EL1:

To access the PMINTENSET_EL1:

```
MRS <Xt>, PMINTENSET_EL1 ; Read PMINTENSET_EL1 into Xt
MSR PMINTENSET_EL1, <Xt> ; Write Xt to PMINTENSET_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1001 | 1110 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.12 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register

The PMOVSCLR_EL0 characteristics are:

**Purpose**

Contains the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<x>. Writing to this register clears these bits.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

This register is accessible at EL0 when PMUSERENR_EL0.EN is set to 1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMOVSCLR_EL0 is architecturally mapped to AArch32 System register PMOVSR.

AArch64 System register PMOVSCLR_EL0 is architecturally mapped to External register PMOVSCLR_EL0.

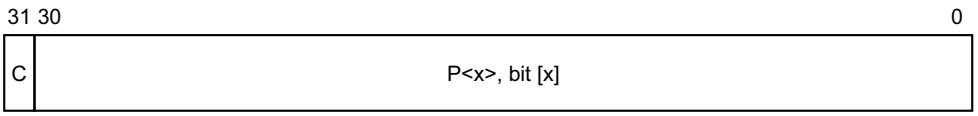
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMOVSCLR_EL0 is a 32-bit register.

## Field descriptions

The PMOVSCLR_EL0 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 overflow bit. Possible values are:

`0` When read, means the cycle counter has not overflowed. When written, has no effect.

`1` When read, means the cycle counter has overflowed. When written, clears the overflow bit to 0.

PMCR_EL0.LC is used to control from which bit of PMCCNTR_EL0 (bit 31 or bit 63) an overflow is detected.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow clear bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

Possible values of each bit are:

`0` When read, means that PMEVCNTR<x> has not overflowed. When written, has no effect.

`1` When read, means that PMEVCNTR<x> has overflowed. When written, clears the PMEVCNTR<x> overflow bit to 0.

## Accessing the PMOVSCLR_EL0:

To access the PMOVSCLR_EL0:

```
MRS <Xt>, PMOVSCLR_EL0 ; Read PMOVSCLR_EL0 into Xt
MSR PMOVSCLR_EL0, <Xt> ; Write Xt to PMOVSCLR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1100 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.13 PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register

The PMOVSSET_EL0 characteristics are:

**Purpose**

Sets the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<x>.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMOVSSET_EL0 is architecturally mapped to AArch32 System register PMOVSSET.

AArch64 System register PMOVSSET_EL0 is architecturally mapped to External register PMOVSSET_EL0.

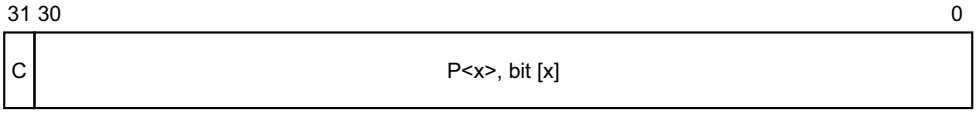
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMOVSSET_EL0 is a 32-bit register.

#### Field descriptions

The PMOVSSET_EL0 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 overflow bit. Possible values are:

`0` When read, means the cycle counter has not overflowed. When written, has no effect.

`1` When read, means the cycle counter has overflowed. When written, sets the overflow bit to 1.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow set bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Possible values are:

0 When read, means that PMEVCNTR<x> has not overflowed. When written, has no effect.

1 When read, means that PMEVCNTR<x> has overflowed. When written, sets the PMEVCNTR<x> overflow bit to 1.

### Accessing the PMOVSSET_EL0:

To access the PMOVSSET_EL0:

```
MRS <Xt>, PMOVSSET_EL0 ; Read PMOVSSET_EL0 into Xt
MSR PMOVSSET_EL0, <Xt> ; Write Xt to PMOVSSET_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1110 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.14 PMSELR_EL0, Performance Monitors Event Counter Selection Register

The PMSELR_EL0 characteristics are:

**Purpose**

Selects the current event counter PMEVCNTR<x> or the cycle counter, CCNT.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, and PMUSERENR_EL0.ER==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMSELR_EL0 is architecturally mapped to AArch32 System register PMSELR.

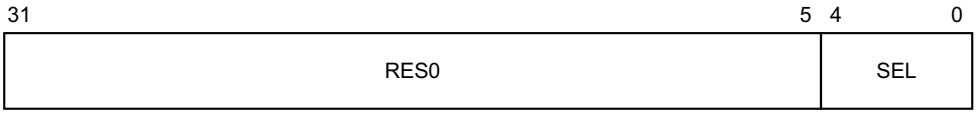
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMSELR_EL0 is a 32-bit register.

#### Field descriptions

The PMSELR_EL0 bit assignments are:




**Bits [31:5]**

Reserved, RES0.

**SEL, bits [4:0]**

Selects event counter, PMEVCNTR<x>, where x is the value held in this field. This value identifies which event counter is accessed when a subsequent access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 occurs.

This field can take any value from 0 (0b00000) to (PMCR.N)-1, or 31 (0b11111).

When PMSELR_EL0.SEL is 0b11111 it selects the cycle counter and:

- A read of the PMXEVTYPER_EL0 returns the value of PMCCFILTR_EL0.
- A write of the PMXEVTYPER_EL0 writes to PMCCFILTR_EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A read or write of PMXEVCNTR_EL0 has CONSTRAINED UNPREDICTABLE effects, that can be one of the following:
  - Access to PMXEVCNTR_EL0 is UNDEFINED.
  - Access to PMXEVCNTR_EL0 behaves as a NOP.
  - Access to PMXEVCNTR_EL0 behaves as if the register is RAZ/WI.
  - Access to PMXEVCNTR_EL0 behaves as if the PMSELR_EL0.SEL field contains an UNKNOWN value.

If this field is set to a value greater than or equal to the number of implemented counters, but not equal to 31, the results of access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 are CONSTRAINED UNPREDICTABLE, and can be one of the following:

- Access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 is UNDEFINED.
- Access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 behaves as a NOP.
- Access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 behaves as if the register is RAZ/WI.
- Access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 behaves as if the PMSELR_EL0.SEL field contains an UNKNOWN value.
- Access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 behaves as if the PMSELR_EL0.SEL field contains 0b11111.

### Accessing the PMSELR_EL0:

To access the PMSELR_EL0:

```
MRS <Xt>, PMSELR_EL0 ; Read PMSELR_EL0 into Xt
MSR PMSELR_EL0, <Xt> ; Write Xt to PMSELR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1100 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.15 PMSWINC_EL0, Performance Monitors Software Increment register

The PMSWINC_EL0 characteristics are:

**Purpose**

Increments a counter that is configured to count the Software increment event, event `0x00`.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-WO | WO | WO | WO | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 write accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 write accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, and PMUSERENR_EL0.SW==0, EL0 write accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMSWINC_EL0 is architecturally mapped to AArch32 System register PMSWINC.

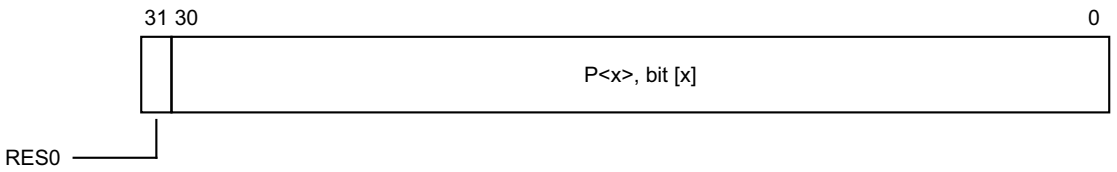
AArch64 System register PMSWINC_EL0 is architecturally mapped to External register PMSWINC_EL0.

**Attributes**

PMSWINC_EL0 is a 32-bit register.

#### Field descriptions

The PMSWINC_EL0 bit assignments are:




**Bit [31]**

Reserved, RES0.

**P<x>, bit [x], for x = 0 to 30**

Event counter software increment bit for PMEVCNTR<x>.

Bits [30:N] are WI.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR.N.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The effects of writing to this bit are:

0 No action. The write to this bit is ignored.

1 If PMEVCNTR<x> is enabled and configured to count the software increment event, increments PMEVCNTR<x> by 1. If PMEVCNTR<x> is disabled, or not configured to count the software increment event, the write to this bit is ignored.

### Accessing the PMSWINC_EL0:

To access the PMSWINC_EL0:

```
MSR PMSWINC_EL0, <Xt> ; Write Xt to PMSWINC_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1100 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.16 PMUSERENR_EL0, Performance Monitors User Enable Register

The PMUSERENR_EL0 characteristics are:

**Purpose**

Enables or disables EL0 access to the Performance Monitors.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RO | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

**Configurations**

AArch64 System register PMUSERENR_EL0 is architecturally mapped to AArch32 System register PMUSERENR.

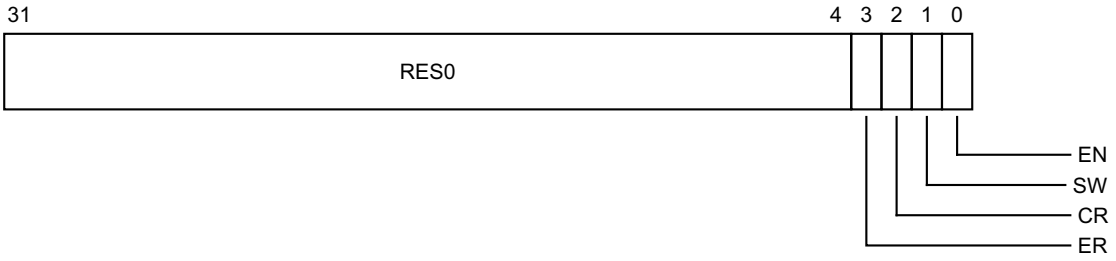
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMUSERENR_EL0 is a 32-bit register.

#### Field descriptions

The PMUSERENR_EL0 bit assignments are:




**Bits [31:4]**

Reserved, RES0.

**ER, bit [3]**

Event counter read trap control:

0 EL0 using AArch64: EL0 reads of the PMXEVCNTR_EL0 and PMEVCNTR<n>_EL0, and EL0 read/write accesses to the PMSELR_EL0, are trapped to EL1 if PMUSERENR_EL0.EN is also 0.

EL0 using AArch32: EL0 reads of the PMXEVCNTR and PMEVCNTR<n>, and EL0 read/write accesses to the PMSELR, are trapped to EL1 if PMUSERENR_EL0.EN is also 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 EL0 using AArch64: EL0 reads of the PMXEVCNTR_EL0 and PMEVCNTR<n>_EL0, and EL0 read/write accesses to the PMSELR_EL0, are not trapped to EL1.

EL0 using AArch32: EL0 reads of the PMXEVCNTR and PMEVCNTR<n>, and EL0 read/write accesses to the PMSELR, are not trapped to EL1.

**CR, bit [2]**

Cycle counter read trap control:

0 EL0 using AArch64: EL0 read accesses to the PMCCNTR_EL0 are trapped to EL1 if PMUSERENR_EL0.EN is also 0.

EL0 using AArch32: EL0 read accesses to the PMCCNTR are trapped to EL1 if PMUSERENR_EL0.EN is also 0.

1 EL0 using AArch64: EL0 read accesses to the PMCCNTR_EL0 are not trapped to EL1.

EL0 using AArch32: EL0 read accesses to the PMCCNTR are not trapped to EL1.

**SW, bit [1]**

Software Increment write trap control:

0 EL0 using AArch64: EL0 writes to the PMSWINC_EL0 are trapped to EL1 if PMUSERENR_EL0.EN is also 0.

EL0 using AArch32: EL0 writes to the PMSWINC are trapped to EL1 if PMUSERENR_EL0.EN is also 0.

1 EL0 using AArch64: EL0 writes to the PMSWINC_EL0 are not trapped to EL1.

EL0 using AArch32: EL0 writes to the PMSWINC are not trapped to EL1.

**EN, bit [0]**

Traps EL0 accesses to the Performance Monitors registers to EL1, from both Execution states:

0 EL0 accesses to the Performance Monitors registers are trapped to EL1.

1 EL0 accesses to the Performance Monitors registers are not trapped to EL1. Software can access all PMU registers at EL0.

**Note**

- The PMUSERENR_EL0 and PMUSERENR registers are always RO at EL0 and not trapped by this bit.
- EL0 cannot read or write PMINTENSET_EL1 and PMINTENCLR_EL1, or PMINTENSET and PMINTENCLR.

### Accessing the PMUSERENR_EL0:

To access the PMUSERENR_EL0:

```
MRS <Xt>, PMUSERENR_EL0 ; Read PMUSERENR_EL0 into Xt
MSR PMUSERENR_EL0, <Xt> ; Write Xt to PMUSERENR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1110 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.17 PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register

The PMXEVCNTR_EL0 characteristics are:

**Purpose**

Reads or writes the value of the selected event counter, PMEVCNTR<x>_EL0. PMSELR_EL0.SEL determines which event counter is selected.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

If PMSELR_EL0.SEL is greater than or equal to the number of accessible counters then reads and writes of PMXEVCNTR_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.
- Accesses to the register behave as if PMSELR_EL0.SEL has an UNKNOWN value less than the number of counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR_EL0.SEL is 31.
- For an access from Non-secure EL1, or an access from Non-secure EL0 when the value of PMUSERENR_EL0.EN is 1, if PMSELR_EL0.SEL is greater than or equal to the number of accessible counters but is less than the number of implemented counters, the register access is trapped to EL2.

**Note**

In an implementation that includes EL2, in Non-secure state at EL0 and EL1, MDCR_EL2.HPMN identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, and PMUSERENR_EL0.ER==0, EL0 read accesses to this register are trapped to EL1.

If PMUSERENR_EL0.EN==0, EL0 write accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMXEVCNTR_EL0 is architecturally mapped to AArch32 System register PMXEVCNTR.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

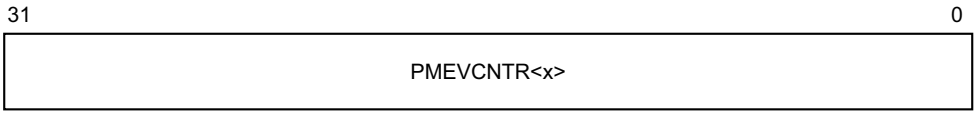
**Attributes**

PMXEVCNTR_EL0 is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The PMXEVCNTR_EL0 bit assignments are:




**PMEVCNTR<x>, bits [31:0]**

Value of the selected event counter, PMEVCNTR<x>_EL0, where x is the value stored in PMSELR_EL0.SEL.

### Accessing the PMXEVCNTR_EL0:

To access the PMXEVCNTR_EL0:

```
MRS <Xt>, PMXEVCNTR_EL0 ; Read PMXEVCNTR_EL0 into Xt
MSR PMXEVCNTR_EL0, <Xt> ; Write Xt to PMXEVCNTR_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1101 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.4.18 PMXEVTYPER_EL0, Performance Monitors Selected Event Type Register

The PMXEVTYPER_EL0 characteristics are:

**Purpose**

When PMSELR_EL0.SEL selects an event counter, this accesses a PMEVTYPER<n>_EL0 register. When PMSELR_EL0.SEL selects the cycle counter, this accesses PMCCFILTR_EL0.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

If PMSELR_EL0.SEL is greater than or equal to the number of accessible counters then reads and writes of PMXEVTYPER_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if PMSELR_EL0.SEL has an UNKNOWN value less than the number of counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR_EL0.SEL is 31.
- For an access from Non-secure EL1, or an access from Non-secure EL0 when the value of PMUSERENR_EL0.EN is 1, if PMSELR_EL0.SEL is greater than or equal to the number of accessible counters but is less than the number of implemented counters, the register access is trapped to EL2.

**Note**

In an implementation that includes EL2, in Non-secure state at EL0 and EL1, MDCR_EL2.HPMN identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If PMUSERENR_EL0.EN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register PMXEVTYPER_EL0 is architecturally mapped to AArch32 System register PMXEVTYPER.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

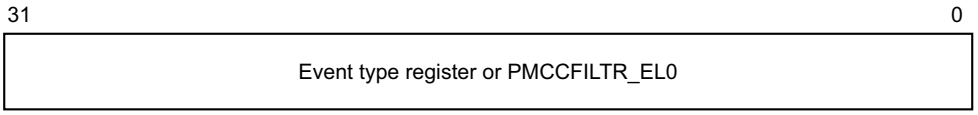
**Attributes**

PMXEVTYPER_EL0 is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The PMXEVTYPER_EL0 bit assignments are:




**Bits [31:0]**

Event type register or PMCCFILTR_EL0.

When PMSELR_EL0.SEL == 31, this register accesses PMCCFILTR_EL0.

Otherwise, this register accesses PMEVTYPER<n>_EL0 where n is the value in PMSELR_EL0.SEL.

### Accessing the PMXEVTYPER_EL0:

To access the PMXEVTYPER_EL0:

```
MRS <Xt>, PMXEVTYPER_EL0 ; Read PMXEVTYPER_EL0 into Xt
MSR PMXEVTYPER_EL0, <Xt> ; Write Xt to PMXEVTYPER_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1001 | 1101 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## D7.5 Generic Timer registers

This section lists the Generic Timer registers in AArch64.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### D7.5.1 CNTFRQ_EL0, Counter-timer Frequency register

The CNTFRQ_EL0 characteristics are:

**Purpose**

Holds the clock frequency of the system counter.

**Usage constraints**

If EL1 is the highest exception level implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 |
|---|---|
| Config-RO | RW |

If EL2 is the highest exception level implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RW |

If EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | RO | RO | RO | RW | RW |

Can only be written at the highest Exception level implemented. For example, if EL3 is the highest implemented Exception level, CNTFRQ_EL0 can only be written at EL3.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CNTKCTL_EL1.EL0PCTEN==0, and CNTKCTL_EL1.EL0VCTEN==0, EL0 read accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register CNTFRQ_EL0 is architecturally mapped to AArch32 System register CNTFRQ.

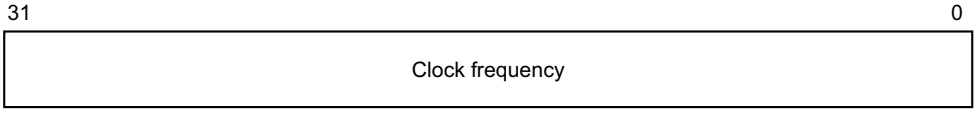
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTFRQ_EL0 is a 32-bit register.

#### Field descriptions

The CNTFRQ_EL0 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:0]**

Clock frequency. Indicates the system counter clock frequency, in Hz.

### Accessing the CNTFRQ_EL0:

To access the CNTFRQ_EL0:

```
MRS <Xt>, CNTFRQ_EL0 ; Read CNTFRQ_EL0 into Xt
MSR CNTFRQ_EL0, <Xt> ; Write Xt to CNTFRQ_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.2 CNTHCTL_EL2, Counter-timer Hypervisor Control register

The CNTHCTL_EL2 characteristics are:

**Purpose**

Controls the generation of an event stream from the physical counter, and access from Non-secure EL1 to the physical counter and the Non-secure EL1 physical timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register CNTHCTL_EL2 is architecturally mapped to AArch32 System register CNTHCTL.

If EL2 is not implemented, this register is RES0 from EL3.

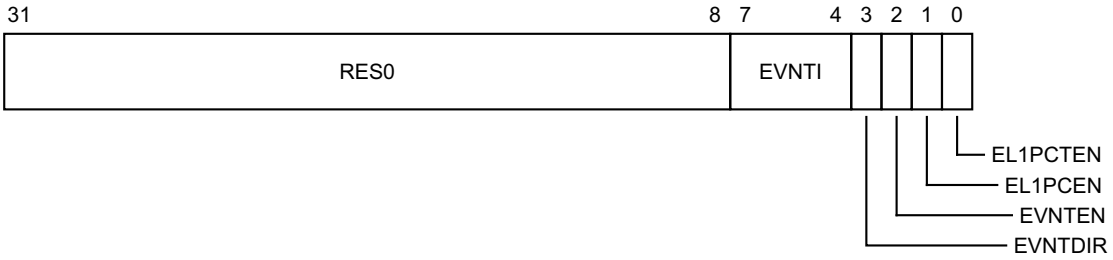
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTHCTL_EL2 is a 32-bit register.

#### Field descriptions

The CNTHCTL_EL2 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**EVNTI, bits [7:4]**

Selects which bit (0 to 15) of the corresponding counter register (CNTPCT_EL0 or CNTVCT_EL0) is the trigger for the event stream generated from that counter, when that stream is enabled.

**EVNTDIR, bit [3]**

Controls which transition of the counter register (CNTPCT_EL0 or CNTVCT_EL0) trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

0 A 0 to 1 transition of the trigger bit triggers an event.

1 A 1 to 0 transition of the trigger bit triggers an event.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EVNTEN, bit [2]**

Enables the generation of an event stream from the corresponding counter:

0 Disables the event stream.

1 Enables the event stream.

**EL1PCEN, bit [1]**

Traps Non-secure EL0 and EL1 accesses to the physical timer registers to EL2.

0 From AArch64 state: Non-secure EL0 and EL1 accesses to the CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0 are trapped to EL2.
From AArch32 state: Non-secure EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are trapped to EL2.

1 From AArch64 state: Non-secure EL0 and EL1 accesses to the CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0 are not trapped to EL2.
From AArch32 state: Non-secure EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are not trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

**EL1PCTEN, bit [0]**

Traps Non-secure EL0 and EL1 accesses to the physical counter register to EL2.

0 From AArch64 state: Non-secure EL0 and EL1 accesses to the CNTPCT_EL0 are trapped to EL2.
From AArch32 state: Non-secure EL0 and EL1 accesses to the CNTPCT are trapped to EL2.

1 From AArch64 state: Non-secure EL0 and EL1 accesses to the CNTPCT_EL0 are not trapped to EL2.
From AArch32 state: Non-secure EL0 and EL1 accesses to the CNTPCT are not trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

### Accessing the CNTHCTL_EL2:

To access the CNTHCTL_EL2:

```
MRS <Xt>, CNTHCTL_EL2 ; Read CNTHCTL_EL2 into Xt
MSR CNTHCTL_EL2, <Xt> ; Write Xt to CNTHCTL_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1110 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.3 CNTHP_CTL_EL2, Counter-timer Hypervisor Physical Timer Control register

The CNTHP_CTL_EL2 characteristics are:

**Purpose**

Control register for the EL2 physical timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register CNTHP_CTL_EL2 is architecturally mapped to AArch32 System register CNTHP_CTL.

If EL2 is not implemented, this register is RES0 from EL3.

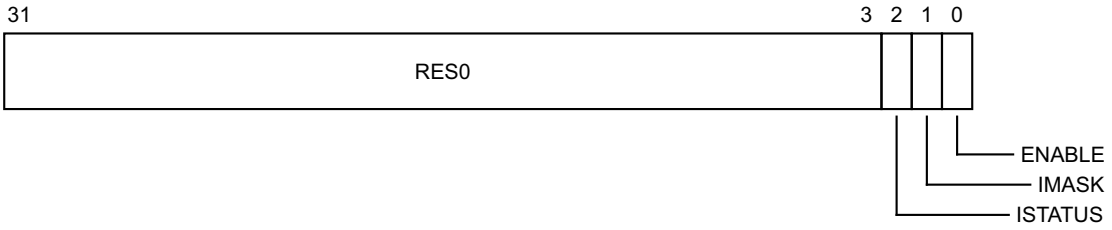
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTHP_CTL_EL2 is a 32-bit register.

## Field descriptions

The CNTHP_CTL_EL2 bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is asserted:

`0` Timer condition is not asserted.

`1` Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see *Operation of the CompareValue views of the timers* on page D6-1904 and *Operation of the TimerValue views of the timers* on page D6-1904.

This bit is read-only.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IMASK, bit [1]**

Timer output signal mask bit. Permitted values are:

0 Timer output signal is not masked.

1 Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

0 Timer disabled.

1 Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHP_TVAL_EL2 continues to count down.

**Note**

Disabling the output signal might be a power-saving option.

## Accessing the CNTHP_CTL_EL2:

To access the CNTHP_CTL_EL2:

```
MRS <Xt>, CNTHP_CTL_EL2 ; Read CNTHP_CTL_EL2 into Xt
MSR CNTHP_CTL_EL2, <Xt> ; Write Xt to CNTHP_CTL_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1110 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.4 CNTHP_CVAL_EL2, Counter-timer Hypervisor Physical Timer CompareValue register

The CNTHP_CVAL_EL2 characteristics are:

**Purpose**

Holds the compare value for the EL2 physical timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register CNTHP_CVAL_EL2 is architecturally mapped to AArch32 System register CNTHP_CVAL.

If EL2 is not implemented, this register is RES0 from EL3.

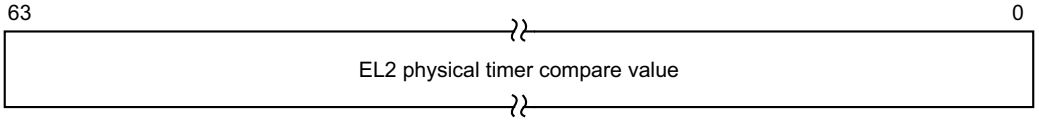
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTHP_CVAL_EL2 is a 64-bit register.

#### Field descriptions

The CNTHP_CVAL_EL2 bit assignments are:




**Bits [63:0]**

EL2 physical timer compare value.

#### Accessing the CNTHP_CVAL_EL2:

To access the CNTHP_CVAL_EL2:

```
MRS <Xt>, CNTHP_CVAL_EL2 ; Read CNTHP_CVAL_EL2 into Xt
MSR CNTHP_CVAL_EL2, <Xt> ; Write Xt to CNTHP_CVAL_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1110 | 0010 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.5 CNTHP_TVAL_EL2, Counter-timer Hypervisor Physical Timer TimerValue register

The CNTHP_TVAL_EL2 characteristics are:

**Purpose**

Holds the timer value for the EL2 physical timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

Where the value of CNTHP_CTL_EL2.ENABLE is 0:

- A write to CNTHP_TVAL_EL2 updates the register
- The value held in CNTHP_TVAL_EL2 continues to decrement
- A read of CNTHP_TVAL_EL2 returns an UNKNOWN value.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register CNTHP_TVAL_EL2 is architecturally mapped to AArch32 System register CNTHP_TVAL.

If EL2 is not implemented, this register is RES0 from EL3.

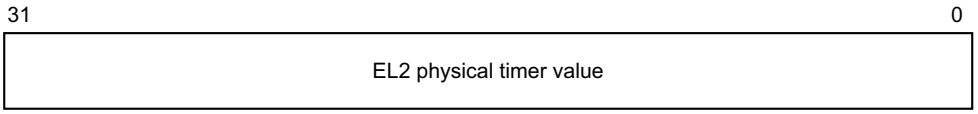
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTHP_TVAL_EL2 is a 32-bit register.

#### Field descriptions

The CNTHP_TVAL_EL2 bit assignments are:




**Bits [31:0]**

EL2 physical timer value.

#### Accessing the CNTHP_TVAL_EL2:

To access the CNTHP_TVAL_EL2:

```
MRS <Xt>, CNTHP_TVAL_EL2 ; Read CNTHP_TVAL_EL2 into Xt
MSR CNTHP_TVAL_EL2, <Xt> ; Write Xt to CNTHP_TVAL_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1110 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.6 CNTKCTL_EL1, Counter-timer Kernel Control register

The CNTKCTL_EL1 characteristics are:

**Purpose**

Controls the generation of an event stream from the virtual counter, and access from EL0 to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | RW | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register CNTKCTL_EL1 is architecturally mapped to AArch32 System register CNTKCTL.

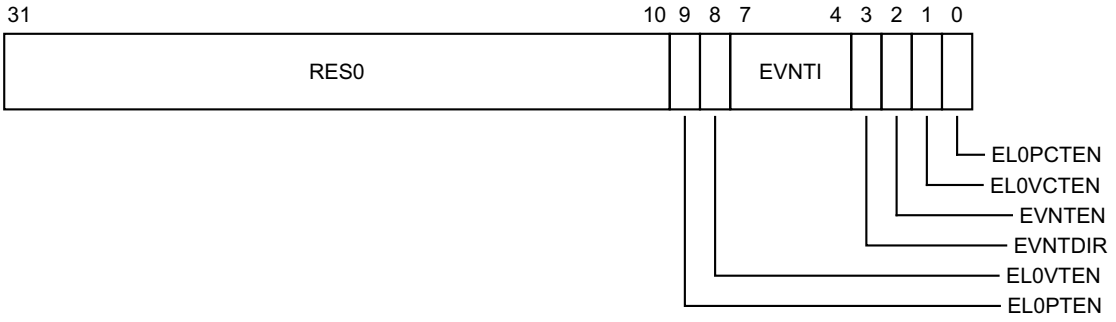
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTKCTL_EL1 is a 32-bit register.

#### Field descriptions

The CNTKCTL_EL1 bit assignments are:




**Bits [31:10]**

Reserved, RES0.

**EL0PTEN, bit [9]**

Traps EL0 accesses to the physical timer registers to EL1.

`0` EL0 using AArch64: EL0 accesses to the CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0 registers are trapped to EL1.

EL0 using AArch32: EL0 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL registers are trapped to EL1.

`1` EL0 using AArch64: EL0 accesses to the CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0 registers are not trapped to EL1.

EL0 using AArch32: EL0 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL registers are not trapped to EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EL0VTEN, bit [8]**

Traps EL0 accesses to the virtual timer registers to EL1.

0 EL0 using AArch64: EL0 accesses to the CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0 registers are trapped to EL1.

EL0 using AArch32: EL0 accesses to the CNTV_CTL, CNTV_CVAL, and CNTV_TVAL registers are trapped to EL1.

1 EL0 using AArch64: EL0 accesses to the CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0 registers are not trapped to EL1.

EL0 using AArch32: EL0 accesses to the CNTV_CTL, CNTV_CVAL, and CNTV_TVAL registers are not trapped to EL1.

**EVNTI, bits [7:4]**

Selects which bit (0 to 15) of the corresponding counter register (CNTPCT_EL0 or CNTVCT_EL0) is the trigger for the event stream generated from that counter, when that stream is enabled.

**EVNTDIR, bit [3]**

Controls which transition of the counter register (CNTPCT_EL0 or CNTVCT_EL0) trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

0 A 0 to 1 transition of the trigger bit triggers an event.

1 A 1 to 0 transition of the trigger bit triggers an event.

**EVNTEN, bit [2]**

Enables the generation of an event stream from the corresponding counter:

0 Disables the event stream.

1 Enables the event stream.

**EL0VCTEN, bit [1]**

Traps EL0 accesses to the frequency register and virtual counter register to Undefined mode.

0 EL0 using AArch64: EL0 accesses to the CNTVCT_EL0 are trapped to EL1.

EL0 using AArch64: EL0 accesses to the CNTFRQ_EL0 register are trapped to EL1, if CNTKCTL_EL1.EL0PCTEN is also 0.

EL0 using AArch32: EL0 accesses to the CNTVCT are trapped to EL1.

EL0 using AArch32: EL0 accesses to the CNTFRQ register are trapped to EL1, if CNTKCTL_EL1.EL0PCTEN is also 0.

1 EL0 using AArch64: EL0 accesses to the CNTFRQ_EL0 and CNTVCT_EL0 are not trapped to EL1.

EL0 using AArch32: EL0 accesses to the CNTFRQ and CNTVCT are not trapped to EL1.

**EL0PCTEN, bit [0]**

Traps EL0 accesses to the frequency register and physical counter register to EL1.

0 EL0 using AArch64: EL0 accesses to the CNTPCT_EL0 are trapped to EL1.

EL0 using AArch64: EL0 accesses to the CNTFRQ_EL0 register are trapped to EL1, if CNTKCTL_EL1.EL0VCTEN is also 0.

EL0 using AArch32: EL0 accesses to the CNTPCT are trapped to EL1.

EL0 using AArch32: EL0 accesses to the CNTFRQ and register are trapped to EL1, if CNTKCTL_EL1.EL0VCTEN is also 0.

1 EL0 using AArch64: EL0 accesses to the CNTFRQ_EL0 and CNTPCT_EL0 are not trapped to EL1.

EL0 using AArch32: EL0 accesses to the CNTFRQ and CNTPCT are not trapped to EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTKCTL_EL1:

To access the CNTKCTL_EL1:

```
MRS <Xt>, CNTKCTL_EL1 ; Read CNTKCTL_EL1 into Xt
MSR CNTKCTL_EL1, <Xt> ; Write Xt to CNTKCTL_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 000 | 1110 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.7 CNTP_CTL_EL0, Counter-timer Physical Timer Control register

The CNTP_CTL_EL0 characteristics are:

**Purpose**

Control register for the EL1 physical timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CNTHCTL_EL2.EL1PCEN==0, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If CNTKCTL_EL1.EL0PTEN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register CNTP_CTL_EL0 is architecturally mapped to AArch32 System register CNTP_CTL.

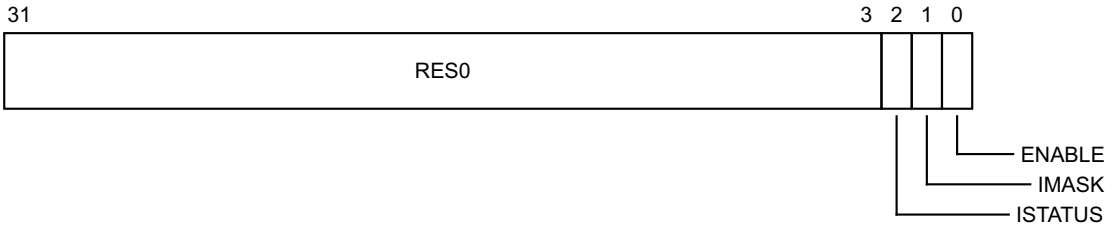
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTP_CTL_EL0 is a 32-bit register.

#### Field descriptions

The CNTP_CTL_EL0 bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is asserted:

`0` Timer condition is not asserted.

`1` Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information see *Operation of the CompareValue views of the timers* on page D6-1904 and *Operation of the TimerValue views of the timers* on page D6-1904.

This bit is read-only.

**IMASK, bit [1]**

Timer output signal mask bit. Permitted values are:

0 Timer output signal is not masked.

1 Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

0 Timer disabled.

1 Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL_EL0 continues to count down.

**Note**

Disabling the output signal might be a power-saving option.

## Accessing the CNTP_CTL_EL0:

To access the CNTP_CTL_EL0:

```
MRS <Xt>, CNTP_CTL_EL0 ; Read CNTP_CTL_EL0 into Xt
MSR CNTP_CTL_EL0, <Xt> ; Write Xt to CNTP_CTL_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.8 CNTP_CVAL_EL0, Counter-timer Physical Timer CompareValue register

The CNTP_CVAL_EL0 characteristics are:

**Purpose**

Holds the compare value for the EL1 physical timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CNTHCTL_EL2.EL1PCEN==0, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If CNTKCTL_EL1.EL0PTEN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register CNTP_CVAL_EL0 is architecturally mapped to AArch32 System register CNTP_CVAL.

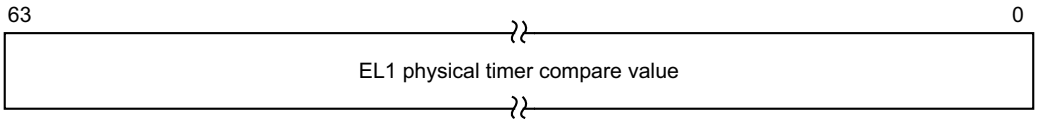
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTP_CVAL_EL0 is a 64-bit register.

#### Field descriptions

The CNTP_CVAL_EL0 bit assignments are:




**Bits [63:0]**

EL1 physical timer compare value.

#### Accessing the CNTP_CVAL_EL0:

To access the CNTP_CVAL_EL0:

```
MRS <Xt>, CNTP_CVAL_EL0 ; Read CNTP_CVAL_EL0 into Xt
MSR CNTP_CVAL_EL0, <Xt> ; Write Xt to CNTP_CVAL_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 0010 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.9 CNTP_TVAL_EL0, Counter-timer Physical Timer TimerValue register

The CNTP_TVAL_EL0 characteristics are:

**Purpose**

Holds the timer value for the EL1 physical timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

Where the value of CNTP_CTL_EL0.ENABLE is 0:

- A write to CNTP_TVAL_EL0 updates the register
- The value held in CNTP_TVAL_EL0 continues to decrement
- A read of CNTP_TVAL_EL0 returns an UNKNOWN value.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CNTHCTL_EL2.EL1PCEN==0, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If CNTKCTL_EL1.EL0PTEN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register CNTP_TVAL_EL0 is architecturally mapped to AArch32 System register CNTP_TVAL.

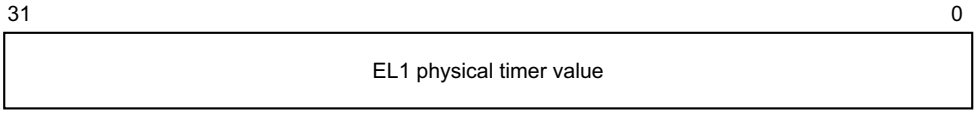
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTP_TVAL_EL0 is a 32-bit register.

#### Field descriptions

The CNTP_TVAL_EL0 bit assignments are:




**Bits [31:0]**

EL1 physical timer value.

#### Accessing the CNTP_TVAL_EL0:

To access the CNTP_TVAL_EL0:

```
MRS <Xt>, CNTP_TVAL_EL0 ; Read CNTP_TVAL_EL0 into Xt
MSR CNTP_TVAL_EL0, <Xt> ; Write Xt to CNTP_TVAL_EL0
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.10 CNTPCT_EL0, Counter-timer Physical Count register

The CNTPCT_EL0 characteristics are:

**Purpose**

Holds the 64-bit physical count value.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CNTHCTL_EL2.EL1PCTEN==0, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If CNTKCTL_EL1.EL0PCTEN==0, EL0 read accesses to this register are trapped to EL1.

**Configurations**

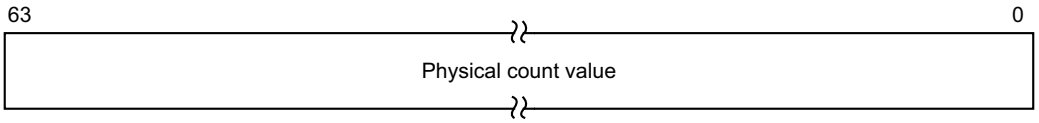
AArch64 System register CNTPCT_EL0 is architecturally mapped to AArch32 System register CNTPCT.

**Attributes**

CNTPCT_EL0 is a 64-bit register.

#### Field descriptions

The CNTPCT_EL0 bit assignments are:




**Bits [63:0]**

Physical count value.

#### Accessing the CNTPCT_EL0:

To access the CNTPCT_EL0:

```
MRS <Xt>, CNTPCT_EL0 ; Read CNTPCT_EL0 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.11 CNTPS_CTL_EL1, Counter-timer Physical Secure Timer Control register

The CNTPS_CTL_EL1 characteristics are:

**Purpose**

Control register for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | Config-RW | - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If SCR_EL3.ST==0, Secure EL1 accesses to this register are trapped to EL3.

**Configurations**

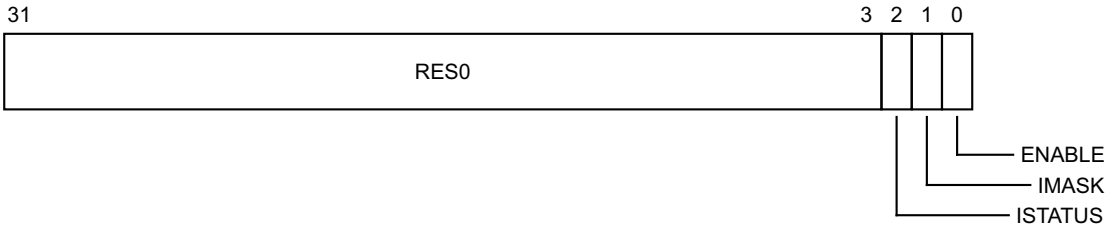
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTPS_CTL_EL1 is a 32-bit register.

#### Field descriptions

The CNTPS_CTL_EL1 bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is asserted:

`0` Timer condition is not asserted.

`1` Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see *Operation of the CompareValue views of the timers* on page D6-1904 and *Operation of the TimerValue views of the timers* on page D6-1904.

This bit is read-only.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IMASK, bit [1]**

Timer output signal mask bit. Permitted values are:

`0` Timer output signal is not masked.

`1` Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

`0` Timer disabled.

`1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTPS_TVAL_EL1 continues to count down.

**Note**

Disabling the output signal might be a power-saving option.

## Accessing the CNTPS_CTL_EL1:

To access the CNTPS_CTL_EL1:

```
MRS <Xt>, CNTPS_CTL_EL1 ; Read CNTPS_CTL_EL1 into Xt
MSR CNTPS_CTL_EL1, <Xt> ; Write Xt to CNTPS_CTL_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 111 | 1110 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.12 CNTPS_CVAL_EL1, Counter-timer Physical Secure Timer CompareValue register

The CNTPS_CVAL_EL1 characteristics are:

**Purpose**

Holds the compare value for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | Config-RW | - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If SCR_EL3.ST==0, Secure EL1 accesses to this register are trapped to EL3.

**Configurations**

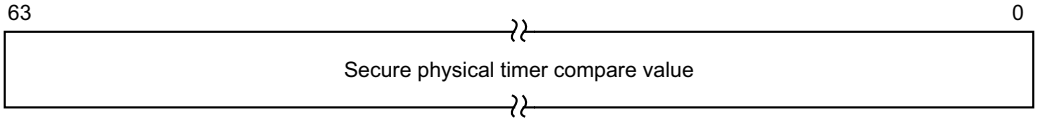
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTPS_CVAL_EL1 is a 64-bit register.

#### Field descriptions

The CNTPS_CVAL_EL1 bit assignments are:




**Bits [63:0]**

Secure physical timer compare value.

#### Accessing the CNTPS_CVAL_EL1:

To access the CNTPS_CVAL_EL1:

```
MRS <Xt>, CNTPS_CVAL_EL1 ; Read CNTPS_CVAL_EL1 into Xt
MSR CNTPS_CVAL_EL1, <Xt> ; Write Xt to CNTPS_CVAL_EL1
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 111 | 1110 | 0010 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.13 CNTPS_TVAL_EL1, Counter-timer Physical Secure Timer TimerValue register

The CNTPS_TVAL_EL1 characteristics are:

**Purpose**

Holds the timer value for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | Config-RW | - | RW | RW |

Where the value of CNTPS_CTL_EL1.ENABLE is 0:

- A write to CNTPS_TVAL_EL1 updates the register
- The value held in CNTPS_TVAL_EL1 continues to decrement
- A read of CNTPS_TVAL_EL1 returns an UNKNOWN value.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If SCR_EL3.ST==0, Secure EL1 accesses to this register are trapped to EL3.

**Configurations**

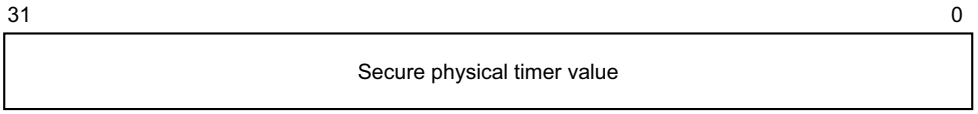
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTPS_TVAL_EL1 is a 32-bit register.

#### Field descriptions

The CNTPS_TVAL_EL1 bit assignments are:




**Bits [31:0]**

Secure physical timer value.

#### Accessing the CNTPS_TVAL_EL1:

To access the CNTPS_TVAL_EL1:

```
MRS <Xt>, CNTPS_TVAL_EL1 ; Read CNTPS_TVAL_EL1 into Xt
MSR CNTPS_TVAL_EL1, <Xt> ; Write Xt to CNTPS_TVAL_EL1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 111 | 1110 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.14 CNTV_CTL_EL0, Counter-timer Virtual Timer Control register

The CNTV_CTL_EL0 characteristics are:

**Purpose**

Control register for the virtual timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CNTKCTL_EL1.EL0VTEN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register CNTV_CTL_EL0 is architecturally mapped to AArch32 System register CNTV_CTL.

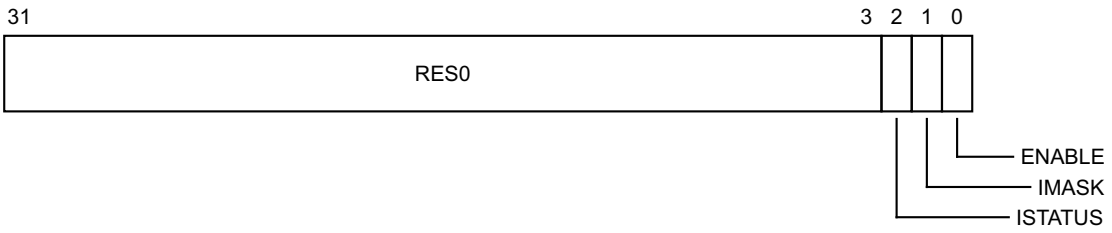
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTV_CTL_EL0 is a 32-bit register.

## Field descriptions

The CNTV_CTL_EL0 bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is asserted:

`0` Timer condition is not asserted.

`1` Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see *Operation of the CompareValue views of the timers* on page D6-1904 and *Operation of the TimerValue views of the timers* on page D6-1904.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit is read-only.

**IMASK, bit [1]**

Timer output signal mask bit. Permitted values are:

0 Timer output signal is not masked.

1 Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

0 Timer disabled.

1 Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL_EL0 continues to count down.

**Note**

Disabling the output signal might be a power-saving option.

## Accessing the CNTV_CTL_EL0:

To access the CNTV_CTL_EL0:

```
MRS <Xt>, CNTV_CTL_EL0 ; Read CNTV_CTL_EL0 into Xt
MSR CNTV_CTL_EL0, <Xt> ; Write Xt to CNTV_CTL_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.15 CNTV_CVAL_EL0, Counter-timer Virtual Timer CompareValue register

The CNTV_CVAL_EL0 characteristics are:

**Purpose**

Holds the compare value for the virtual timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CNTKCTL_EL1.EL0VTEN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register CNTV_CVAL_EL0 is architecturally mapped to AArch32 System register CNTV_CVAL.

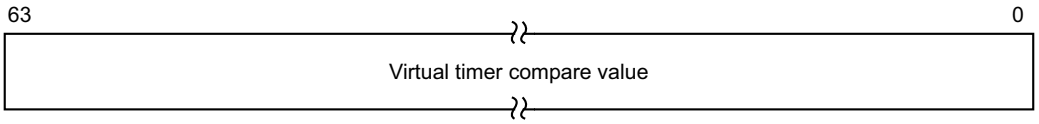
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTV_CVAL_EL0 is a 64-bit register.

#### Field descriptions

The CNTV_CVAL_EL0 bit assignments are:




**Bits [63:0]**

Virtual timer compare value.

#### Accessing the CNTV_CVAL_EL0:

To access the CNTV_CVAL_EL0:

```
MRS <Xt>, CNTV_CVAL_EL0 ; Read CNTV_CVAL_EL0 into Xt
MSR CNTV_CVAL_EL0, <Xt> ; Write Xt to CNTV_CVAL_EL0
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 0011 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.16 CNTV_TVAL_EL0, Counter-timer Virtual Timer TimerValue register

The CNTV_TVAL_EL0 characteristics are:

**Purpose**

Holds the timer value for the virtual timer.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | RW | RW | RW | RW | RW |

Where the value of CNTV_CTL_EL0.ENABLE is 0:

- A write to CNTV_TVAL_EL0 updates the register
- The value held in CNTV_TVAL_EL0 continues to decrement
- A read of CNTV_TVAL_EL0 returns an UNKNOWN value.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CNTKCTL_EL1.EL0VTEN==0, EL0 accesses to this register are trapped to EL1.

**Configurations**

AArch64 System register CNTV_TVAL_EL0 is architecturally mapped to AArch32 System register CNTV_TVAL.

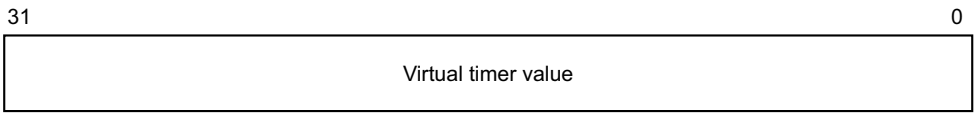
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTV_TVAL_EL0 is a 32-bit register.

#### Field descriptions

The CNTV_TVAL_EL0 bit assignments are:




**Bits [31:0]**

Virtual timer value.

#### Accessing the CNTV_TVAL_EL0:

To access the CNTV_TVAL_EL0:

```
MRS <Xt>, CNTV_TVAL_EL0 ; Read CNTV_TVAL_EL0 into Xt
MSR CNTV_TVAL_EL0, <Xt> ; Write Xt to CNTV_TVAL_EL0
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.17 CNTVCT_EL0, Counter-timer Virtual Count register

The CNTVCT_EL0 characteristics are:

**Purpose**

Holds the 64-bit virtual count value.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | RO | RO | RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

Subject to the prioritization rules:

If CNTKCTL_EL1.EL0VCTEN==0, EL0 read accesses to this register are trapped to EL1.

**Configurations**

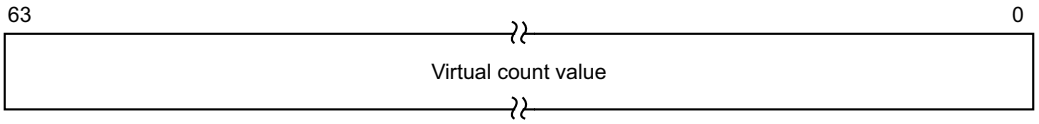
AArch64 System register CNTVCT_EL0 is architecturally mapped to AArch32 System register CNTVCT.

**Attributes**

CNTVCT_EL0 is a 64-bit register.

#### Field descriptions

The CNTVCT_EL0 bit assignments are:




**Bits [63:0]**

Virtual count value.

#### Accessing the CNTVCT_EL0:

To access the CNTVCT_EL0:

```
MRS <Xt>, CNTVCT_EL0 ; Read CNTVCT_EL0 into Xt
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 011 | 1110 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### D7.5.18 CNTVOFF_EL2, Counter-timer Virtual Offset register

The CNTVOFF_EL2 characteristics are:

**Purpose**

Holds the 64-bit virtual offset.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch64 System register CNTVOFF_EL2 is architecturally mapped to AArch32 System register CNTVOFF.

If EL2 is not implemented, this register is RES0 from EL3.

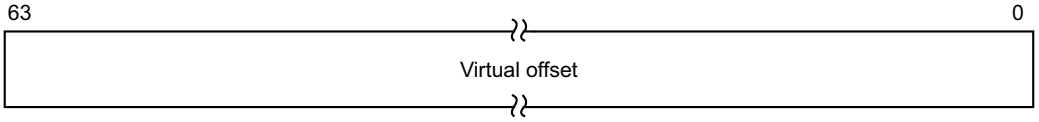
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTVOFF_EL2 is a 64-bit register.

#### Field descriptions

The CNTVOFF_EL2 bit assignments are:




**Bits [63:0]**

Virtual offset.

#### Accessing the CNTVOFF_EL2:

To access the CNTVOFF_EL2:

```
MRS <Xt>, CNTVOFF_EL2 ; Read CNTVOFF_EL2 into Xt
MSR CNTVOFF_EL2, <Xt> ; Write Xt to CNTVOFF_EL2
```

Register access is encoded as follows:

| op0 | op1 | CRn | CRm | op2 |
|---|---|---|---|---|
| 11 | 100 | 1110 | 0000 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Part E
## The AArch32 Application Level Architecture

# Chapter E1
# The AArch32 Application Level Programmers' Model

This chapter gives an Application level description of the programmers' model for software executing in AArch32 state. This means it describes execution in EL0 when EL0 is using AArch32. It contains the following sections:


- *About the Application level programmers' model* on page E1-2310.
- *Additional information about the programmers' model in AArch32 state* on page E1-2311.
- *Advanced SIMD and floating-point instructions* on page E1-2325.
- *Conceptual coprocessor support* on page E1-2353.
- *Exceptions* on page E1-2354.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## E1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for the development of applications that will execute in AArch32 state.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of that system information is needed to put the Application level programmers' model into context.

Depending on the implementation, the architecture supports multiple levels of execution privilege. These privilege levels are indicated by different *Exception levels* that number upwards from EL0, where EL0 corresponds to the lowest privilege level and is often described as *unprivileged*. The Application level programmers' model is the programmers' model for software executing at EL0. For more information see *ARMv8 architectural concepts* on page A1-33.

System software determines the Exception level, and therefore the level of privilege, at which application software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged. This has the following effects:

- It means that the operating system can allocate system resources to an application in a unique or shared manner.
- It provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some System level understanding is helpful, and if appropriate it gives a reference to the System level description.

When included in an implementation:

- EL3 provides two Security states, Secure and Non-secure. Secure state provides additional hardware features that support the development of secure applications.
- EL2 provides virtualization of operation in Non-secure state.

However, application level software is generally unaware of its Security state, and of any virtualization. For more information, see *The ARMv8-A security model* on page G1-3753 and *The effect of implementing EL2 on the Exception model* on page G1-3756.

**Note**

- When an implementation includes EL3, application and operating system software normally executes in Non-secure state.
- EL2, that provides the virtualization features, is implemented only in Non-secure state.
- Older documentation, describing implementations or architecture versions that support only two privilege levels, often refers to execution at EL1 as *privileged* execution.
- In this manual, the terms *CONSTRAINED UNPREDICTABLE*, *IMPLEMENTATION DEFINED*. *OPTIONAL*, *RES0*, *RES1*, *UNDEFINED*, *UNKNOWN*, and *UNPREDICTABLE* have ARM-specific meanings, as defined in the *Glossary*. In body text, these terms are shown in SMALL CAPS, for example IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E1.2 Additional information about the programmers' model in AArch32 state

The following sections give more information about the Application level programmers' model in AArch32 state:

- *Instruction sets, arithmetic operations, and register files*.
- *Core data types and arithmetic in AArch32 state*.
- *The general-purpose registers, and the PC, in AArch32 state* on page E1-2316.
- *Process state, PSTATE* on page E1-2319.

### E1.2.1 Instruction sets, arithmetic operations, and register files

The A32 and T32 instruction sets both provide a wide range of integer arithmetic and logical operations, that operate on a register file of sixteen 32-bit registers, that are comprised of the AArch32 general-purpose registers and the PC. As described in *The general-purpose registers, and the PC, in AArch32 state* on page E1-2316, these registers include the registers SP (R13) and LR (R14), which have specialized uses. *Core data types and arithmetic in AArch32 state* gives more information about these operations.

In addition, an implementation that implements the T32 and A32 instruction sets includes both:

- Scalar floating-point instructions.
- The Advanced SIMD vector instructions.

Floating-point and vector instructions operate on a separate common register file, described in *The SIMD and floating-point register file* on page E1-2325. *Advanced SIMD and floating-point instructions* on page E1-2325 gives more information about these instructions.

### E1.2.2 Core data types and arithmetic in AArch32 state

When executing in AArch32 state, a PE supports the following data types in memory:

**Byte** 8 bits
**Halfword** 16 bits
**Word** 32 bits
**Doubleword** 64 bits.

PE registers are 32 bits in size. The instruction sets provide instructions that use the following data types for data held in registers:

- 32-bit pointers.
- Unsigned or signed 32-bit integers.
- Unsigned 16-bit or 8-bit integers, held in zero-extended form.
- Signed 16-bit or 8-bit integers, held in sign-extended form.
- Two 16-bit integers packed into a register.
- Four 8-bit integers packed into a register.
- Unsigned or signed 64-bit integers held in two registers.

Load and store operations can transfer bytes, halfwords, or words to and from memory. Loads of bytes or halfwords zero-extend or sign-extend the data as it is loaded, as specified in the appropriate load instruction.

The instruction sets include load and store operations that transfer two or more words to and from memory. Software can load and store doublewords using these instructions.

**Note**

For information about the atomicity of memory accesses see *Atomicity in the ARM architecture* on page E2-2368.

When any of the data types is described as *unsigned*, the N-bit data value represents a non-negative integer in the range 0 to $2^N$-1, using normal binary format.

When any of these types is described as *signed*, the N-bit data value represents an integer in the range $-2^{(N-1)}$ to $+2^{(N-1)}$-1, using two's complement format.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The instructions that operate on packed halfwords or bytes include some multiply instructions that use only one of two halfwords, and SIMD instructions that perform parallel addition or subtraction on all of the halfwords or bytes.

**Note**

These SIMD instructions operate on values held in the general-purpose registers, and must not be confused with the Advanced SIMD instructions that operate on a separate register file that provides registers of up 128 bits.

Direct instruction support for 64-bit integers is limited, and most 64-bit operations require sequences of two or more instructions to synthesize them.

## Integer arithmetic

The instruction set provides a wide range of operations on the values in registers, including bitwise logical operations, shifts, additions, subtractions, multiplications, and divisions. The pseudocode described in Appendix K11 *ARM Pseudocode Definition* defines these operations, usually in one of three ways:

- By direct use of the pseudocode operators and built-in functions defined in *Operators* on page K11-5656.
- By use of pseudocode helper functions defined in the main text. See Appendix K12 *Pseudocode Index*.
- By a sequence of the form:
  1. Use of the `SInt()`, `UInt()`, and `Int()` built-in functions defined in *Converting bitstrings to integers* on page K11-5669 to convert the bitstring contents of the instruction operands to the unbounded integers that they represent as two's complement or unsigned integers.
  2. Use of mathematical operators, built-in functions and helper functions on those unbounded integers to calculate other such integers.
  3. Use of either the bitstring extraction operator defined in *Bitstring concatenation and slicing* on page K11-5657 or of the saturation helper functions described in *Pseudocode description of saturation* on page E1-2315 to convert an unbounded integer result into a bitstring result that can be written to a register.

### *Shift and rotate operations*

The following types of shift and rotate operations are used in instructions:

**Logical Shift Left**

(`LSL`) moves each bit of a bitstring left by a specified number of bits. Zeros are shifted in at the right end of the bitstring. Bits that are shifted off the left end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

**Logical Shift Right**

(`LSR`) moves each bit of a bitstring right by a specified number of bits. Zeros are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

**Arithmetic Shift Right**

(`ASR`) moves each bit of a bitstring right by a specified number of bits. Copies of the leftmost bit are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

**Rotate Right** (`ROR`) moves each bit of a bitstring right by a specified number of bits. Each bit that is shifted off the right end of the bitstring is re-introduced at the left end. The last bit shifted off the right end of the bitstring can be produced as a carry output.

**Rotate Right with Extend**

(`RRX`) moves each bit of a bitstring right by one bit. A carry input is shifted in at the left end of the bitstring. The bit shifted off the right end of the bitstring can be produced as a carry output.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Pseudocode description of shift and rotate operations*

These shift and rotate operations are supported in pseudocode by the following functions:

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);

// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;

// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;

// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;

// ROR_C()
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);

// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;

// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);

// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

### *Pseudocode description of addition and subtraction*

In pseudocode, addition and subtraction can be performed on any combination of unbounded integers and bitstrings, provided that if they are performed on two bitstrings, the bitstrings must be identical in length. The result is another unbounded integer if both operands are unbounded integers, and a bitstring of the same length as the bitstring operand or operands otherwise. For the definition of these operations, see *Addition and subtraction* on page K11-5658.

The main addition and subtraction instructions can produce status information about both unsigned carry and signed overflow conditions. When necessary, multi-word additions and subtractions can be synthesized from this status information. In pseudocode the `AddWithCarry()` function provides an addition with a carry input and a set of output Condition flags including carry output and overflow:

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

An important property of the `AddWithCarry()` function is that if:

```
(result, nzcv) = AddWithCarry(x, NOT(y), carry_in)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Then:

- If `carry_in == '1'`, then `result == x-y` with:
  - `nzcv<0> == '1'` if signed overflow occurred during the subtraction.
  - `nzcv<1> == '1'` if unsigned borrow did not occur during the subtraction, that is, if x≥y.
- If `carry_in == '0'`, then `result == x-y-1` with:
  - `nzcv<0> == '1'` if signed overflow occurred during the subtraction.
  - `nzcv<1> == '1'` if unsigned borrow did not occur during the subtraction, that is, if x≥y.

Taken together, this means that the `carry_in` and `nzcv<1>` output in `AddWithCarry()` calls can act as NOT borrow flags for subtractions as well as carry flags for additions.

### *Pseudocode description of saturation*

Some instructions perform *saturating arithmetic*, that is, if the result of the arithmetic overflows the destination signed or unsigned N-bit integer range, the result produced is the largest or smallest value in that range, rather than wrapping around modulo $2^N$. This is supported in pseudocode by:

- The `SignedSatQ()` and `UnsignedSatQ()` functions when an operation requires, in addition to the saturated result, a Boolean argument that indicates whether saturation occurred.
- The `SignedSat()` and `UnsignedSat()` functions when only the saturated result is required.

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);

// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);

// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;

// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

`SatQ(i, N, unsigned)` returns either `UnsignedSatQ(i, N)` or `SignedSatQ(i, N)` depending on the value of its third argument, and `Sat(i, N, unsigned)` returns either `UnsignedSat(i, N)` or `SignedSat(i, N)` depending on the value of its third argument:

```
// SatQ()
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);

// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

### E1.2.3 The general-purpose registers, and the PC, in AArch32 state

In the AArch32 Application level view, a PE has:

- Fifteen general-purpose 32-bit registers, R0 to R14, of which R13 and R14 have alternative names reflecting how they are, or can be, used:
  - R13 is usually identified as SP.
  - R14 is usually identified as LR.
- The PC (*program counter*), that can be described as R15.

The specialized uses of the SP (R13), LR (R14), and PC (R15) are:

**SP, the stack pointer**

The PE uses SP as a pointer to the active stack.

In the T32 instruction set, some instructions cannot access SP, and for most T32 instructions ARM deprecates using SP as a general-purpose register. The only T32 instructions for which the use of SP is not deprecated are those designed to use SP as a stack pointer.

The A32 instruction set provides more general access to the SP, and it can be used as a general-purpose register. However, ARM deprecates the use of SP for any purpose other than as a stack pointer.

**Note**

- Using SP for any purpose other than as a stack pointer is likely to break the requirements of operating systems, debuggers, and other software systems, causing them to malfunction.
- Before ARMv8, for most T32 instructions, using SP as a general-purpose register was UNPREDICTABLE. In ARMv8, most of these uses of SP behave predictably, but are deprecated by ARM. The instruction descriptions give more information.

Software can refer to SP as R13.

**LR, the link register**

The link register can be used to hold return link information, and some cases described in this manual require this use of the LR. When software does not require the LR for linking, it can use it for other purposes. Software can refer to LR as R14.

**PC, the program counter**

- When executing an A32 instruction, PC reads as the address of the current instruction plus 8.
- When executing a T32 instruction, PC reads as the address of the current instruction plus 4.
- Writing an address to PC causes a branch to that address.

Most T32 instructions cannot access PC.

The A32 instruction set provides more general access to the PC, and many A32 instructions can use the PC as a general-purpose register. However, ARM deprecates the use of PC for any purpose other than as the program counter. See *Writing to the PC* on page E1-2317 for more information.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

Software can refer to PC as R15.

See *AArch32 general-purpose registers, the PC, and the Special-purpose registers* on page G1-3763 for the system level view of these registers.

**Note**

In general, ARM strongly recommends using the names SP, LR and PC instead of R13, R14 and R15. However, sometimes it is simpler to use the R13-R15 names when referring to a group of registers. For example, it is simpler to refer to *registers R8 to R15*, rather than to *registers R8 to R12, the SP, LR and PC*. These two descriptions of the group of registers have exactly the same meaning.

### Writing to the PC

In the A32 and T32 instruction sets, many data-processing instructions can write to the PC. Writes to the PC are handled as follows:

- In T32 state, the following 16-bit T32 instruction encodings branch to the value written to the PC:
  - Encoding T2 of *ADD, ADDS (register)* on page F6-2589.
  - Encoding T1 of *MOV, MOVS (register)* on page F6-2813.

  The value written to the PC is forced to be halfword-aligned by ignoring its least significant bit, treating that bit as being 0.
- The `B`, `BL`, `CBNZ`, `CBZ`, `CHKA`, `HB`, `HBL`, `HBLP`, `HBP`, `TBB`, and `TBH` instructions remain in the same instruction set state and branch to the value written to the PC.

  The definition of each of these instructions ensures that the value written to the PC is correctly aligned for the current instruction set state.
- The `BLX` (immediate) instruction switches between A32 and T32 states and branches to the value written to the PC. Its definition ensures that the value written to the PC is correctly aligned for the new instruction set state.
- The following instructions write a value to the PC, treating that value as an interworking address to branch to, with low-order bits that determine the new instruction set state:
  - `BLX` (register), `BX`, and `BXJ`.
  - `LDR` instructions with `<Rt>` equal to the PC.
  - `POP` and all forms of `LDM` except `LDM` (exception return), when the register list includes the PC.
  - In A32 state only, `ADC`, `ADD`, `ADR`, `AND`, `ASR` (immediate), `BIC`, `EOR`, `LSL` (immediate), `LSR` (immediate), `MOV`, `MVN`, `ORR`, `ROR` (immediate), `RRX`, `RSB`, `RSC`, `SBC`, and `SUB` instructions with `<Rd>` equal to the PC and without flag-setting specified.

  For details of how an interworking address specifies the new instruction set state and instruction address, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

  **Note**

  The register-shifted register instructions, that are available only in the A32 instruction set and are summarized in *Data-processing register (register shift)* on page F4-2519, are UNPREDICTABLE if they attempt to write to the PC.
- Some instructions are treated as exception return instructions, and write both the PC and the CPSR. For more information, including which instructions are exception return instructions, see *Exception return to an Exception level using AArch32* on page G1-3796.
- Some instructions cause an exception, and the exception handler address is written to the PC as part of the exception entry.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Pseudocode description of operations on the AArch32 general-purpose registers and the PC

In pseudocode, the uses of the `R[]` function, with an index parameter `n`, are:

- Reading or writing R0-R12, SP, and LR, using n = 0-12, 13, and 14 respectively.
- Reading the PC, using n = 15.

*Pseudocode description of general-purpose register and PC operations* on page G1-3765 describes accesses to these registers.

Descriptions of A32 store instructions that store the PC value use the `PCStoreValue()` pseudocode function to specify the PC value stored by the instruction:

```
// PCStoreValue()
// ==============

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before ARMv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

Writing an address to the PC causes either a simple branch to that address or an *interworking* branch that also selects the instruction set to execute after the branch. A simple branch is performed by the `BranchWritePC()` function:

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        address<1:0> = '00';
    else
        address<0> = '0';
    BranchTo(address, BranchType_UNKNOWN);
```

An interworking branch is performed by the `BXWritePC()` function:

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address)
    if address<0> == '1' then
        SelectInstrSet(InstrSet_T32);
        address<0> = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the processor takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address<1> == '1' && ConstrainUnpredictableBool() then
            address<1> = '0';
    BranchTo(address, BranchType_UNKNOWN);
```

The `LoadWritePC()` and `ALUWritePC()` functions are used for two cases where the behavior was systematically modified between architecture versions:

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address);

// ALUWritePC()
// ============
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
ALUWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address);
    else
        BranchWritePC(address);
```

### E1.2.4 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

The following PSTATE information is accessible at EL0:

**The condition flags**

Flag-setting instructions set these. They are:

**N** Negative condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:

- 1 if the result is negative.
- 0 if the result is positive or zero.

**Z** Zero condition flag. Set to:

- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

**C** Carry condition flag. Set to:

- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

**V** Overflow condition flag. Set to:

- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

Conditional instructions test the N, Z, C, and V condition flags, combining them with the *condition code* for the instruction, to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see *Conditional execution* on page F2-2447.

**The overflow or saturation flag**

**Q** Some instructions can set this. For those instructions that can, the PE:

- Sets it to 1 if the instruction indicates overflow or saturation.
- Leaves it unchanged otherwise.

For more information, see *Pseudocode description of saturation* on page E1-2315.

**The greater than or equal flags**

**GE[3:0]** The instructions described in *Parallel addition and subtraction instructions* on page F1-2418 update these to indicate the results from individual bytes or halfwords of the operation. These flags can control a later `SEL` instruction. For more information, see *SEL* on page F6-2960.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PSTATE also contains *PE state controls*. There is no direct access to these from application level instructions, but they can be changed by side-effects of application level instructions. They are:

**Instruction set state**

**J, T** The current instruction set state, as shown in Table E1-1. In ARMv8, the J bit is RES0, see the Note in this section.

**Table E1-1 PSTATE.{J, T} encoding**

| J | T | Instruction set state |
|---|---|---|
| 0 | 0 | A32 |
| 0 | 1 | T32 |

**A32** The PE is executing the A32 instruction set, summarized in both:

- Chapter F4 *A32 Base Instruction Set Encoding*.
- Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*.

**T32** The PE is executing the T32 instruction set, summarized in both:

- Chapter F3 *T32 Base Instruction Set Encoding*.
- Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*.

**Note**

**Encoding with J==1 before ARMv8, Jazelle and T32EE states**

In previous versions of the ARM architecture, the encoding {1, 0} selected Jazelle state, and encoding {1, 1} selected T32EE state. ARMv8 does not support either of these states, and these are encodings for unimplemented instruction set states, see *Unimplemented instruction sets* on page G1-3772.

ARMv8 AArch32 state requires a Trivial Jazelle implementation, see *Trivial implementation of the Jazelle extension* on page G1-3772.

**The IT block state**

**IT[7:0]** The If-Then controls for the T32 `IT` instruction, that applies to the *IT block* of instructions that immediately follows the `IT` instruction. See *IT* on page F6-2690 for a description of the `IT` instruction and its associated `IT` block.

PSTATE.IT divides into two subfields:

**IT[7:5]** Holds the *base condition* for the current IT block. The base condition is the top three bits of the condition code specified by the `<firstcond>` field of the IT instruction.

**IT[4:0]** Encodes:

- The size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is implied by the position of the least significant 1 in this field, as shown in Table E1-2 on page E1-2321.
- For each instruction in the IT block, the least significant bit of the condition code. This is encoded where Table E1-2 on page E1-2321 shows N<index>.

  **Note**

  Changing the least significant bit of a condition code from 0 to 1 has the effect of inverting the condition code.

Both subfields are all zeros when no IT block is active.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When an IT instruction is executed, PSTATE.IT is set according to the <firstcond> field of the instruction and the *Then* and *Else* (T and E) parameters in the instruction. See *IT* on page F6-2690.

When permitted, an instruction in an IT block is conditional, see *Conditional instructions* on page F1-2409 and *Conditional execution* on page F2-2447. The condition code used is the current value of IT[7:4]. When an instruction in an IT block completes its execution normally, PSTATE.IT advances to the next line of Table E1-2. A few instructions, for example BKPT, cannot be conditional and therefore are always executed, ignoring the current value of PSTATE.IT.

For details of what happens if an instruction in an IT block takes an exception, see *Overview of exception entry* on page G1-3781.

An instruction that might complete its normal execution by branching is only permitted in an IT block as the last instruction in the block. This means that normal execution of the instruction always results in PSTATE.IT advancing to normal execution.

Table E1-2 shows the effect of the If-Then controls. N<index> represents an instruction in the IT block. For example, N1 is the first instruction in the block, and N2 is the second instruction in the block.

**Table E1-2 Effect of IT block state controls**

| IT bits[a] | | | | | | Notes |
|---|---|---|---|---|---|---|
| [7:5] | [4] | [3] | [2] | [1] | [0] | |
| cond_base | N1 | N2 | N3 | N4 | 1 | Entry point for an IT block with four instructions |
| cond_base | N1 | N2 | N3 | 1 | 0 | Entry point for an IT block with three instructions |
| cond_base | N1 | N2 | 1 | 0 | 0 | Entry point for an IT block with two instructions |
| cond_base | N1 | 1 | 0 | 0 | 0 | Entry point for an IT block with one instruction |
| 000 | 0 | 0 | 0 | 0 | 0 | Normal execution, not in an IT block |

a. Combinations of the IT bits not shown in this table are reserved.

On a branch or an exception return, if PSTATE.IT is set to a value that is not consistent with the instruction stream being branched to or returned to, then instruction execution is UNPREDICTABLE.

PSTATE.IT affects instruction execution only in T32 state. In A32 state, PSTATE.IT must be 0b00000000, otherwise the behavior is UNPREDICTABLE.

**Endianness mapping**

**E** For data accesses, controls the endianness:

**0** Little-endian.

**1** Big-endian.

If an implementation does not provide:

- Big-endian support for data accesses, this bit is RES0.
- Little-endian support for data accesses, this bit is RES1.

Instruction fetches are always little-endian, and ignore PSTATE.E.

## Accessing PSTATE fields at EL0

The following sections describe which PSTATE fields can be directly accessed at EL0, and how they can be accessed:

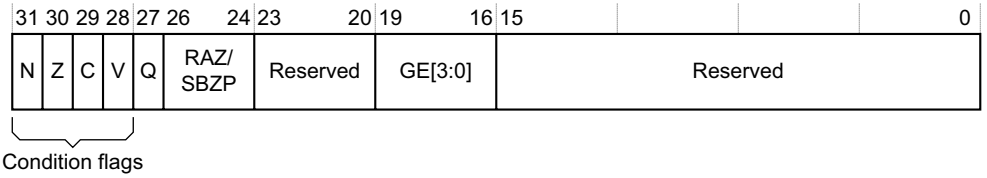
- *The Application Program Status Register, APSR* on page E1-2322.
- *The SETEND instruction* on page E1-2322.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### The Application Program Status Register, APSR

At EL0, some PSTATE fields can be accessed using the Special-purpose *Application Program Status Register* (APSR). The APSR can be directly read using the MRS instruction, and directly written using the MSR (register) and MSR (immediate) instructions.

The APSR bit assignments are:




**N, Z, C, V, bits [31:28]**

The PSTATE condition flags.

**Q, bit [27]** The PSTATE overflow or saturation flag.

**Bits[26:24]** Reserved, RAZ/SBZP. Software can use `MSR` instructions that write the top byte of the APSR without using a read-modify-write sequence. If it does this, it must write zeros to bits[26:24].

**Bits[23:20, 15:0]**

Reserved bits that are allocated to system features, or are available for future expansion. Unprivileged execution ignores writes to fields that are accessible only at EL1 or higher. However, application level software that writes to the APSR must treat reserved bits as *Do-Not-Modify* (DNM) bits. For more information about the reserved bits, see *The Current Program Status Register, CPSR* on page G1-3768.

**GE[3:0], bits [19:16]**

The PSTATE greater than or equal flags.

The other PSTATE fields cannot be accessed by using the APSR.

The system level alias for the APSR is the CPSR. The CPSR is a superset of the APSR. See *The Current Program Status Register, CPSR* on page G1-3768.

Writes to the PSTATE fields have side-effects on various aspects of PE operation. All of these side-effects, except side-effects on memory accesses associated with fetching instructions, are synchronous to the APSR write. This means they are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

### The SETEND instruction

The A32 and T32 instruction sets both include an instruction to manipulate PSTATE.E:

`SETEND BE` Sets PSTATE.E to 1, for big-endian operation.

`SETEND LE` Sets PSTATE.E to 0, for little-endian operation.

The `SETEND` instruction is unconditional. For more information, see *SETEND* on page F6-2962. ARM deprecates use of the `SETEND` instruction.

## Pseudocode description of PSTATE PE state fields

The following pseudocode functions return the current instruction set and select a new instruction set:

```
// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()

    if UsingAArch32() then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;

// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';

    return;
```

PSTATE.IT advances after normal execution of an IT block instruction. This is described by the `ITAdvance()` pseudocode function:

```
// AArch32.ITAdvance()
// ===================

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

The following functions test whether the current instruction is in an IT block, and whether it is the last instruction in an IT block.

```
// InITBlock()
// ===========

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;


// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

The `BigEndian()` pseudocode function tests whether big-endian data memory accesses are currently selected.

```
// BigEndian()
// ===========

boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR_EL1.E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### E1.2.5 Jazelle support

ARMv8 requires AArch32 state to include a trivial implementation of the Jazelle extension, as described in *Trivial implementation of the Jazelle extension* on page G1-3772.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## E1.3 Advanced SIMD and floating-point instructions

In general, ARMv8 requires implementation of Advanced SIMD and floating-point instructions in the T32 and A32 instruction sets, but see *Implications of not including Advanced SIMD and floating-point support* on page E1-2333.

The Advanced SIMD instructions perform packed *Single Instruction Multiple Data* (SIMD) operations, either integer or single-precision floating-point. The floating-point instructions perform single-precision or double-precision scalar floating-point operations.

These instructions permit *floating-point exceptions*, such as overflow or division by zero, to be handled without trapping. When handled in this way, a floating-point exception causes a cumulative status register bit to be set to 1 and a default result to be produced by the operation. ARMv8 also optionally supports the trapping of floating-point exceptions, see *Trapping of floating-point exceptions* on page E1-2329. For more information about floating-point exceptions see *Floating-point exceptions* on page E1-2329.

The floating-point and Advanced SIMD instructions also provide conversion functions in both directions between half-precision floating-point and single-precision floating-point.

Some Advanced SIMD instructions support polynomial arithmetic over {0, 1}, as described in *Polynomial arithmetic over {0, 1}* on page A1-45.

For system level information about the Advanced SIMD and Floating-point implementation see *Advanced SIMD and floating-point support* on page G1-3841.

The following sections give more information about the Advanced SIMD and floating-point instructions:

- *Floating-point standards, and terminology* on page A1-48.
- *The SIMD and floating-point register file*.
- *Data types supported by the Advanced SIMD implementation* on page E1-2328.
- *Advanced SIMD and floating-point System registers* on page E1-2328.
- *Trapping of floating-point exceptions* on page E1-2329.
- *Floating-point data types and arithmetic* on page E1-2329.
- *Floating-point exceptions* on page E1-2329.
- *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-2332.
- *Implications of not including Advanced SIMD and floating-point support* on page E1-2333.
- *Pseudocode description of floating-point operations* on page E1-2333.

### E1.3.1 The SIMD and floating-point register file

The Advanced SIMD and floating-point instructions use the same register file, that comprises 32 registers. This is distinct from the register file that holds the general-purpose registers and the PC.

The Advanced SIMD and floating-point views of the register file are different. The following sections describe these different views. Figure E1-1 on page E1-2326 shows the views of the register file, and the way the word, doubleword, and quadword registers overlap.

#### Advanced SIMD views of the register file

Advanced SIMD can view this register file as:

- Sixteen 128-bit quadword registers, `Q0-Q15`.
- Thirty-two 64-bit doubleword registers, `D0-D31`.

These views can be used simultaneously. For example, a program might hold 64-bit vectors in `D0` and `D1` and a 128-bit vector in `Q1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Floating-point views of the register file

The Advanced SIMD and floating-point register file consists of thirty-two doubleword registers, that can be viewed as:

- Thirty-two 64-bit doubleword registers, `D0-D31`. This view is also available to Advanced SIMD instructions.
- Thirty-two 32-bit single word registers, `S0-S31`. Only half of the set is accessible in this view.

The two views can be used simultaneously.

### SIMD and Floating-point register file mapping onto registers

Figure E1-1 shows the different views of the SIMD and floating-point register file, and the relationship between them.

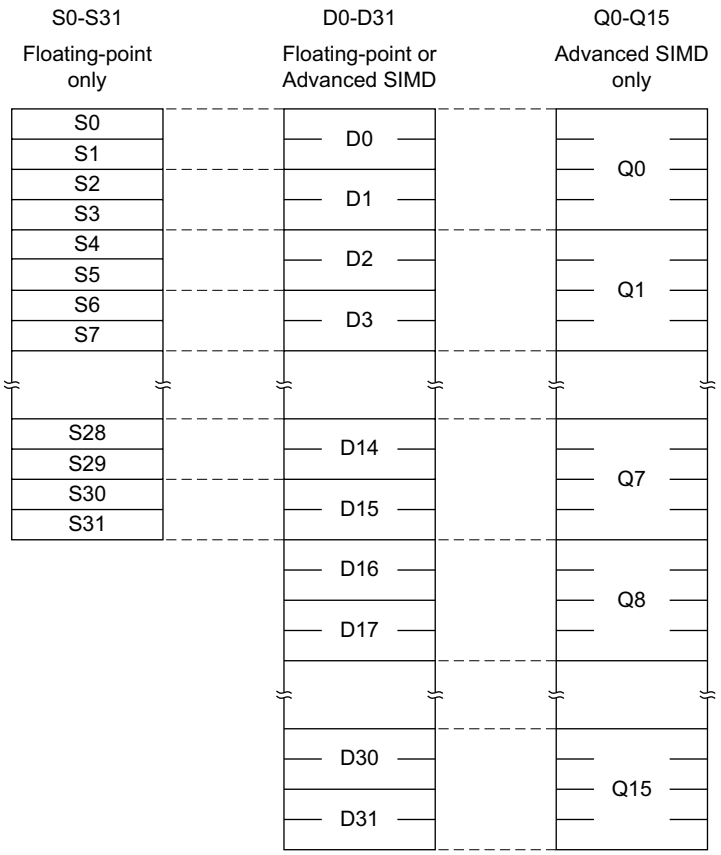



**Figure E1-1 SIMD and floating-point register file, AArch32 operation**

The mapping between the registers is as follows:

- `S<2n>` maps to the least significant half of `D<n>`.
- `S<2n+1>` maps to the most significant half of `D<n>`.
- `D<2n>` maps to the least significant half of `Q<n>`.
- `D<2n+1>` maps to the most significant half of `Q<n>`.

For example, software can access the least significant half of the elements of a vector in `Q6` by referring to `D12`, and the most significant half of the elements by referring to `D13`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Pseudocode description of the SIMD and Floating-point register file

The following functions provide the S0-S31, D0-D31, and Q0-Q15 views of the Advanced SIMD and floating-point registers:

```
array bits(64) _Dclone[0..31];

// S[] - non-assignment form
// =========================

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    return _V[n DIV 4]<base+31:base>;

// S[] - assignment form
// =====================

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    _V[n DIV 4]<base+31:base> = value;
    return;

// D[] - non-assignment form
// =========================

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    return _V[n DIV 2]<base+63:base>;

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    _V[n DIV 2]<base+63:base> = value;
    return;
```

The Din[] function returns a Doubleword register from the _Dclone[] copy of the SIMD and Floating-point register file, and the Qin[] function returns a Quadword register from that register file.

**Note**

The CheckAdvancedSIMDEnabled() function copies the D[] register file to _Dclone[], see *Pseudocode description of enabling SIMD and floating-point functionality* on page G1-3879.

```
// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];

// Qin[] - non-assignment form
// ===========================

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### E1.3.2 Data types supported by the Advanced SIMD implementation

Advanced SIMD instructions can operate on integer and floating-point data, and the implementation defines a set of data types that support the required data formats. *Vector formats in AArch32 state* on page A1-38 describes these formats.

#### Advanced SIMD vectors

In an implementation that includes support for Advanced SIMD operation, a register can hold one or more packed elements, all of the same size and type. The combination of a register and a data type describes a vector of elements. The vector is considered to be an array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the size of the register.

Vector indices are in the range 0 to (number of elements – 1). An index of 0 refers to the least significant end of the vector. In *Vector formats in AArch32 state* on page A1-38, Figure A1-3 on page A1-40 shows the Advanced SIMD vector formats.

##### *Pseudocode description of Advanced SIMD vectors*

The pseudocode function `Elem[]` accesses the element of a specified index and size in a vector:

```
// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector<e*size+size-1 : e*size>;

// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e]
    return Elem[vector, e, size];

// Elem[] - assignment form
// ========================

Elem[bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;

// Elem[] - assignment form
// ========================

Elem[bits(N) &vector, integer e] = bits(size) value
    Elem[vector, e, size] = value;
    return;
```

### E1.3.3 Advanced SIMD and floating-point System registers

The Advanced SIMD and floating-point instructions have a shared register space for System registers. Only one register in this space is accessible at the Application level, see *FPSCR, Floating-Point Status and Control Register* on page G6-4317.

Writes to the FPSCR can have side-effects on various aspects of PE operation. All of these side-effects are synchronous to the FPSCR write. This means they are guaranteed not to be visible to earlier instructions in the execution stream, and they are guaranteed to be visible to later instructions in the execution stream.

See *Advanced SIMD and floating-point System registers* on page G1-3843 for the system level view of the registers.

These registers can be described as the *SIMD and floating-point System registers*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### E1.3.4 Trapping of floating-point exceptions

It is IMPLEMENTATION DEFINED whether the floating-point implementation supports the trapping of floating-point exceptions:

- If it does, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable the exception traps.
- Otherwise, the FPSCR trap bits are RES0.

Trapped exception handling never causes the corresponding cumulative exception bit of the FPSCR to be set to 1. If this behavior is desired, the trap handler routine must use a read, modify, write sequence on the FPSCR to set the cumulative exception bit.

### E1.3.5 Floating-point data types and arithmetic

The T32 and A32 floating-point instructions support single-precision (32-bit) and double-precision (64-bit) data types and arithmetic as defined by the IEEE 754 floating-point standard. They also support the half-precision (16-bit) floating-point data type for data storage only, by supporting conversions between single-precision and half-precision data types.

*ARM standard floating-point arithmetic* means IEEE 754 floating-point arithmetic with the restrictions described in *Floating-point and Advanced SIMD support* on page A1-46, including supporting only the input and output values described in *ARM standard floating-point input and output values* on page A1-49.

The AArch32 Advanced SIMD instructions support only single-precision ARM standard floating-point arithmetic.

**Note**

The floating-point instructions require *support code* to be installed in the system if trapped floating-point exception handling is required. See *Floating-point exception traps* on page G1-3844.

The following sections describe the Advanced SIMD and floating-point formats:

- *Half-precision floating-point formats* on page A1-40.
- *Single-precision floating-point format* on page A1-42.
- *Double-precision floating-point format* on page A1-43.

The following sections describe features of Advanced SIMD and floating-point processing:

- *Flush-to-zero* on page A1-49.
- *NaN handling and the Default NaN* on page A1-50.

### E1.3.6 Floating-point exceptions

ARM Advanced SIMD and floating-point instructions record the following floating-point exceptions in the FPSCR cumulative bits, unless the floating-point exception is trapped and generates an exception:

**FPSCR.IOC** Invalid Operation. The bit is set to 1 if the result of an operation has no mathematical value or cannot be represented. Cases include, for example:

- (infinity) × 0.
- (+infinity) + (–infinity).

These tests are made after flush-to-zero processing. For example, if flush-to-zero mode is selected, multiplying a denormalized number and an infinity is treated as (0 × infinity), and causes an Invalid Operation floating-point exception.

IOC is also set on any floating-point operation with one or more signaling NaNs as operands, except for negation and absolute value, as described in *Floating-point negation and absolute value* on page E1-2334.

**FPSCR.DZC** Division by Zero. The bit is set to 1 if a divide operation has a zero divisor and a dividend that is not zero, an infinity or a NaN. These tests are made after flush-to-zero processing, so if flush-to-zero processing is selected, a denormalized dividend is treated as zero and prevents Division by Zero from occurring, and a denormalized divisor is treated as zero and causes Division by Zero to occur if the dividend is a normalized number.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the reciprocal and reciprocal square root estimate functions the dividend is assumed to be +1.0. This means that a zero or denormalized operand to these functions sets the DZC bit.

**FPSCR.OFC** Overflow. The bit is set to 1 if the absolute value of the result of an operation, produced after rounding, is greater than the maximum positive normalized number for the destination precision.

**FPSCR.UFC** Underflow. The bit is set to 1 if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, and the rounded result is inexact.

The criteria for the Underflow exception to occur are different in Flush-to-zero mode. For details, see *Flush-to-zero* on page A1-49.

**FPSCR.IXC** Inexact. The bit is set to 1 if the result of an operation is not equivalent to the value that would be produced if the operation were performed with unbounded precision and exponent range.

The criteria for the Inexact exception to occur are different in Flush-to-zero mode. For details, see *Flush-to-zero* on page A1-49.

**FPSCR.IDC** Input Denormal. The bit is set to 1 if a denormalized input operand is replaced in the computation by a zero, as described in *Flush-to-zero* on page A1-49.

For Advanced SIMD instructions, and for floating-point instructions when floating-point exception trapping is not supported, these are non-trapping exceptions and the data-processing instructions do not generate any trapped exceptions.

For floating-point instructions when floating-point exception trapping is supported:

- These exceptions can be trapped, by setting trap enable bits in the FPSCR, see *Trapping of floating-point exceptions* on page E1-2329 and *Floating-point exception traps* on page G1-3844, and:
  - When a trap is not enabled the corresponding floating-point exception updates the corresponding FPSCR cumulative bit does not generate an exception.
  - When a trap is enabled the corresponding floating-point exception does not update the FPSCR, but generates an exception. In this case, bits in the FPEXC indicate which floating-point exceptions have occurred.
- The definition of the Underflow exception is different in the trapped and cumulative exception cases. In the trapped case the definition is:
  - The trapped Underflow exception occurs if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, regardless of whether the rounded result is inexact.
- As with cumulative exceptions, higher priority trapped exceptions can prevent lower priority exceptions from occurring, as described in *Combinations of floating-point exceptions* on page E1-2331.
- For Invalid Operation exceptions, for details of which quiet NaN is produced as the default result see *NaN handling and the Default NaN* on page A1-50.
- For Overflow exceptions, the sign bit of the default result is determined normally for the overflowing operation.
- For Division by Zero exceptions, the sign bit of the default result is determined normally for a division. This means it is the exclusive OR of the sign bits of the two operands.

Table E1-3 on page E1-2331 shows the results of untrapped floating-point exceptions. That table uses the following abbreviations:

**MaxNorm** The maximum normalized number of the destination precision.

**RM** Round towards Minus Infinity mode, as defined in the IEEE 754 standard.

**RN** Round to Nearest mode, as defined in the IEEE 754 standard.

**RP** Round towards Plus Infinity mode, as defined in the IEEE 754 standard.

**RZ** Round towards Zero mode, as defined in the IEEE 754 standard.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about the IEEE 754 descriptions of the rounding modes see *Floating-point standards, and terminology* on page A1-48.

**Table E1-3 Results of untrapped floating-point exceptions**

| Exception type | Default result for positive sign | Default result for negative sign |
|---|---|---|
| IOC, Invalid Operation | Quiet NaN | Quiet NaN |
| DZC, Division by Zero | +infinity | -infinity |
| OFC, Overflow | RN, RP: +infinity<br>RM, RZ: +MaxNorm | RN, RM: -infinity<br>RP, RZ: -MaxNorm |
| UFC, Underflow | Normal rounded result | Normal rounded result |
| IXC, Inexact | Normal rounded result | Normal rounded result |
| IDC, Input Denormal | Normal rounded result | Normal rounded result |

## Combinations of floating-point exceptions

Many pseudocode functions perform *floating-point operations*, including `FixedToFP()`, `FPAdd()`, `FPCompare()`, `FPCompareEQ()`, `FPCompareGE()`, `FPCompareGT()`, `FPDiv()`, `FPMax()`, `FPMin()`, `FPMul()`, `FPMulAdd()`, `FPRecipEstimate()`, `FPRecipStep()`, `FPRSqrtEstimate()`, `FPRSqrtStep()`, `FPSqrt()`, `FPSub()`, and `FPToFixed()`. All of these operations can generate floating-point exceptions.

**Note**

`FPAbs()` and `FPNeg()` are not classified as *floating-point operations* because:

- They cannot generate floating-point exceptions.
- The floating-point operation behavior described in the following sections does not apply to them:
  - *Flush-to-zero* on page A1-49.
  - *NaN handling and the Default NaN* on page A1-50.

More than one exception can occur on the same operation. The only combinations of exceptions that can occur are:

- Overflow with Inexact.
- Underflow with Inexact.
- Input Denormal with other exceptions.

When none of the exceptions caused by an operation is trapped, any exception that occurs causes the associated cumulative bit in the FPSCR to be set.

When one or more exceptions caused by an operation are trapped, the behavior of the instruction depends on the priority of the exceptions. The Inexact exception is treated as lowest priority, and Input Denormal as highest priority:

- If the higher priority exception is trapped, its trap handler is called. It is IMPLEMENTATION DEFINED whether any information about the lower priority exception is provided.

  **Note**

  Information about the lower priority exception might be provided in:

  — The FPEXC, if the exception generated by the trap is taken to an Exception level that is using AArch32.

  — The ESR_ELx.ISS field, if the exception generated by the trap is taken to an Exception level that is using AArch64.

  However, information might be provided in another IMPLEMENTATION DEFINED way, for example using an IMPLEMENTATION DEFINED register.

  Apart from this, the lower priority exception is ignored in this case.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If the higher priority exception is untrapped, its cumulative bit is set to 1 and its default result is evaluated. Then the lower priority exception is handled normally, using this default result.

Some floating-point instructions specify more than one floating-point operation, as indicated by the pseudocode descriptions of the instruction. In such cases, an exception on one operation is treated as higher priority than an exception on another operation if the occurrence of the second exception depends on the result of the first operation. Otherwise, it is UNPREDICTABLE which exception is treated as higher priority.

For example, a `VMLA.F32` instruction specifies a floating-point multiplication followed by a floating-point addition. The addition can generate Overflow, Underflow and Inexact exceptions, all of which depend on both operands to the addition and so are treated as lower priority than any exception on the multiplication. The same applies to Invalid Operation exceptions on the addition caused by adding opposite-signed infinities. The addition can also generate an Input Denormal exception, caused by the addend being a denormalized number while in Flush-to-zero mode. It is UNPREDICTABLE which of an Input Denormal exception on the addition and an exception on the multiplication is treated as higher priority, because the occurrence of the Input Denormal exception does not depend on the result of the multiplication. The same applies to an Invalid Operation exception on the addition caused by the addend being a signaling NaN.

**Note**

- The `VFMA` instruction performs a vector addition and a vector multiplication as a single operation. The `VFMS` instruction performs a vector subtraction and a vector multiplication as a single operation.
- Like other details of Floating-point instruction execution, these rules about exception handling apply to the overall results produced by an instruction when the system uses a combination of hardware and support code to implement it. See *Floating-point exception traps* on page G1-3844 for more information.

### E1.3.7 Controls of Advanced SIMD operation that do not apply to floating-point operation

ARMv7 permitted implementation of either, both, or neither of the Advanced SIMD and floating-point additions to the base instruction set, and provided some controls that applied to the Advanced SIMD functionality but not to the floating-point functionality. In ARMv8, Advanced SIMD functionality cannot be separated from floating-point functionality, but in AArch32 state these controls function as they did in ARMv7. This means they apply only to the following instructions and instruction encodings:

- All instructions with encodings defined in *Advanced SIMD data-processing instructions* on page F5-2553.
- All instructions with encodings defined in *Advanced SIMD element or structure load/store instructions* on page F5-2569.
- The form of the `VDUP` instruction described in *VDUP (general-purpose register)* on page F7-3370.
- The byte and halfword forms of the VMOV instructions described in each of:
  - *VMOV (general-purpose register to scalar)* on page F7-3483.
  - *VMOV (scalar to general-purpose register)* on page F7-3487.

The controls of this functionality are:

- The CPACR.ASEDIS field.
- The HCPTR.TASE field.

In an implementation that supports Advanced SIMD functionality, support for each of these controls is optional:

- If the CPACR.ASEDIS control is not supported then the CPACR.ASEDIS field is RAZ/WI. This is equivalent to the control permitting the execution of Advanced SIMD instructions at EL1 and EL0.
- If the HCPTR.TASE control is not supported then the HCPTR.TASE field is RAZ/WI. This means the HCPTR does not provide a control that can trap Non-secure execution of Advanced SIMD instructions to Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### E1.3.8 Implications of not including Advanced SIMD and floating-point support

In general, ARMv8 requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets. Exceptionally, for implementation targeting specialized markets, ARM might produce or license an ARMv8-A implementation that does not provide any support for Advanced SIMD and floating-point instructions. In such an implementation, in AArch32 state:

- Each of the CPACR.{cp10, cp11} fields is RES0.
- The CPACR.ASEDIS bit is RES0.
- Each of the HCPTR.{TASE, TCP10, TCP11} fields is RES1.
- Each of the NSACR.{NSASEDIS, cp10, cp11} fields is RES0.
- The FPEXC register is UNDEFINED.

### E1.3.9 Pseudocode description of floating-point operations

The following subsections contain pseudocode definitions of the floating-point functionality supported by the ARMv8 architecture:

- *Generation of specific floating-point values*.
- *Floating-point negation and absolute value* on page E1-2334.
- *Floating-point value unpacking* on page E1-2335.
- *Floating-point exception and NaN handling* on page E1-2336.
- *Floating-point rounding* on page E1-2338.
- *Selection of ARM standard floating-point arithmetic* on page E1-2339.
- *Floating-point comparisons* on page E1-2340.
- *Floating-point maximum and minimum* on page E1-2341.
- *Floating-point addition and subtraction* on page E1-2342.
- *Floating-point multiplication and division* on page E1-2343.
- *Floating-point fused multiply-add* on page E1-2343.
- *Floating-point reciprocal estimate and step* on page E1-2344.
- *Floating-point square root* on page E1-2347.
- *Floating-point reciprocal square root estimate and step* on page E1-2347.
- *Floating-point conversions* on page E1-2350.

#### Generation of specific floating-point values

The following pseudocode functions generate specific floating-point values. The `sign` argument is `'0'` for the positive version and `'1'` for the negative version.

```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Ones(E);
    frac = Zeros(F);
    return sign : exp : frac;

// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// FPZero()
// ========

bits(N) FPZero(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Zeros(E);
    frac = Zeros(F);
    return sign : exp : frac;

// FPTwo()
// =======

bits(N) FPTwo(bit sign)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    return sign : exp : frac;

// FPThree()
// =========

bits(N) FPThree(bit sign)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;

// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN()
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = '0';
    exp  = Ones(E);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

### Floating-point negation and absolute value

The floating-point negation and absolute value operations only affect the sign bit. They do not treat NaN operands specially, nor denormalized number operands when flush-to-zero is selected.

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)
    assert N IN {32,64};
    return NOT(op<N-1>) : op<N-2:0>;

// FPAbs()
// =======

bits(N) FPAbs(bits(N) op)
    assert N IN {32,64};
    return '0' : op<N-2:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Floating-point value unpacking

The FPUnpack() function determines the type and numerical value of a floating-point number. It also does flush-to-zero processing on input operands.

```
enumeration FPType      {FPType_Nonzero, FPType_Zero, FPType_Infinity,
                         FPType_QNaN, FPType_SNaN};

// FPUnpack()
// ==========
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)
    assert N IN {16,32,64};

    if N == 16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            // Produce zero if value is zero
            if IsZero(frac16) then
                type = FPType_Zero;  value = 0.0;
            else
                type = FPType_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 then

        sign   = fpval<31>;
        exp32  = fpval<30:23>;
        frac32 = fpval<22:0>;
        if IsZero(exp32) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac32) || fpcr.FZ == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac32) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpcr);
            else
                type = FPType_Nonzero;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        sign   = fpval<63>;
        exp64  = fpval<62:52>;
        frac64 = fpval<51:0>;
        if IsZero(exp64) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac64) || fpcr.FZ == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac64) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpcr);
            else
                type = FPType_Nonzero;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;
    return (type, sign, value);
```

### Floating-point exception and NaN handling

The FPProcessException() procedure checks whether a floating-point exception is trapped, and handles it accordingly:

```
enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                         FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};

// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)
    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp     cumul = 0;
        when FPExc_DivideByZero  cumul = 1;
        when FPExc_Overflow      cumul = 2;
        when FPExc_Underflow     cumul = 3;
        when FPExc_Inexact       cumul = 4;
        when FPExc_InputDenorm   cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrapException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

The FPProcessNaN() function processes a NaN operand, producing the correct result value and generating an Invalid Operation exception if necessary:

```
// FPProcessNaN()
// ==============
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(N) FPProcessNaN(FPType type, bits(N) op, FPCRType fpcr)
    assert N IN {32,64};
    assert type IN {FPType_QNaN, FPType_SNaN};

    topfrac = if N == 32 then 22 else 51;
    result = op;
    if type == FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN();
    return result;
```

The `FPProcessNaNs()` function performs the standard NaN processing for a two-operand operation:

```
// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                 bits(N) op1, bits(N) op2,
                                 FPCRType fpcr)
    assert N IN {32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

The `FPProcessNaNs3()` function performs the standard NaN processing for a three-operand operation:

```
// FPProcessNaNs3()
// ================
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRType fpcr)
    assert N IN {32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_QNaN then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

### Floating-point rounding

The FPRound() function rounds and encodes a floating-point result value to a specified destination format. This includes processing Overflow, Underflow and Inexact floating-point exceptions and performing flush-to-zero processing on result values.

```
// FPRound()
// =========

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Deal with flush-to-zero.
    if fpcr.FZ == '1' && N != 16 && exponent < minimum_exp then
        // Flush-to-zero never generates a trapped exception
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            FPSR.UFC = '1';
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped.
    if biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
        FPProcessException(FPExc_Underflow, fpcr);

    // Round result according to rounding mode.
    case rounding of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then      // Rounded up from denormalized to normalized
            biased_exp = 1;
        if int_mant == 2^(F+1) then  // Rounded up to next exponent
            biased_exp = biased_exp + 1;  int_mant = int_mant DIV 2;

    // Handle rounding to odd aka Von Neumann rounding
    if error != 0.0 && rounding == FPRounding_ODD then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
            FPProcessException(FPExc_Overflow, fpcr);
            error = 1.0;  // Ensure that an Inexact exception occurs
        else
            result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;
    else                                    // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0;  // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;

    // Deal with Inexact exception.
    if error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
```

## Selection of ARM standard floating-point arithmetic

The `StandardFPSCRValue()` function returns the FPSCR value that selects ARM standard floating-point arithmetic. Most of the arithmetic functions have a Boolean `fpscr_controlled` argument that is `TRUE` for Floating-point operations and `FALSE` for Advanced SIMD operations, and that selects between using the real FPSCR value and this value.

```
// StandardFPSCRValue()
// ====================

FPCRType StandardFPSCRValue()
    return '00000' : FPSCR.AHP : '1100000000000000000000000';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Floating-point comparisons

The FPCompare() function compares two floating-point numbers, producing a {N, Z, C, V} Condition flags result as shown in Table E1-4:

**Table E1-4 Effect of a Floating-point comparison on the Condition flags**

| Comparison result | N | Z | C | V |
|---|---|---|---|---|
| Equal | 0 | 1 | 1 | 0 |
| Less than | 1 | 0 | 0 | 0 |
| Greater than | 0 | 0 | 1 | 0 |
| Unordered | 0 | 0 | 1 | 1 |

This result defines the operation of the VCMP floating-point instruction. The VCMP instruction writes these flag values in the FPSCR. After using a VMRS instruction to transfer them to the APSR, they can control conditional execution as shown in Table F2-1 on page F2-2447.

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = '0011';
        if type1==FPType_SNaN || type2==FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';
    return result;
```

The FPCompareEQ(), FPCompareGE() and FPCompareGT() functions describe the operation of Advanced SIMD instructions that perform floating-point comparisons.

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        if type1==FPType_SNaN || type2==FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;

// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
    return result;

// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
    return result;
```

### Floating-point maximum and minimum

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 > value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType_Infinity then
            result = FPInfinity(sign);
        elsif type == FPType_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign);
        else
            result = FPRound(value, fpcr);
    return result;

// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 < value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType_Infinity then
            result = FPInfinity(sign);
        elsif type == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result = FPZero(sign);
        else
            result = FPRound(value, fpcr);
    return result;
```

### Floating-point addition and subtraction

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;

// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return result;
```

### Floating-point multiplication and division

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;

// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);
    return result;
```

### Floating-point fused multiply-add

```
// FPMulAdd()
// ==========
//
// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);

    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity);  zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);

    return result;
```

### Floating-point reciprocal estimate and step

The Advanced SIMD implementation includes instructions that support Newton-Raphson calculation of the reciprocal of a number.

The `VRECPE` instruction produces the initial estimate of the reciprocal. It uses the following pseudocode functions:

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {32, 64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPType_Infinity then
        result = FPZero(sign);
    elsif type == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (N == 32 && Abs(value) < 2.0^-128)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
          || (N == 64 && Abs(value) < 2.0^-1024) then
        case FPRoundingMode(fpcr) of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    elsif fpcr.FZ == '1'
          && ((N == 32 && Abs(value) >= 2.0^126)
              || (N == 64 && Abs(value) >= 2.0^1022)) then
        // Result flushed to zero of correct sign
        result = FPZero(sign);
        FPProcessException(FPExc_Underflow, fpcr);
    else
        // Scale to a double-precision value in the range 0.5 <= x < 1.0, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction extended with zeros.

        if N == 32 then
            fraction = operand<22:0> : Zeros(29);
            exp = UInt(operand<30:23>);
        else // N == 64
            fraction = operand<51:0>;
            exp = UInt(operand<62:52>);

        if exp == 0 then
            if fraction<51> == 0 then
                exp = -1;
                fraction = fraction<49:0>:'00';
            else
                fraction = fraction<50:0>:'0';
        scaled = '0' : '01111111110' : fraction<51:44> : Zeros(44);

        if N == 32 then
            result_exp = 253 - exp;     // In range 253-254 = -1 to 253+1 = 254
        else // N == 64
            result_exp = 2045 - exp;    // In range 2045-2046 = -1 to 2045+1 = 2046

        // Call C function to get reciprocal estimate of scaled value.
        // Input is rounded down to a multiple of 1/512.
        estimate = recip_estimate(scaled);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Convert to scaled single-precision result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.

        fraction = estimate<51:0>;
        if result_exp == 0 then
            fraction = '1' : fraction<51:1>;
        elsif result_exp == -1 then
            fraction = '01' : fraction<51:2>;
            result_exp = 0;
        if N == 32 then
            result = sign : result_exp<N-25:0> : fraction<51:29>;
        else // N == 64
            result = sign : result_exp<N-54:0> : fraction<51:0>;

    return result;

// UnsignedRecipEstimate()
// =======================
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(32) UnsignedRecipEstimate(bits(32) operand)

    if operand<31> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(32);
    else
        // Generate double-precision value = operand * 2^(-32). This has zero sign bit, with:
        //     exponent = 1022 = double-precision representation of 2^(-1)
        //     fraction taken from operand, excluding its most significant bit.
        dp_operand = '0 01111111110' : operand<30:0> : Zeros(21);

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_estimate(dp_operand);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Multiply by 2^31 and convert to an unsigned integer - this just involves
        // concatenating the implicit units bit with the top 31 fraction bits.
        result = '1' : estimate<51:21>;

    return result;
```

`recip_estimate()` is defined by the following C function:

```
double recip_estimate(double a)
{
  int q, s;
  double r;
  q = (int)(a * 512.0);         /* a in units of 1/512 rounded down */
  r = 1.0 / (((double)q + 0.5) / 512.0);  /* reciprocal r */
  s = (int)(256.0 * r + 0.5);   /* r in units of 1/256 rounded to nearest */
  return (double)s / 256.0;
}
```

Table E1-5 shows the results where input values are out of range.

**Table E1-5 VRECPE results for out of range inputs**

| Number type | Input Vm[i] | Result Vd[i] |
|---|---|---|
| Integer | <= 0x7FFFFFFF | 0xFFFFFFFF |
| Floating-point | NaN | Default NaN |
| Floating-point | ±0 or denormalized number | ±infinity [a] |
| Floating-point | ±infinity | ±0 |
| Floating-point | Absolute value >= $2^{126}$ | ±0 |

a. FPSCR.DZC is set to 1

The Newton-Raphson iteration:

$x_{n+1} = x_n(2-dx_n)$

converges to $(1/d)$ if $x_0$ is the result of VRECPE applied to $d$.

The VRECPS instruction performs a (2 - op1×op2) calculation and can be used with a multiplication to perform a step of this iteration. The functionality of this instruction is defined by the following pseudocode function:

```
// FPRecipStep()
// =============

bits(32) FPRecipStep(bits(32) op1, bits(32) op2)
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(32) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        result = FPSub(FPTwo('0'), product, fpcr);
    return result;
```

Table E1-6 shows the results where input values are out of range.

**Table E1-6 VRECPS results for out of range inputs**

| Input Vn[i] | Input Vm[i] | Result Vd[i] |
|---|---|---|
| Any NaN | - | Default NaN |
| - | Any NaN | Default NaN |
| ±0.0 or denormalized number | ±infinity | 2.0 |
| ±infinity | ±0.0 or denormalized number | 2.0 |

### Floating-point square root

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
    assert N IN {32,64};
    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    elsif type == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr);
    return result;
```

### Floating-point reciprocal square root estimate and step

The Advanced SIMD implementation includes instructions that support Newton-Raphson calculation of the reciprocal of the square root of a number.

The `VRSQRTE` instruction produces the initial estimate of the reciprocal of the square root. It uses the following pseudocode functions:

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {32, 64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPType_Zero then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result = FPInfinity(sign);
            FPProcessException(FPExc_DivideByZero, fpcr);
        elsif sign == '1' then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif type == FPType_Infinity then
            result = FPZero('0');
        else
            // Scale to a double-precision value in the range 0.25 <= x < 1.0, with the
            // evenness or oddness of the exponent unchanged, and calculate result exponent.
            // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
            // biased version of -1 or -2, fraction = original fraction extended with zeros.

            if N == 32 then
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            else // N == 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

            if exp == 0 then
                while fraction<51> == 0 do
                    fraction = fraction<50:0> : '0';
                    exp = exp - 1;
                fraction = fraction<50:0> : '0';

            if exp<0> == '0' then
                scaled = '0' : '01111111110' : fraction<51:44> : Zeros(44);
            else
                scaled = '0' : '01111111101' : fraction<51:44> : Zeros(44);

            if N == 32 then
                result_exp = (380 - exp) DIV 2;
            else // N == 64
                result_exp = (3068 - exp) DIV 2;

            // Call C function to get reciprocal estimate of scaled value.
            estimate = recip_sqrt_estimate(scaled);

            // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
            // Convert to scaled single-precision result with copied sign bit and high-order
            // fraction bits, and exponent calculated above.

            if N == 32 then
                result = '0' : result_exp<N-25:0> : estimate<51:29>;
            else // N == 64
                result = '0' : result_exp<N-54:0> : estimate<51:0>;
        return result;

    // UnsignedRSqrtEstimate()
    // =======================

    bits(32) UnsignedRSqrtEstimate(bits(32) operand)

        if operand<31:30> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
            result = Ones(32);
        else
            // Generate double-precision value = operand * 2^(-32). This has zero sign bit, with:
            //     exponent = 1022 or 1021 = double-precision representation of 2^(-1) or 2^(-2)
            //     fraction taken from operand, excluding its most significant one or two bits.
            if operand<31> == '1' then
                dp_operand = '0 01111111110' : operand<30:0> : Zeros(21);
            else  // operand<31:30> == '01'
                dp_operand = '0 01111111101' : operand<29:0> : Zeros(22);

            // Call C function to get reciprocal estimate of scaled value.
            estimate = recip_sqrt_estimate(dp_operand);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Multiply by 2^31 and convert to an unsigned integer - this just involves
        // concatenating the implicit units bit with the top 31 fraction bits.
        result = '1' : estimate<51:21>;

    return result;
```

`recip_sqrt_estimate()` is defined by the following C function:

```
double recip_sqrt_estimate(double a)
{
  int q0, q1, s;
  double r;
  if (a < 0.5)  /* range 0.25 <= a < 0.5 */
  {
    q0 = (int)(a * 512.0);       /* a in units of 1/512 rounded down */
    r = 1.0 / sqrt(((double)q0 + 0.5) / 512.0); /* reciprocal root r */
  }
  else          /* range 0.5  <= a < 1.0 */
  {
    q1 = (int)(a * 256.0);       /* a in units of 1/256 rounded down */
    r = 1.0 / sqrt(((double)q1 + 0.5) / 256.0); /* reciprocal root r */
  }
  s = (int)(256.0 * r + 0.5);  /* r in units of 1/256 rounded to nearest */
  return (double)s / 256.0;
}
```

Table E1-7 shows the results where input values are out of range.

**Table E1-7 VRSQRTE results for out of range inputs**

| Number type | Input Vm[i] | Result Vd[i] |
|---|---|---|
| Integer | <= 0x3FFFFFFF | 0xFFFFFFFF |
| Floating-point | NaN, –(normalized number), –infinity | Default NaN |
| Floating-point | –0 or –(denormalized number) | – infinity [a] |
| Floating-point | +0 or +(denormalized number) | +infinity [a] |
| Floating-point | +infinity | +0 |

a. FPSCR.DZC is set to 1.

The Newton-Raphson iteration:

$x_{n+1} = x_n(3-dx_n^2)/2$

converges to $(1/\sqrt{d})$ if $x_0$ is the result of VRSQRTE applied to $d$.

The VRSQRTS instruction performs a (3 – op1×op2)/2 calculation and can be used with two multiplications to perform a step of this iteration. The `FPRSqrtStep()` pseudocode function defines the functionality of this instruction:

```
// FPRSqrtStep()
// =============

bits(32) FPRSqrtStep(bits(32) op1, bits(32) op2)
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(32) product;
        if (inf1 && zero2) || (zero1 && inf2) then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        result = FPHalvedSub(FPThree('0'), product, fpcr);
    return result;
```

Table E1-8 shows the results where input values are out of range.

**Table E1-8 VRSQRTS results for out of range inputs**

| Input Vn[i] | Input Vm[i] | Result Vd[i] |
|---|---|---|
| Any NaN | - | Default NaN |
| - | Any NaN | Default NaN |
| ±0.0 or denormalized number | ±infinity | 1.5 |
| ±infinity | ±0.0 or denormalized number | 1.5 |

`FPRSqrtStep()` calls the `FPHalvedSub()` pseudocode function:

```
// FPHalvedSub()
// =============

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

## Floating-point conversions

The following function performs conversions between half-precision, single-precision and double-precision floating-point numbers.

```
// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding)
    assert M IN {16,32,64};
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (type,sign,value) = FPUnpack(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if type == FPType_SNaN || type == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        if type == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif type == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRound(value, fpcr, rounding);

    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType fpcr)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

The following functions perform conversions between floating-point numbers and integers or fixed-point numbers:

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
    assert N IN {32,64};
    assert M IN {32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    // If NaN, set cumulative flag or take exception
    if type == FPType_SNaN || type == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Generate saturated result and exceptions
        (result, overflow) = SatQ(int_result, M, unsigned);
        if overflow then
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif error != 0.0 then
            FPProcessException(FPExc_Inexact, fpcr);

        return result;

    // FixedToFP()
    // ===========

    // Convert M-bit fixed point OP with FBITS fractional bits to
    // N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

    bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
        assert N IN {32,64};
        assert M IN {32,64};
        bits(N) result;
        assert fbits >= 0;
        assert rounding != FPRounding_ODD;

        // Correct signed-ness
        int_operand = Int(op, unsigned);

        // Scale by fractional bits and generate a real value
        real_operand = Real(int_operand) / 2.0^fbits;

        if real_operand == 0.0 then
            result = FPZero('0');
        else
            result = FPRound(real_operand, fpcr, rounding);

        return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E1.4 Conceptual coprocessor support

AArch32 state provides a coprocessor interface, that comprises the coprocessor instructions summarized in *Coprocessor instructions* on page F1-2427. These can provide access to sixteen conceptual coprocessors, described as CP0 to CP15. The following coprocessors are reserved by ARM for specific purposes, and from ARMv8 are the only supported coprocessors:

- Coprocessor 15 (CP15) provides system control functionality, by providing access to System registers. This includes architecture and feature identification, as well as control, status information and configuration support.

  The following sections give a general description of CP15:
  - *About the System registers for VMSAv8-32* on page G4-4116.
  - *Organization of the CP15 registers in VMSAv8-32* on page G4-4141.
  - *Functional grouping of VMSAv8-32 System registers* on page G4-4161.

  CP15 also provides performance monitor registers, see Chapter D5 *The Performance Monitors Extension*.

- Coprocessor 14 (CP14) provides access to additional registers, that support:
  - Debug, see Chapter G2 *AArch32 Self-hosted Debug*.
  - The Jazelle identification registers, see *Jazelle support* on page E1-2324.

- Coprocessors 10 and 11 (CP10 and CP11) together support floating-point and Advanced SIMD vector operations, and the control and configuration of Advanced SIMD and floating-point operation.

**Note**

To enable use of the Advanced SIMD and floating-point instructions, software must enable access to both CP10 and CP11, see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

*UNPREDICTABLE and UNDEFINED behavior for AArch32 System register accesses* on page G4-4119 gives information more information about permitted accesses to coprocessors CP14 and CP15.

Most CP14 and CP15 functions cannot be accessed by software executing at EL0. This manual clearly identifies those functions that can be accessed at EL0. However, software executing at EL1 can enable the unprivileged execution of all load, store, branch and data operation instructions associated with floating-point, Advanced SIMD and execution environment support.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E1.5 Exceptions

The ARM architecture uses the following terms to describe various types of exceptional condition:

**Exceptions** In the ARM architecture, an *exception* causes entry to EL1, EL2, or EL3. If the Exception level that is entered is using AArch32, it also causes entry to the PE mode in which the exception must be taken. A software handler for the exception is then executed.

**Note**

The term *floating-point exception* does not use this meaning of *exception*. This term is described later in this list.

Exceptions include:

- Reset.
- Interrupts.
- Memory system aborts.
- Undefined instructions.
- Supervisor calls (SVCs), Secure Monitor calls (SMCs), and hypervisor calls (HVCs).
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in *Handling exceptions that are taken to an Exception level using AArch32* on page G1-3774. In an ARMv8 implementation that includes all the Exception levels, aspects that are visible to application level software are:

- The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a call to the operating system, or other system component that is accessible only at EL1.
- The `SMC` instruction causes a Secure Monitor Call exception, but only if software execution is at EL1 or higher. Unprivileged software can only cause a Secure Monitor Call exception by methods defined by the operating system, or by another component of the software system that executes at EL1 or higher.
- The `HVC` instruction causes a Hypervisor Call exception, but only if software execution is at EL1 or higher. Unprivileged software can only cause a Hypervisor Call exception by methods defined by the hypervisor, or by another component of the software system that executes at EL1 or higher.
- The `BKPT` instruction causes a Software Breakpoint Instruction exception, that is taken as a Prefetch Abort exception. This provides a mechanism for a debugger to insert breakpoints into unprivileged software, or for unprivileged software to make a call into a debugger that is accessible at EL1.
- The `WFI` (Wait for Interrupt) instruction provides a hint that nothing needs to be done until an interrupt or another WFI wake-up event occurs, see *Wait For Interrupt* on page G1-3837. This means the hardware might enter a low-power state until the wake-up event occurs.
- The `WFE` (Wait for Event) instruction provides a hint that nothing needs to be done until either an `SEV` instruction generates an event, or another WFE wake-up event occurs, see *Wait For Event and Send Event* on page G1-3834. This means the hardware might enter a low-power state until the wake-up event occurs.

**Floating-point exceptions**

These relate to exceptional conditions encountered during floating-point arithmetic, such as division by zero or overflow. For more information see:

- *Floating-point exceptions* on page E1-2329.
- *FPEXC, Floating-Point Exception Control register* on page G6-4312.
- *FPSCR, Floating-Point Status and Control Register* on page G6-4317.
- ANSI/IEEE Std. 754, *IEEE Standard for Binary Floating-Point Arithmetic*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter E2
# The AArch32 Application Level Memory Model

This chapter gives an application level description of the memory model for software executing in AArch32 state. This means it describes the memory model for execution in EL0 when EL0 is using AArch32 in the following sections:


- *Address space* on page E2-2356.
- *Memory type overview* on page E2-2357.
- *Caches and memory hierarchy* on page E2-2358.
- *Alignment support* on page E2-2363.
- *Endian support* on page E2-2365.
- *Atomicity in the ARM architecture* on page E2-2368.
- *Memory ordering* on page E2-2372.
- *Memory types and attributes* on page E2-2382.
- *Mismatched memory attributes* on page E2-2392.
- *Synchronization and semaphores* on page E2-2395


**Note**

In this chapter, System register names usually link to the description of the register in Chapter G6 *AArch32 System Register Descriptions*, for example SCTLR.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## E2.1 Address space

Address calculations are performed using 32-bit registers. Supervisory software determines the valid address range.

Attempting to access an address that is not valid generates an MMU fault.

Address calculations are performed modulo $2^{32}$.

The result of an address calculation is UNKNOWN if it overflows or underflows the 32-bit address range A[31:0].

Memory accesses use the `MemA[]`, `MemO[]`, `MemU[]`, and `MemU_unpriv[]` pseudocode functions:

- The `MemA[]` function makes an aligned access of the required type.
- The `MemO[]` function makes an ordered access of the required type.
- The `MemU[]` function makes an unaligned access of the required type
- The `MemU_unpriv[]` function makes an unaligned, unprivileged access of the required type.

Each of these functions calls `Mem_with_type[]` function, that specifies the required access. This calls `AArch32.MemSingle[]`, which performs an atomic, little-endian read of `size` bytes.

The `AccType` enumeration defines the different access types.

**Note**

- Chapter G3 *The AArch32 System Level Memory Model* and Chapter G4 *The AArch32 Virtual Memory System Architecture* include descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also reference pseudocode descriptions of these operations.
- For references to the pseudocode that relates to memory accesses, see *Basic memory access* on page G3-3980, *Unaligned memory access* on page G3-3981, and *Aligned memory access* on page G3-3980.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E2.2 Memory type overview

ARMv8 provides the following mutually-exclusive memory types:

**Normal** This is generally used for bulk memory operations, both read-write and read-only operations.

**Device** The ARM architecture forbids speculative reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive locations.

Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.

Device memory has additional attributes that have the following effects:

- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See *Gathering* on page E2-2389.
- They preserve the access order and synchronization requirements, both for accesses to a single peripheral and where there is a synchronization requirement on the observability of one or more memory write and read accesses. See *Reordering* on page E2-2390
- They indicate whether a write can be acknowledged other than at the end point. See *Early Write Acknowledgement* on page E2-2390.

- For more information on Normal memory and Device memory, see *Memory types and attributes* on page E2-2382.

**Note**

Earlier versions of the ARM architecture defined a single Device memory type and a Strongly-Ordered memory type. A Note in *Device memory* on page E2-2387 describes how these memory types map onto the ARMv8 memory types.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E2.3 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. ARMv8 defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- *Introduction to caches*.
- *Memory hierarchy*.
- *Implication of caches for the application programmer* on page E2-2360.
- *Preloading caches* on page E2-2361.

### E2.3.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a *tag*.
- The associated data.

Caches increase the average speed of a memory access and take account of two principles of locality:

**Spatial locality**

An access to one location is likely to be followed by accesses to adjacent locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

**Temporal locality**

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a *cache line*. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a *cache hit*, and other accesses are called *cache misses*.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. ARMv8 permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

### E2.3.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, an ARMv8 memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure E2-1 on page E2-2359 shows an example of such a system in an ARMv8-A system that supports virtual addressing.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

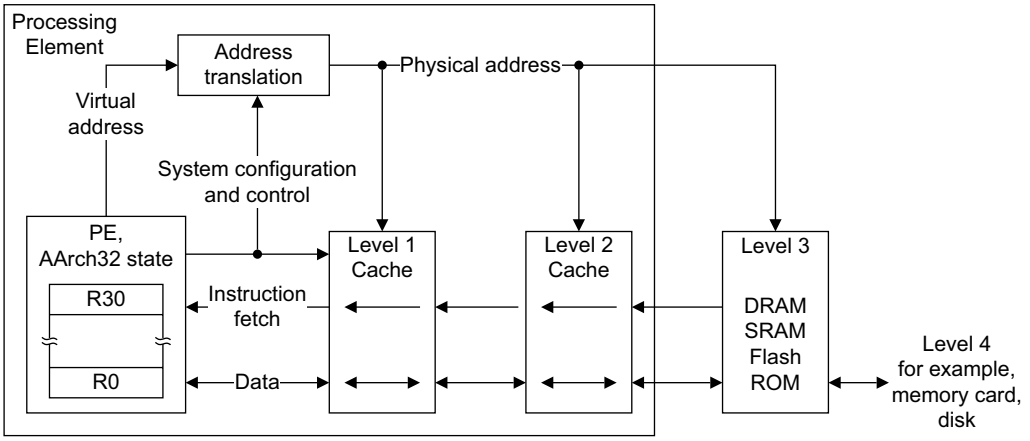



**Figure E2-1 Multiple levels of cache in a memory hierarchy**

**Note**

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the PE, as shown in Figure E2-1.

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches located at the levels closest to the main memory. Memory coherency for cache topologies can be defined by two conceptual points:

**Point of Unification (PoU)**

The point at which the instruction cache, data cache, and translation table walks of a particular PE are guaranteed to see the same copy of a memory location. In many cases, the point of unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged. The point of unification might coincide with the point of coherency.

**Point of Coherency (PoC)**

The point at which all agents that can access memory are guaranteed to see the same copy of a memory location. In many cases this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherency between memory system agents.

**Note**

The presence of system caches can affect the definition of the point of coherency as described in *System level caches* on page D3-1726.

See also *The ARMv8 cache maintenance functionality* on page G3-3957.

## The Cacheability and Shareability memory attributes

Cacheability and Shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

**Cacheability** This term defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.

**Shareability** This term defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about the Cacheability and Shareability attributes see *Memory types and attributes* on page E2-2382.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### E2.3.3 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

For example:

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.
- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

#### Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
  - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  - Not enabling caches in the system.
- By using system calls to functions using cache maintenance instructions that execute at a higher Exception level.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see *Non-shareable Normal memory* on page E2-2384 and *Shareable, Inner Shareable, and Outer Shareable Normal memory* on page E2-2383.

  **Note**

  The performance of these hardware coherency mechanisms is highly implementation-specific. In some implementations the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

#### Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last *Context synchronization operation* on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being re-fetched from memory.

The ARM architecture does not require the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory.

If software requires coherency between instruction execution and memory, it must manage this coherency using the `ISB` and `DSB` memory barriers and cache maintenance instructions. These can only be accessed from an Exception level that is higher than EL0, and therefore require a system call, see *Exception-generating and exception-handling instructions* on page F1-2426. The following code sequence can be used for this purpose:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line
; instead of STR for a 16-bit instruction.
       STR Rt, [Rn]
       DCCMVAU Rn           ; Clean data cache by MVA to point of unification (PoU)
       DSB                  ; Ensure visibility of the data cleaned from cache
       ICIMVAU Rn           ; Invalidate instruction cache by MVA to PoU
       BPIMVA  Rn           ; Invalidate branch predictor by MVA to PoU
       DSB                  ; Ensure completion of the invalidations
       ISB                  ; Synchronize the fetched instruction stream
```

**Note**

- For accesses that are Non-cacheable or Write-Through, the clean data cache instruction is not required. For accesses that are Non-cacheable, the invalidate instruction cache is not required, because in AArch32 state these accesses are not permitted to be held in an instruction cache.
- This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The ARMv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See *Concurrent modification and execution of instructions* on page E2-2370.

### E2.3.4 Preloading caches

The ARM architecture provides the memory system hints PLD (Preload Data), PLDW (Preload Data With Intent To Write) and PLI (Preload Instruction) that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring data or instruction locations into caches.

The Preload instructions are hints, and so implementations can treat them as NOPs without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous external abort, which is taken using an asynchronous Data Abort exception. For more information, see *Data Abort exception* on page G1-3821.

A PLD, PLDW, or PLI instruction can only cause allocation to software-visible caching structures such caches or TLBs for memory locations that can be accessed, according to the permissions defined by the current translation regime or a translation regime for a higher Exception level in the current Security state, by any of:

- Reads.
- Writes.
- Instruction fetches.

A PLD, PLDW, or PLI instruction can access any memory location in Normal memory that can be accessed, according to the permissions defined by the current translation regime or a translation regime for a higher Exception level in the current Security state, by any of:

- Reads.
- Writes.
- Instruction fetches.

**Note**

In each case, the entire list applies to each of PLD, PLDW, and PLI.

A PLD, PLDW, or PLI instruction is guaranteed not to access any type of Device memory.

A PLI instruction must not perform any access that cannot be performed by a speculative instruction fetch by the processor. Therefore in a VMSA implementation, if all associated MMUs are disabled, a PLI instruction cannot access any memory location that cannot be accessed by instruction fetches.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The pseudocode enumeration `PrefetchHint` defines the prefetch hint types.

The `Hint_Prefetch()` pseudocode function signals to the memory system that memory accesses of the type `hint` to or from the specified `address` are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as preloading the specified address into one or more caches as indicated by the innermost cache level `target` and non-temporal hint `stream`.

For more information on `PLD`, `PLI`, and `PLDW`, see:

- *PLD, PLDW (immediate)* on page F6-2866.
- *PLD (literal)* on page F6-2868.
- *PLD, PLDW (register)* on page F6-2870.
- *PLI (immediate, literal)* on page F6-2872.
- *PLI (register)* on page F6-2875.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E2.4 Alignment support

This section describes alignment support. It contains the following subsections:

- *Instruction alignment*.
- *Unaligned data access*.
- *Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE* on page E2-2364.
- *Unaligned data access restrictions* on page E2-2364.

### E2.4.1 Instruction alignment

A32 instructions are word-aligned.

T32 instructions are halfword-aligned.

### E2.4.2 Unaligned data access

An ARMv8 implementation must support unaligned data accesses to Normal memory by some load and store instructions. As Table E2-1 shows, software can control whether a misaligned access to Normal memory by one of these instructions causes an Alignment fault Data Abort exception:

- By setting SCTLR.A, for unaligned accesses from any mode other than Hyp mode.
- By setting HSCTLR.A, for unaligned accesses from Hyp mode.

**Table E2-1 Alignment requirements of load/store instructions**

| Instructions | Alignment check | Result if check fails when: | |
|---|---|---|---|
| | | SCTLR.A or HSCTLR.A is 0 | SCTLR.A or HSCTLR.A is 1 |
| LDRB, LDREXB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, TBB | None | - | - |
| LDRH, LDRHT, LDRSH, LDRSHT, STRH, STRHT, TBH | Halfword | Unaligned access | Alignment fault |
| LDREXH, STREXH, LDAH, STLH, LDAEXH, STLEXH | Halfword | Alignment fault | Alignment fault |
| LDR, LDRT, STR, STRT<br>PUSH, encodings T3 and A2 only<br>POP, encodings T3 and A2 only | Word | Unaligned access | Alignment fault |
| LDREX, STREX, LDA, STL, LDAEX, STLEX | Word | Alignment fault | Alignment fault |
| LDREXD, STREXD, LDAEXD, STLEXD | Doubleword | Alignment fault | Alignment fault |
| All forms of LDM and STM, LDRD, RFE, SRS, STRD | Word | Alignment fault | Alignment fault |
| LDC, LDC2, STC, STC2 | Word | Alignment fault | Alignment fault |
| VLDM, VLDR, VPOP, VPUSH, VSTM, VSTR | Word | Alignment fault | Alignment fault |
| VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with standard alignment | Element size | Unaligned access | Alignment fault |
| VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with :<align> specified[a] | As specified by :<align> | Alignment fault | Alignment fault |

a. Previous versions of this manual used @<align> to specify alignment. Both forms are supported, see Chapter F7 *T32 and A32 Advanced SIMD and floating-point Instruction Descriptions* for more information.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Any unaligned access to any type of Device memory generates an Alignment fault, see *Alignment faults* on page G4-4085.

### E2.4.3 Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE

Any load instruction that is not faulted by the alignment restrictions shown in Table E2-1 on page E2-2363 and that loads the PC has CONSTRAINED UNPREDICTABLE behavior if the address it loads from is not word-aligned, see *Loads and Stores to unaligned locations* on page K1-5424. This overrules any permitted Load/Store behavior shown in Table E2-1 on page E2-2363.

### E2.4.4 Unaligned data access restrictions

The following points apply to unaligned data accesses in ARMv8:

- Accesses are not guaranteed to be single-copy atomic except at the byte access level, see *Atomicity in the ARM architecture* on page E2-2368.
- Unaligned accesses typically take a number of additional cycles to complete compared to a naturally-aligned access.
- An operation that performs an unaligned access can abort on any memory access that it makes, and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the boundary.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E2.5 Endian support

*General description of endianness in the ARM architecture* describes the relationship between endianness and memory addressing in the ARM architecture.

The following subsections then describe the endianness schemes supported by the architecture:

- *Instruction endianness*.
- *Data endianness* on page E2-2366.

### E2.5.1 General description of endianness in the ARM architecture

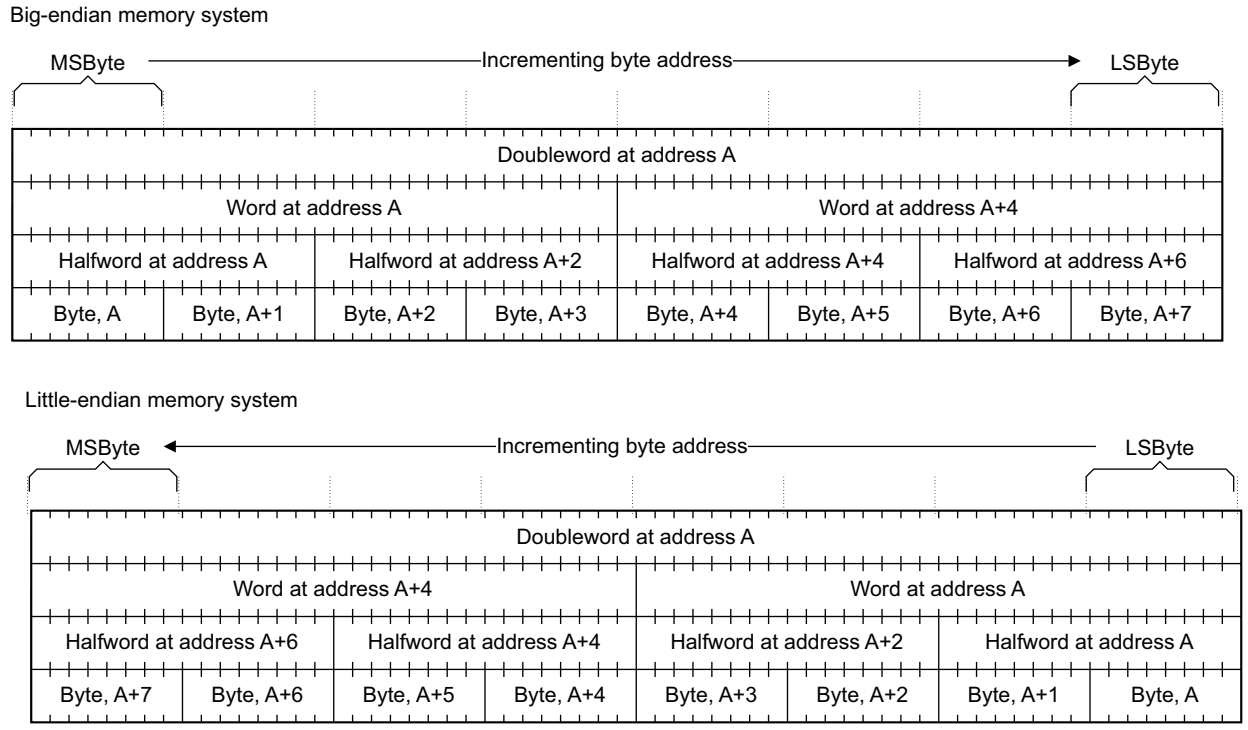
This section only describes memory addressing and the effects of endianness for data elements up to doubleword of 64 bits. However, this description can be extended to apply to larger data elements.

For an address A, Figure E2-2 shows, for big-endian and little-endian memory systems, the relationship between:

- The doubleword at address A.
- The words at addresses A and A+4.
- The halfwords at addresses A, A+2, A+4, and A+6.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, and A+7.

The terms in Figure E2-2 have the following definitions:

**MSByte** Most-significant byte.
**LSByte** Least-significant byte.




In this figure, *Byte, A+1* is an abbreviation for *Byte at address A+1*

**Figure E2-2 Endianness relationships in AArch32**

### E2.5.2 Instruction endianness

In ARMv8-A, the mapping of instruction memory is always little-endian.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### E2.5.3 Data endianness

The size of the data value that is loaded or stored is the size that is used for the purpose of endian conversion for floating-point, Advanced SIMD, and general-purpose register loads and stores.

Table E2-2 shows the element sizes of all the load/store instructions, for all instruction sets.

**Table E2-2 Element size of load/store instructions**

| Instructions | Element size |
|---|---|
| LDRB, LDREXB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, TBB | Byte |
| LDRH, LDREXH, LDRHT, LDRSH, LDRSHT, STRH, STREXH, STRHT, TBH | Halfword |
| LDR, LDRT, LDREX, STR, STRT, STREX | Word |
| LDRD, LDREXD, STRD, STREXD | Word |
| All forms of LDM, PUSH, POP, RFE, SRS, all forms of STM, | Word |
| LDC, LDC2, STC, STC2 | Word |
| Forms of VLDM, VLDR, VPOP, VPUSH, VSTM, VSTR that transfer 32-bit Si registers | Word |
| Forms of VLDM, VLDR, VPOP, VPUSH, VSTM, VSTR that transfer 64-bit Di registers | Doubleword |
| VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4 | Element size of the Advanced SIMD access |

CPSR.E determines the data endianness.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for Advanced SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for Advanced SIMD element and data structure loads and stores. For more information see *Endianness in Advanced SIMD* on page E2-2367.

#### Instructions to reverse bytes in registers

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Table E2-3 shows the instructions that provide this functionality in the A32 and T32 instruction sets:

**Table E2-3 Byte reversal instructions**

| Function | T32 / A32 Instruction | Notes |
|---|---|---|
| Reverse bytes in whole register | REV | For use with general purpose registers. |
| Reverse bytes in 16-bit halfwords | REV16 | For use with general purpose registers. |
| Reverse bytes in halfword and sign-extend | REVSH | For use with general purpose registers. |
| Reverse elements in doublewords, vector | VREV64 | For use with registers in the SIMD and floating-point register file |
| Reverse elements in words, vector | VREV32 | For use with registers in the SIMD and floating-point register file |
| Reverse elements in halfwords, vector | VREV16 | For use with registers in the SIMD and floating-point register file |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Endianness in Advanced SIMD

Advanced SIMD element Load/Store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used by the PE to load and store data correctly in both big-endian and little-endian systems.

Consider, for example, the A32 or T32 instruction:

```
VLD1.16 {D0}, [R1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure E2-3. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

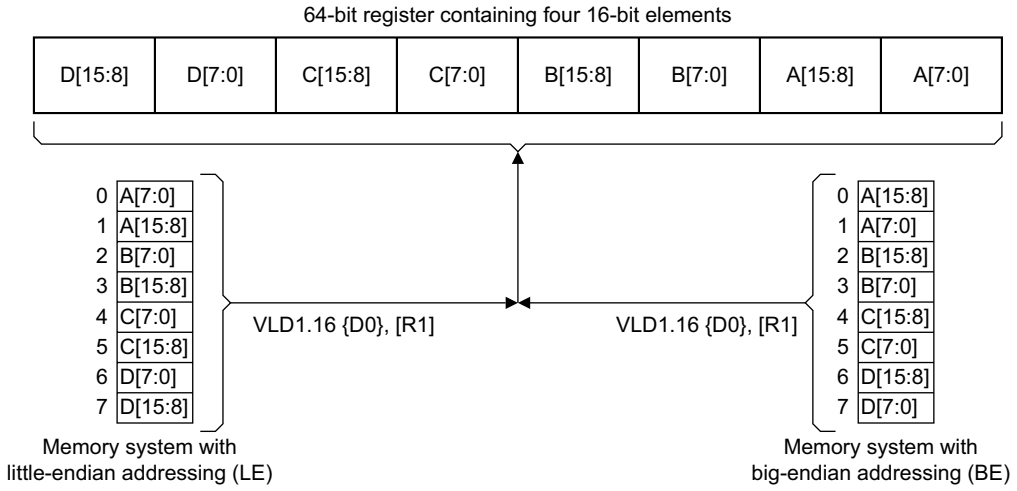



**Figure E2-3 Advanced SIMD byte order example for AArch32**

For information about the alignment of Advanced SIMD instructions see *Alignment support* on page E2-2363.

The `BigEndian()` pseudocode function determines the current endianness of the data.

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in Chapter J1 *ARMv8 Pseudocode*.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## E2.6 Atomicity in the ARM architecture

*Atomicity* is a feature of memory accesses, described as *atomic* accesses. The ARM architecture description refers to two types of atomicity, *single-copy atomicity* and *multicopy atomicity*. In the ARMv8 architecture, the atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information see:

- *Requirements for single-copy atomicity*.
- *Properties of single-copy atomic accesses* on page E2-2369.
- *Multi-copy atomicity* on page E2-2369.
- *Requirements for multi-copy atomicity* on page E2-2370.

For more information about the memory types, see *Memory type overview* on page E2-2357.

### E2.6.1 Requirements for single-copy atomicity

In AArch32 state, the single-copy atomic PE accesses are:

- All byte accesses.
- All halfword accesses to halfword-aligned locations.
- All word accesses to word-aligned locations.
- Memory accesses caused by `LDREXD` and `STREXD` instructions to doubleword-aligned locations.

`LDM`, `LDC`, `LDC2`, `LDRD`, `STM`, `STC`, `STC2`, `STRD`, `PUSH`, `POP`, `RFE`, `SRS`, `VLDM`, `VLDR`, `VSTM`, and `VSTR` instructions are executed as a sequence of word-aligned word accesses. Each 32-bit word access is guaranteed to be single-copy atomic. The architecture does not require subsequences of two or more word accesses from the sequence to be single-copy atomic.

`LDRD` and `STRD` accesses to 64-bit aligned locations are 64-bit single-copy atomic as seen by translation table walks and accesses to translation tables.

**Note**

This requirement has been added to avoid the need for complex measures to avoid atomicity issues when changing translation table entries, without creating a requirement that all locations in the memory system are 64-bit single-copy atomic. This addition means:

- The system designer must ensure that all writable memory locations that might be used to hold translations, such as bulk SDRAM, can be accessed with 64-bit single-copy atomicity.
- Software must ensure that translation tables are not held in memory locations that cannot meet this atomicity requirement, such as peripherals that are typically accessed using a narrow bus.

This requirement places no burden on read-only memory locations for which reads have no side effects, since it is impossible to detect the size of memory accesses to such locations.

---

Advanced SIMD element and structure loads and stores are executed as a sequence of accesses of the element or structure size. The architecture requires the element accesses to be single-copy atomic if and only if both:

- The element size is 64 bits, or smaller.
- The elements are naturally aligned.

Accesses to 64-bit elements or structures that are 32-bit aligned are executed as a sequence of 32-bit accesses, each of which is single-copy atomic. The architecture does not require subsequences of two or more 32-bit accesses from the sequence to be single-copy atomic.

When an access is not single-copy atomic by the rules described in this section, it is executed as a sequence of one or more accesses that aggregate to the size of the original access. Each of the accesses in this sequence is single-copy atomic, at least at the byte level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

In this section, the terms *before the write operation* and *after the write operation* mean before or after the write operation has had its effect on the coherence order of the bytes of the memory location accessed by the write operation.

If, according to these rules, an instruction is executed as a sequence of accesses, a synchronous Data Abort exception or Debug state entry can be taken during that sequence. This causes execution of the instruction to be abandoned. See *Data Abort exception* on page G1-3821.

If the synchronous Data Abort exception is returned from using the preferred return address, the instruction that generated the sequence of accesses is re-executed and so any access that was performed before the exception was taken is repeated. This also applies to an exit from Debug state.

**Note**

The exception behavior for these multiple access instructions means they are not suitable for use for writes to memory for the purpose of software synchronization.

For implicit accesses:

- Cache linefills and evictions have no effect on the single-copy atomicity of explicit transactions or instruction fetches.
- Instruction fetches are single-copy atomic:
  - At 32-bit granularity in A32 state.
  - At 16-bit granularity in T32 state.
- *Concurrent modification and execution of instructions* on page E2-2370 describes additional constraints on the behavior of instruction fetches.
- Translation table walks are performed using accesses that are single-copy atomic:
  - At 32-bit granularity when using Short-descriptor format translation tables.
  - At 64-bit granularity when using Long-descriptor format translation tables.

### E2.6.2 Properties of single-copy atomic accesses

A read or write operation that is *single-copy atomic* has the following properties:

1. For a single-copy atomic store, if the store overlaps another single-copy atomic store, then all of the writes from one of the stores are inserted into the *Coherence order* of each overlapping byte before any of the writes of the other store are inserted into the Coherence orders of the overlapping bytes.
2. If a single-copy atomic load overlaps a single-copy atomic store and for any of the overlapping bytes the load returns the data written by the write inserted into the Coherence order of that byte by the single-copy atomic store then the load must return data from a point in the Coherence order no earlier than the writes inserted into the Coherence order by the single-copy atomic store of all of the overlapping bytes.

### E2.6.3 Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

**Note**

Writes that are not coherent are not multi-copy atomic.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### E2.6.4 Requirements for multi-copy atomicity

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory with the non-Gathering attribute, writes that are single-copy atomic are also multi-copy atomic.

For Device memory with the Gathering attribute, writes are not required to be multi-copy atomic.

### E2.6.5 Concurrent modification and execution of instructions

The ARMv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where the instruction before modification or the instruction after modification is a:

- `B`, `BL`, `NOP`, `BKPT`, `SVC`, `HVC`, or `SMC` A32 instruction
- `B, BL, BLX, NOP`, `BKPT`, or `SVC` 16-bit T32 instruction.

In addition, for the T32 instructions:

- The most-significant halfword of a32-bit `BL` immediate instruction can be concurrently modified to the most significant halfword of another `BL` immediate instruction:
  - This means that the most significant bits of the immediate value can be modified.
- The most-significant halfword of a 32-bit `BLX` immediate instruction can be concurrently modified to the most significant halfword of another `BLX` immediate instruction:
  - This means that the most significant bits of the immediate value can be modified.
- The most-significant halfword of a 32-bit `BL` immediate or `BLX` immediate instruction can be concurrently modified to a T32 16-bit `B`, `BL`, `BLX`, `BKPT`, or `SVC` instruction. This modification also works in reverse.
- The least-significant halfword of a 32-bit `BL` immediate instruction can be concurrently modified to the least significant halfword of another `BL` instruction with a different immediate:
  - This means that the least significant bits of the immediate value can be modified.
- The least-significant halfword of a 32-bit `BLX` immediate instruction can be concurrently modified to the least significant halfword of another `BLX` immediate instruction with a different immediate:
  - This means that the least significant bits of the immediate value can be modified.
- The least-significant halfword of a 32-bit `B` immediate instruction with a condition field can be concurrently modified to the least significant halfword of another 32-bit `B` immediate instruction with a condition field with a different immediate:
  - This means that the least significant bits of the immediate value can be modified.
- The least-significant halfword of a 32-bit `B` immediate instruction without a condition field can be concurrently modified to the least significant halfword of another 32-bit `B` immediate instruction without a condition field:
  - This means that the least significant bits of the immediate value can be modified.

**Note**

In the T32 instruction set:

- The only encodings of `BKPT` and `SVC` are 16-bit.
- The only encoding of `BL` is 32-bit.

For the instructions explicitly identified in this section, the architecture guarantees that, after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The instructions to which this applies are the B, BL, NOP, BKPT, SVC, HVC, and SMC instructions.

For both instruction sets, if one thread of execution changes a conditional branch instruction to another conditional branch instruction, and the change affects both the condition field and the branch target, execution of the changed instruction by another thread of execution before the change is synchronized can lead to either:

- The old condition being associated with the new target address.
- The new condition being associated with the old target address.

These possibilities apply regardless of whether the condition, either before or after the change to the branch instruction, is the *always* condition.

For all other instructions, to avoid UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. No PE must be executing an instruction when another PE is modifying that instruction.

2. To ensure that the modified instructions are observable, the PE that modified the instructions must issue the following sequence of instructions and operations:

```
; Coherency example for self-modifying code
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first
; line instead of STR for a 16-bit instruction.
STR <Rt>, [Rn]
DCCMVAU Rn                ; Clean data cache by MVA to point of unification (PoU)
DSB                       ; Ensure visibility of the data stored
ICIMVAU Rn                ; Invalidate instruction cache by VA to PoU
BPIMVA  Rn                ; Invalidate branch predictor by MVA to PoU
DSB                       ; Ensure completion of the invalidations
```

   **Note**

   The DCCMVAU operation is not required if the area of memory is either Non-cacheable or Write-through Cacheable.

3. In a multiprocessor system, the ICIMVAU and BPIMVA are broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. However, once the modified instructions are observable, each PE that is executing the modified instructions must issue the following instruction to ensure execution of the modified instructions:

```
ISB                           ; Synchronize fetched instruction stream
```

For more information about the required synchronization operation, see *Synchronization and coherency issues between data and instruction accesses* on page E2-2360.

**Note**

For information about memory accesses caused by instruction fetches, see *Ordering requirements* on page E2-2373.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E2.7 Memory ordering

This section describes observation ordering. It contains the following subsections:

- *Observability and completion*.
- *Ordering requirements* on page E2-2373.
- *Memory barriers* on page E2-2375.
- *Summary of the memory ordering rules* on page E2-2379.

For information on endpoint ordering of memory accesses, see *Reordering* on page E2-2390.

In the ARMv8 memory model, the shareability memory attribute indicates whether hardware must ensure memory coherency.

The ARMv8 memory system architecture defines additional attributes and associated behaviors, defined in the system level section of this manual. See:

- Chapter G3 *The AArch32 System Level Memory Model*.
- Chapter G4 *The AArch32 Virtual Memory System Architecture*.

See also *Mismatched memory attributes* on page E2-2392.

### E2.7.1 Observability and completion

An *observer* is a master in the system that is capable of observing memory accesses. For a PE, the following mechanisms must be treated as independent observers:

- The mechanism that performs reads or writes to memory.
- A mechanism that causes an instruction cache to be filled from memory or that fetches instructions to be executed directly from memory. These are treated as reads.
- A mechanism that performs translation table walks. These are treated as reads.

The set of observers that can observe a memory access is defined by the system.

In the definitions in this subsection, *subsequent* means whichever of the following is appropriate to the context:

- After the point in time where the location is observed by that observer.
- After the point in time where the location is globally observed.

For all memory:

- A write to a location in memory is said to be *observed* by an observer when:
  - A subsequent read of the location by the same observer returns the value written by the observed write, or written by a write to that location by any observer that is sequenced in the *Coherence order* of the location after the observed write.
  - A subsequent write of the location by the same observer is sequenced in the *Coherence order* of the location after the observed write.
- A write to a location in memory is said to be *globally observed* for a shareability domain or set of observers when:
  - A subsequent read of the location by any observer in that shareability domain returns the value written by the globally observed write, or written by a write to that location by any observer that is sequenced in the *Coherence order* of the location after the globally observed write.
  - A subsequent write of the location by any observer in that shareability domain is sequenced in the *Coherence order* of the location after the globally observed write.
- A read of a location in memory is said to be observed by an observer when a subsequent write to the location by the same observer has no effect on the value returned by the read.
- A read of a location in memory is said to be globally observed for a shareability domain when a subsequent write to the location by any observer in that shareability domain has no effect on the value returned by the read.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Additionally, for Device-nGnRnE memory:

- A read or write of a memory-mapped location in a peripheral that exhibits side-effects is said to be observed, and globally observed, only when the read or write:
  - Meets the general conditions listed.
  - Can begin to affect the state of the memory-mapped peripheral.
  - Can trigger all associated side-effects, whether they affect other peripheral devices, processors, or memory.

  **Note**

  This definition is consistent with the memory access having reached the peripheral.

For all memory, the completion rules are defined as:

- A read or write is complete for a shareability domain when all of the following are true:
  - The read or write is globally observed for that shareability domain.
  - Any translation table walks associated with the read or write are complete for that shareability domain.
- A translation table walk is complete for a shareability domain when the memory accesses associated with the translation table walk are globally observed for that shareability domain, and the TLB is updated.
- A cache or TLB maintenance instruction is complete for a shareability domain when the effects of the instruction are globally observed for that shareability domain, and any translation table walks that arise from the instruction are complete for that shareability domain.

  The completion of any cache or TLB maintenance instruction includes its completion on all processors that are affected by both the instruction and the `DSB` operation that is required to guarantee visibility of the maintenance instruction.

### Completion of side-effects of accesses to Device memory

The completion of a memory access to Device memory is not guaranteed to be sufficient to determine that the side-effects of the memory access are visible to all observers. The mechanism that ensures the visibility of side-effects of a memory access is IMPLEMENTATION DEFINED.

## E2.7.2 Ordering requirements

ARMv8 defines restrictions for the permitted ordering of memory accesses. These restrictions depend on the memory type of the addresses that are accessed, see *Memory types and attributes* on page E2-2382.

**Note**

See *Summary of the memory ordering rules* on page E2-2379 for the definition of *address dependency*.

For accesses to all memory types, the only stores by an observer that can be observed by another observer are those stores that have been *Architecturally executed*. Speculative writes by an observer cannot be observed by another observer. For the purposes of this requirement, speculative writes are all of:

- Writes generated by store instructions that appear in the *Execution stream* after a branch that is not architecturally resolved.
- Writes generated by store instructions that appear in the *Execution stream* after an instruction where a synchronous exception condition has not been architecturally resolved.
- Writes generated by conditional store instructions for which the conditions for the instruction have not been architecturally resolved.
- Writes generated by store instructions for which the data being written comes from a register that has not been architecturally committed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following additional restrictions apply to the order in which accesses to memory are observed:

- Reads and writes can be observed in any order provided the following constraints are met:
  - If an address dependency exists between two reads or between a read and a write, then those memory accesses are observed in program order by all observers within the common shareability domain of the memory addresses being accessed.
  - Ordering can be achieved by using a `DMB` or `DSB` barrier. For more information on `DMB` and `DSB` instructions, see *Memory barriers* on page E2-2375.
- Reads and writes to the same address are coherent within the shareability domain of the memory address being accessed.
- Two reads to the same address by the same observer are observed in program order by all observers within the shareability domain of the memory address being accessed.
- Writes are not required to be multi-copy atomic. This means that in the absence of barriers, the observation of a store by one observer does not imply the observation of the store by another observer.
- Instructions that access multiple elements have no defined ordering requirements for the memory accesses relative to each other.

For Device memory with the non-Reordering attribute, the order of memory accesses arriving at a *single peripheral* is the same as occurs in a *Simple sequential execution* on page Glossary-5775 of the program. This means the accesses arrive in program order. This ordering applies for all accesses using any of the memory types with the non-Reordering attribute, which means Device-nGnRE accesses are ordered with respect to Device-nGnRnE accesses to the same peripheral. If the memory accesses are not to a peripheral then there are no ordering restrictions from the non-Reordering attribute. For the purposes of this definition, a *single peripheral* is a region of memory of an IMPLEMENTATION DEFINED size that is defined by the peripheral.

Memory accesses caused by instruction fetches are not required to be observed in program order, unless they are separated by an `ISB` or other context synchronization event.

### Address dependencies and order

In the ARMv8 architecture, a register data dependency between the value returned by a load instruction and the address used by a subsequent memory transaction creates order between that load instruction and the subsequent memory transaction.

A register data dependency exists between a first data value and a second data value when either:

- The register used to hold the first data value is used in the calculation of the second data value, and the calculation between the first data value and the second data value does not consist of either:
  - A conditional branch whose condition is determined by the first data value.
  - A conditional selection, move, or computation whose condition is determined by the first data value, where the input data values for the selection, move, or computation do not have a data dependency on the first data value.
- There is a register data dependency between the first data value and a third data value, and between the third data value and the second data value.

**Note**

A register data dependency can exist even if the value of the first data value is discarded as part of the calculation, as might be the case if it is ANDed with `0x0` or if arithmetic using the first data value cancels out its contribution.

For example, each of the following code sequences creates order between the memory transactions:

**Sequence 1**
```
LDR R1,[R2]
AND R1, R1, #0
LDR R4,[R3, R1]
```

**Sequence 2**
```
LDR R1, [R2]
ADD R3, R3, R1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
SUB R3, R3, R1
STR R4, [R3]
```

### E2.7.3 Memory barriers

The ARM architecture is a weakly ordered memory architecture that supports out of order completion. *Memory barrier* is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring Load/Store instructions. The memory barriers defined by the ARMv8 architecture provide a range of functionality, including:

- Ordering of Load/Store instructions.
- Completion of Load/Store instructions.
- Context synchronization.

The following subsections describe the ARMv8 memory barrier instructions:

- *Instruction Synchronization Barrier (ISB)*.
- *Data Memory Barrier (DMB)* on page E2-2376.
- *Data Synchronization Barrier (DSB)* on page E2-2377.
- *Shareability and access limitations on the data barrier operations* on page E2-2377.
- *Load-Acquire, Store-Release* on page E2-2378.

**Note**

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are only available when software execution is at EL1 or higher.

The DMB and DSB memory barriers affect reads and writes to the memory system generated by Load/Store instructions and data or unified cache maintenance instructions being executed by the PE. Instruction fetches or accesses caused by a hardware translation table access are not explicit accesses.

AArch32 state also supports the legacy barrier instructions CP15DMB, CP15DSB, and CP15ISB. These instructions are executed as MCRs using the appropriate encoding, and are accessible from EL0. However, for performance reasons ARM deprecates any use of these operations, and strongly recommends that software uses the DMB, DSB, and ISB instructions described in this section instead. Optionally, an implementation can support a CP15DEN control that supervisory software can use to disable use of these instructions, meaning the corresponding MCR encodings are UNDEFINED. When the CP15BEN control is supported, setting one of the following CP15BEN fields to 0 makes execution of CP15DMB, CP15DSB, and CP15ISB UNDEFINED:

- SCTLR_EL1.CP15BEN, for execution of these instructions at EL0 using AArch32 when EL1 is using AArch64.
- SCTLR.CP15BEN, for execution of these instructions at EL0 or EL1 when EL1 is using AArch32.
- HSCTLR.CP15BEN, for execution of these instructions at EL2 when EL2 is using AArch32.

#### Instruction Synchronization Barrier (ISB)

An ISB instruction flushes the pipeline in the PE, so that all instructions that come after the ISB instruction in program order are fetched from the cache or memory only after the ISB instruction has completed. Using an ISB ensures that the effects of context-changing operations executed before the ISB are visible to the instructions fetched after the ISB instruction. Examples of context-changing operations that require the insertion of an ISB instruction to ensure the effects of the operation are visible to instructions fetched after the ISB instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the ISB instruction only take effect after the ISB has been executed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The pseudocode function for the operation of an ISB is InstructionSynchronizationBarrier().

See also *Memory barriers* on page G3-3987.

### Data Memory Barrier (DMB)

The DMB instruction is a data memory barrier. The PE that executes the DMB instruction is referred to as the executing PE, PEe. The DMB instruction takes an <option> argument that specifies the shareability domains and access types to which the instruction applies, see *Shareability and access limitations on the data barrier operations* on page E2-2377.

If the required shareability is *Full system* then the operation applies to all observers within the system.

A DMB creates two groups of memory accesses, Group A and Group B:

**Group A** Contains:

- All explicit memory accesses of the required access types from observers in the same required shareability domain as PEe that are observed by PEe before the DMB instruction. These accesses include any accesses of the required access types performed by PEe.
- All loads of required access types from an observer PEx in the same required shareability domain as PEe that have been observed by any given different observer, PEy, in the same required shareability domain as PEe before PEy has performed a memory access that is a member of Group A.

**Group B** Contains:

- All explicit memory accesses of the required access types by PEe that occur in program order after the DMB instruction.
- All explicit memory accesses of the required access types by any given observer PEx in the same required shareability domain as PEe that can only occur after a load by PEx has returned the result of a store that is a member of Group B.

Any observer with the same required shareability domain as PEe observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined by the shareability and cacheability of the memory addresses accessed by the group members.

If members of Group A and members of Group B access the same memory-mapped peripheral of arbitrary system-defined size, then members of Group A that are accessing Device or Normal Non-cacheable memory arrive at that peripheral before members of Group B that are accessing Device or Normal Non-cacheable memory. Where the members of Group A and Group B that must be ordered are from the same PE, a DMB NSH is sufficient for this guarantee.

**Note**

- A memory access might be in neither Group A nor Group B. The DMB does not affect the order of observation of such a memory access.
- The second part of the definition of Group A is recursive. Ultimately, membership of Group A derives from the observation by PEy of a load before PEy performs an access that is a member of Group A as a result of the first part of the definition of Group A.
- The second part of the definition of Group B is recursive. Ultimately, membership of Group B derives from the observation by any observer of an access by PEe that is a member of Group B as a result of the first part of the definition of Group B.

DMB only affects memory accesses and the operation of data cache and unified cache maintenance instructions, see *The ARMv8 cache maintenance functionality* on page G3-3957. It has no effect on the ordering of any other instructions executing on the PE. A DMB intended to ensure the completion of cache maintenance instructions must have an access type of both loads and stores.

The pseudocode function for the operation of a DMB is DataMemoryBarrier().

See also *Memory barrier instructions* on page G3-3978 and *Memory barriers* on page G3-3987.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Data Synchronization Barrier (DSB)

The DSB instruction is a memory barrier, that synchronizes the execution stream with memory accesses.

The DSB instruction takes an <option> argument that specifies the shareability domains and access types to which the instruction applies, see *Shareability and access limitations on the data barrier operations*.

If the required shareability is *Full system* then the operation applies to all observers within the system.

A DSB behaves as a DMB with the same arguments, and also has the additional properties defined in this section. The PE that executes the DSB instruction is referred to as the executing PE, PEe.

Execution of a DSB at EL2 ensures that any memory accesses caused by speculative translation table walks from the Non-secure PL1&0 translation regime have been observed.

For more information, see *Use of out-of-context translation regimes* on page G4-3998.

A DSB completes when all of the following apply:

- All explicit memory accesses that are observed by PEe before the DSB is executed and are of the required access types, and are from observers in the same required shareability domain as PEe, are complete for the set of observers in the required shareability domain.
- If the required access types of the DSB is reads and writes, then all cache and branch predictor maintenance instructions and all TLB maintenance instructions issued by PEe before the DSB are complete for the required shareability domain.

In addition, no instruction that appears in program order after the DSB instruction can execute until the DSB completes.

See also *Memory barrier instructions* on page G3-3978 and *Memory barriers* on page G3-3987.

### Shareability and access limitations on the data barrier operations

The DMB and DSB instructions can each take an optional limitation argument that specifies:

- The shareability domain over which the instruction must operate. This is one of:
  - Full system.
  - Outer Shareable.
  - Inner Shareable.
  - Non-shareable.
- The accesses for which the instruction operates. This is one of:
  - Read and write accesses in Group A and Group B.
  - Write accesses only in Group A and Group B.
  - Read access only in Group A and read and write accesses in Group B.

**Note**

This form of a DMB or DSB instruction can be described as a Load-Load/Store barrier.

Table E2-4 shows how these options are encoded in the <option> field of the instruction:

**Table E2-4 Encoding of the DMB and DSB <option> parameter**

| Accesses | | Shareability domain | | | |
|---|---|---|---|---|---|
| **Group A** | **Group B** | **Full system** | **Outer Shareable** | **Inner Shareable** | **Non-shareable** |
| Reads and writes | Reads and writes | SY | OSH | ISH | NSH |
| Writes | Writes | ST | OSHST | ISHST | NSHST |
| Reads | Reads and writes | LD | OSHLD | ISHLD | NSHLD |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If no <option> is specified then the instruction operates for read and write accesses, over the full system, meaning the operation is the same as for the SY option. See the instruction descriptions for more information:

- *DMB* on page F6-2671.
- *DSB* on page F6-2673.

**Note**

ISB also supports an optional limitation argument that can only contain one value that corresponds to full system operation, see *ISB* on page F6-2688.

### Load-Acquire, Store-Release

ARMv8 provides a set of instructions with Acquire semantics for loads, and Release semantics for stores.

For all memory types, these instructions have the following ordering requirements:

- A Store-Release followed by a Load-Acquire is observed in program order by each observer within the shareability domain of the memory address being accessed by the Store-Release and the memory address being accessed by the Load-Acquire.
- For a Load-Acquire, observers in the shareability domain of the address accessed by the Load-Acquire observe accesses in the following order:
  1. The read caused by the Load-Acquire.
  2. Reads and writes caused by loads and stores that appear in program order after the Load-Acquire for which the shareability of the address accessed by the load or store requires that the observer observes the access.

  There are no other ordering requirements on loads or stores that appear before the Load-Acquire.
- For a Store-Release, observers in the shareability domain of the address accessed by the Store-Release observe accesses in the following order:
  1. All of the following for which the shareability of the address accessed requires that the observer observes the access:
     - Reads and writes caused by loads and stores that appear in program order before the Store-Release.
     - Writes that were observed by the PE executing the Store-Release before it executed the Store-Release.
  2. The write caused by the Store-Release.

  There are no additional ordering requirements on loads or stores that appear in program order after the Store-Release.
- All Store-Release instructions must be multi-copy atomic when they are observed with Load-Acquire instructions.

In addition, for accesses to a memory-mapped peripheral of an arbitrary system-defined size that are defined as any type of Device memory accesses, these instructions have the following requirements:

- A Load-Acquire to an address in the memory-mapped peripheral will ensure that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed after the Load-Acquire will arrive at the memory-mapped peripheral after the memory access of the Load-Acquire.
- A Store-Release to an address in the memory-mapped peripheral will ensure that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed before the Store-Release will arrive at the memory-mapped peripheral before the memory access of the Store-Release.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If a Load-Acquire to a memory address in the memory-mapped peripheral has observed the value stored to that address by a Store-Release, then any memory access to the memory-mapped peripheral that is architecturally required to be ordered before the memory access of the Store-Release will arrive at the memory-mapped peripheral before any memory access to the same peripheral that is architecturally required to be ordered after the memory access of the Load-Acquire.

Load-Acquire and Store-Release, other than LDAEXD and STLEXD, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.

LDAEXD and STLEXD access two data elements. The address supplied to the instructions must be doubleword aligned, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction only has the release semantics if the store is successful.

**Note**

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
  - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.
- The Load-Acquire/Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction.

Table E2-5 summarizes the Load-Acquire/Store-release instructions:

**Table E2-5 Load-Acquire/Store-Release instructions**

| Data type | Load-Acquire | Store-Release | Load-Acquire Exclusive | Store-Release Exclusive |
|---|---|---|---|---|
| 32-bit word | LDA | STL | LDAEX | STLEX |
| 16-bit halfword | LDAH | STLH | LDAEXH | STLEXH |
| 8-bit byte | LDAB | STLB | LDAEXB | STLEXB |
| 64-bit doubleword | - | - | LDAEXD | STLEXD |

### E2.7.4 Summary of the memory ordering rules

The following is a concise list of the situations that are required, by the ARM architecture specification, to cause externally-visible order of memory. This ordering means that if memory transaction A has externally visible order ahead of memory transaction B, then all observers within the shareability domains of A and B will observe A before B. See *Terms used in the summary of the memory ordering rules* on page E2-2380 for definitions of the terms used.

**Note**

This list applies to both AArch32 state and AArch64 state, and is consistent with the requirements of ARMv7.

1. DMB and DSB barrier instructions, and load acquire/store release instructions, create externally-visible order as defined by those instructions.
2. A True or False Address dependency from a Load to a Load or from a Load to a Store creates externally-visible order.
3. A True Control dependency from a Load to an ISB instruction creates externally-visible order between the load and any memory accesses after the ISB instruction.
4. A True Register data dependency from a Load to a Store creates externally-visible order.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

5. A True Control dependency from a Load to a Store creates externally-visible order.

6. Memory is coherent within the shareability domain of a memory address, which means there is a total order of all writes to that address that all observers within that shareability domain will agree on.

   **Note**

   A consequence of this is that reads to the same address by the same processor are observed in order.

7. A Dependency from a Store to a Load through memory between different PEs creates externally-visible order but stores are not multi-copy atomic except where explicitly defined to be by the definition of the store.

   **Note**

   A consequence of the lack of multi-copy atomicity is that a Store to Load dependency through memory on the same PE does not create externally-visible order.

No other effects are required to create externally visible order in the ARM architecture.

## Terms used in the summary of the memory ordering rules

The summary uses the following terms:

**Register data dependency**

This is defined in *Address dependencies and order* on page E2-2374.

**False Register data dependency**

A False Register data dependency is a Register data dependency where no register in the system holds a variable for which a change of the first data value causes a change of the second data value.

**True Register data dependency**

A True Register data dependency is a Register data dependency that is not a false Register data dependency.

**True Address dependency**

A True Address dependency between a load and a subsequent memory transaction exists where there is a True Register data dependency between the data value returned from the load and the address used by the subsequent memory transaction.

**False Address dependency**

A False Address dependency between a load and a subsequent memory transaction exists where there is a False Register data dependency between the data value returned from the load and the address used by the subsequent memory transaction.

**True Control dependency**

A True Control dependency between a load and a subsequent instruction exists:

- Where there is a True Register data dependency between the data value returned from the load and data value used in the evaluation of a conditional branch and the subsequent instruction is only executed as a result of one of the possible outcomes of that conditional branch.
- Where there is a True Register data dependency between the data value returned from the load and the data value used in the evaluation of a subsequent instruction that is a conditional selection, move or computation for which both:
  - The condition is determined by the returned data value.
  - No input data value for the selection, move or computation has a register data dependency on the returned data value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Dependency from a Store to a Load through memory**

A Dependency from a Store to a Load through memory exists where the Store and Load are to the same physical address, and value returned by the Load is the value that was written by the Store, and could not be the value that was previously held in that memory address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E2.8 Memory types and attributes

In ARMv8 the ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- *Normal memory*.
- *Device memory* on page E2-2387.
- *Memory access restrictions* on page E2-2391.

### E2.8.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform speculative data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- A write to a memory location with the Normal attribute completes in finite time. This means that it is globally observed for the shareability domain of the memory location in finite time. For a Non-cacheable location, the location is observed by all observers in finite time.
- A completed write to a memory location with the Normal attribute is globally observed for the shareability domain of the memory location in finite time without the need for explicit cache maintenance instructions or barriers. For a Non-cacheable location, the completed write is globally observed for all observers in finite time without the need for explicit cache maintenance instructions or barriers.
- Writes to a memory location with the Normal memory attribute that is Non-cacheable must reach the endpoint for that location in the memory system in finite time.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register Load/Store instructions. See *Multi-register loads and stores that access Normal memory* on page E2-2386.

**Note**

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side-effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.
- An instruction that generates a sequence of accesses as described in *Atomicity in the ARM architecture* on page E2-2368 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

The following sections describe the other attributes for Normal memory:

- *Shareable Normal memory* on page E2-2383.
- *Non-shareable Normal memory* on page E2-2384.
- *Write-Through Cacheable, Write-Back Cacheable and Non-cacheable Normal memory* on page E2-2384.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See also:

- *Multi-register loads and stores that access Normal memory* on page E2-2386.
- *Atomicity in the ARM architecture* on page E2-2368.
- *Memory barriers* on page E2-2375. For accesses to Normal memory, a `DMB` instruction is required to ensure the required ordering.
- *Concurrent modification and execution of instructions* on page E2-2370.

## Shareable Normal memory

A Normal memory location has a Shareability attribute that is defined as one of:

- Inner Shareable.
- Outer Shareable.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, that hardware must enforce. They do not affect the coherency requirements of instruction fetches, see *Synchronization and coherency issues between data and instruction accesses* on page E2-2360.

**Note**

- System designers can use the Shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the Shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different PEs.
- This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

### *Shareable, Inner Shareable, and Outer Shareable Normal memory*

The ARM architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is only a member of a single Inner Shareability domain.
- Each observer is only a member of a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

**Note**

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.
- The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The details of the use of the Shareability attributes are system-specific. Example E2-1 shows how they might be used.

**Example E2-1 Use of shareability attributes**

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different Shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same Shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable Shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable Shareability domain.

For Shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

### Non-shareable Normal memory

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

### Write-Through Cacheable, Write-Back Cacheable and Non-cacheable Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.

Also, for cacheable Normal memory regions:

- A region might be assigned a cache allocation hint.
- It is IMPLEMENTATION DEFINED whether the Write-Through Cacheable and Write-Back Cacheable attributes can have an additional attribute of Transient or Non-transient, see *Transient cacheability attribute* on page E2-2386.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

A memory location can be marked as having different cacheability attributes, for example when using aliases in a virtual to physical address mapping:

- If the attributes differ only in the cache allocation hint this does not affect the behavior of accesses to that location.
- For other cases see *Mismatched memory attributes* on page E2-2392.

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the Shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

- A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.

**Note**

Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the ARM architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.
- No cache that is controlled by the Inner cacheability attributes can lie outside a cache that is controlled by the Outer cacheability attributes.
- An implementation might not have any outer cache.

Example E2-2, Example E2-3 on page E2-2386, and Example E2-4 on page E2-2386 describe the possible ways of implementing a system with three levels of cache, *level 1* (L1) to *level 3* (L3).

**Note**

- L1 cache is the level closest to the PE, see *Memory hierarchy* on page E2-2358.
- When managing coherency, system designs must consider both the inner and outer cacheability attributes, as well as the Shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

**Example E2-2 Implementation with two inner and one outer cache levels**

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 and L2 cache.
- The Outer cacheability attribute applied to L3 cache.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Example E2-3 Implementation with three inner and no outer cache levels**

Implement the three levels of cache in the system, L1 to L3, with the Inner cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer cacheability attribute.

**Example E2-4 Implementation with one inner and two outer cache levels**

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 cache.
- The Outer cacheability attribute applied to L2 and L3 cache.

#### *Transient cacheability attribute*

In ARMv8, it is IMPLEMENTATION DEFINED whether a Transient attribute is supported for cacheable Normal memory regions. If an implementation supports this attribute, the set of possible cacheability attributes for a Normal memory region becomes:

- Write-Through Cacheable, Non-transient.
- Write-Back Cacheable, Non-transient.
- Write-Through Cacheable, Transient.
- Write-Back Cacheable, Transient.
- Non-cacheable.

The cacheability attribute can be defined independently for the inner and outer levels of caching.

The transient attribute indicates that the benefit of caching is for a relatively short period. Therefore it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.

**Note**

The architecture does not specify what is meant by a *relatively short period*.

The description of the MAIR0, MAIR1, HMAIR0, and HMAIR1 registers includes the assignment of the Transient attribute in an implementation that supports this option.

### Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more registers from the SIMD and floating-point register file from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### E2.8.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for ARMv8 Device memory are:

**Gathering** Identified as G or nG, see *Gathering* on page E2-2389.

**Reordering** Identified as R or nR, see *Reordering* on page E2-2390.

**Early Write Acknowledgement hint**

Identified as E or nE, see *Early Write Acknowledgement* on page E2-2390.

The ARMv8 Device memory types are:

**Device-nGnRnE** Device non-Gathering, non-Reordering, No Early write acknowledgement.

Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.

**Device-nGnRE** Device non-Gathering, non-Reordering, Early Write Acknowledgement.

Equivalent to the Device memory type in earlier versions of the architecture.

**Device-nGRE** Device non-Gathering, Reordering, Early Write Acknowledgement.

ARMv8 adds this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.

The Device-nGRE memory type is introduced into the AArch32 translation table formats when the PE is using the Long Descriptor Translation Table format.

**Device-GRE** Device Gathering, Reordering, Early Write Acknowledgement.

ARMv8 adds this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that speculative accesses to Device-GRE memory is forbidden.

The Device-GRE memory type is introduced into the AArch32 translation table formats when the PE is using the Long Descriptor Translation Table format.

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.

**Note**

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
- The architecture does not require an implementation to distinguish between each of these memory types and ARM recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.
- Earlier versions of the ARM architecture defined the following memory types:
  - Strongly-ordered memory. This is the equivalent of the Device-nGnRnE memory type.
  - Device memory. This is the equivalent of the Device-nGnRE memory type.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An exception to this applies:

— Reads generated by the Advanced SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

**Note**

- An instruction that generates a sequence of accesses as described in *Atomicity in the ARM architecture* on page E2-2368 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program only defines a single access. For this reason, ARM strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.

  — Write speculation that is visible to other observers is prohibited for all memory types.

- A write to a memory location with any Device memory attribute completes in finite time. This means that it is globally observed for all observers in the system in finite time.
- If a location with any Device memory attribute changes without an explicit write by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.
- A completed write to a memory location with any Device memory attribute is globally observed for all observers in finite time without the need for explicit maintenance.
- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.
- A memory location with any Device memory attribute cannot be allocated into a cache.
- Writes to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Typically, the endpoint is a peripheral or some physical memory.
- All accesses to memory with any Device memory attribute must be aligned. Any unaligned access generates an Alignment fault at the first stage of translation that defined the location as being Device.

  **Note**

  In the Non-secure PL1&0 translation regime in systems where HCR.TGE==1 and HCR.DC==0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes the value of HSR.ISS.[24] to be 0.

- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as Execute-never for all Exception levels.

  **Note**

  This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as Execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as Execute-never for all Exception levels is a programming error.

See also *Memory access restrictions* on page E2-2391.

The memory types for Translation table walks cannot be defined as any Device memory type within the TCR. For the Non-secure EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR.PTW bit is 1, a stage 2 permission fault is generated if a first stage translation table walk is made to any Device memory type.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For instruction fetches, if branches cause the program counter to point to an area of memory with the Device attribute which is not marked as Execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

### Gathering

In the Device memory attribute:

**G** Indicates that the location has the Gathering attribute.

**nG** Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All access occur at their programmed size, except that there is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register Load/Store instructions. See *Multi-register loads and stores that access Device memory* on page E2-2391.

Gathering between memory accesses separated by a memory barrier that affects those memory accesses is not permitted. This applies if one memory access is in Group A and one memory access is in Group B. That is, gathering is not permitted between a memory access in Group A and a memory access in Group B if the two accesses are separated by a barrier that affects at least one of the accesses.

Gathering between two memory accesses generated by a Load-Acquire/Store-Release is not permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

**Note**

- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  - The accesses are not separated by a `DMB` or `DSB` barrier that affects both of the accesses, for example if one access is in Group A and the other is in Group B.
  - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
  - The accesses are not generated by a Store-Release instruction.
- The ARM architecture only defines programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the Non-gathering attribute.

An implementation is not permitted to perform an access with the Non-gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Reordering

In the Device memory attribute:

**R** Indicates that the location has the Reordering attribute.

**nR** Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

**Note**

- The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the `DMB` instruction.
- The ARM architecture only defines programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.

An implementation is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.

An additional relaxation is that an implementation is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses with the non-Reordering attribute and accesses with the Reordering attribute.
- Accesses with the non-Reordering attribute and accesses to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

### Early Write Acknowledgement

In the Device memory attribute:

**E** Indicates that the location has the Early Write Acknowledgement attribute.

**nE** Indicates that the location has the No Early Write Acknowledgement attribute.

Early Write Acknowledgement is a hint to the platform memory system. Assigning the No Early Write Acknowledgement attribute to a Device memory location recommends that only the endpoint of the write access returns a write acknowledgement of the access, and that no earlier point in the memory system returns a write acknowledge. This means that a `DSB` barrier, executed by the PE that performed the write to the No Early Write Acknowledgement location, completes only after the write has reached its endpoint in the memory system. Typically, this endpoint is a peripheral or physical memory.

When the Early Write Acknowledgement attribute is assigned to a Device memory location, there is no such recommendation for the handling of accesses to that location.

**Note**

- The Early Write Acknowledgement hint has no effect on the ordering rules. The purpose of signaling no Early Write Acknowledgement is to signal to the interconnect that the peripheral requires the ability to signal the acknowledgement. The No Write Acknowledgement signal also provides an additional semantic that can be interpreted by the driver that is accessing the peripheral.
- This attribute is treated as a hint, as the exact nature of the interconnects attached to a PE is outside the scope of the ARM architecture definition, and not all interconnects provide a mechanism to ensure that a write has reached the physical endpoint of the memory system.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- ARM recommends that writes with the No Early Write Acknowledgement hint are used for PCIe configuration writes. However, the mechanisms by which PCIe configuration writes are identified are IMPLEMENTATION DEFINED.
- ARM strongly recommends that the Early Write Acknowledgement hint is not ignored by a PE, but is made available for use by the system.

---

Because the No Early Write Acknowledgement attribute is a hint:

- An implementation is permitted to perform an access with the Early Write Acknowledgement attribute in a manner consistent with the requirements specified by the No Early Write Acknowledgement attribute.
- An implementation is permitted to perform an access with the No Early Write Acknowledgement attribute in a manner consistent with the relaxations allowed by the Early Write Acknowledgement attribute.

### Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load and store instructions.

For all instructions that load or store one or more registers from the SIMD and floating-point register file there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load and store instructions.

For an `LDRD`, `STRD`, or `LDM` instruction with a register list that includes the PC, or an `STM` instruction with a register list that includes the PC, the order in which the registers are accessed is not defined by the architecture.

For a load or store of an Advanced SIMD element or structure, the order in which the registers are accessed is not defined by the architecture.

For a `VLDM`, `VSTM`, `LDM` and `STM` instruction with a register list that does not include the PC, all registers are accessed in ascending address order for Device accesses with the non-Reordering attribute.

## E2.8.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For accesses to any two bytes, *p* and *q*, that are generated by the same instruction:
  - The bytes *p* and *q* must have the same memory type and Shareability attributes. otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an `LDC`, `LDM`, `LDRD` `STC`, `STM` or `STRD` instruction, or an unaligned load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.
  - Except for possible differences in the cache allocation hints, ARM deprecates having different cacheability attributes for bytes *p* and *q*.

  For the permitted CONSTRAINED UNPREDICTABLE behavior, see *Crossing a page boundary with different memory types or Shareability attributes* on page K1-5431.
- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross a 4KB address boundary then behavior is CONSTRAINED UNPREDICTABLE and *Crossing a 4KB boundary with a Device access* on page K1-5432 describes the permitted behaviors.

  **Note**
  - The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.
  - This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses a 4KB address boundary.
  - ARM expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.

---


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E2.9 Mismatched memory attributes

In the ARMv8 architecture mismatched memory attributes are controlled by privileged software. For more information, see Chapter G4 *The AArch32 Virtual Memory System Architecture*.

Physical memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all of the following attributes of that location:

- Memory type, Device or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.

Collectively these are referred to as memory attributes.

**Note**

The terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory location is accessed with mismatched attributes the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory location might be lost. This means:
  - A read of the memory location by one agent might not return the value most recently written to that memory location by the same agent.
  - Multiple writes to the memory location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory location do not use consistent memory attributes, the exclusive monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of speculative read accesses.
- Prohibition on Gathering.
- Prohibition on Re-ordering.

For the following situations, when a physical memory location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. If the only memory type mismatch associated with a memory location across all users of the memory location is between different types of Device memory, then all accesses might take the properties of the weakest Device memory type.
2. Any agent that reads that memory location using the same common definition of the Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all of the following conditions are met:
   - All aliases to the memory location with write permission both use a common definition of the Shareability and Cacheability attributes for the memory location, and either:
     - Have the Inner Cacheability attribute the same as the Outer Cacheability attribute.
     - In the Non-secure PL1&0 translation regime, have HCR2.MIOCNCE set to 0.
   - All aliases to a memory location use a definition of the Shareability attributes that encompasses all the agents with permission to access the location.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

3. The possible software-visible effects caused by mismatched attributes for a memory location are defined more precisely if all of the mismatched attributes define the memory location as one of:
   - Any Device memory type.
   - Normal Inner Non-cacheable, Outer Non-cacheable memory.

   In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:
   - Possible loss of properties derived from the memory type when multiple agents attempt to access the memory location.
   - Possible reordering of memory transactions to the same memory location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting `DMB` barrier instructions between accesses to the same memory location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory location all assign the same Shareability attribute to the location, any loss of uniprocessor semantics, ordering, or coherency within a Shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable locations between agents in different shareability domains. This means:
   - Before writing to a location not using the Write-Back attribute, software must invalidate, or clean, a location from the caches if any agent might have written to the location with the Write-Back attribute. This avoids the possibility of overwriting the location with stale data.
   - After writing to a location with the Write-Back attribute, software must clean the location from the caches, to make the write visible to external memory.
   - Before reading the location with a cacheable attribute, software must invalidate the location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
   - Executing a `DMB` barrier instruction, with scope that applies to the common Shareability of the accesses, between any accesses to the same memory location that use different attributes.

   **Note**

   In AArch32 state, cache maintenance instructions can only be accessed from an Exception level that is higher than EL0, and therefore require a system call. For information on system calls, see *Exception-generating and exception-handling instructions* on page F1-2426. For information about the AArch32 cache maintenance instructions, see *AArch32 cache and branch predictor support* on page G3-3952.

   In all cases:
   - Location refers to any byte within the current coherency granule.
   - A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
   - In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see *Ordering of cache and branch predictor maintenance instructions* on page G3-3969.

   **Note**

   With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2. If the mismatched attributes for a location mean that multiple cacheable accesses to the location might be made with different Shareability attributes, then ordering and coherency are guaranteed only if:
   - Each PE that accesses the location with a cacheable attribute performs a clean and invalidate of the location before and after accessing that location.
   - A `DMB` barrier with scope that covers the full Shareability of the accesses is placed between any accesses to the same memory location that use different attributes.

   **Note**

   The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a location, and the accesses from the different agents have different memory attributes associated with the location, the exclusive monitor state becomes UNKNOWN.

ARM strongly recommends that software does not use mismatched attributes for aliases of the same location. An implementation might not optimize the performance of a system that uses mismatched aliases.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## E2.10 Synchronization and semaphores

ARMv8 provides non-blocking synchronization of shared memory, using *synchronization primitives*. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal and Device memory.

**Note**

Use of the ARMv8 synchronization primitives scales for multiprocessing system designs.

Table E2-6 shows the synchronization primitives and the associated CLREX instruction.

**Table E2-6 Synchronization primitives and associated instruction**

| Function | A32/T32 Instruction |
|---|---|
| Load-Exclusive | |
| Byte | LDREXB, LDAEXB |
| Halfword | LDREXH, LDAEXH |
| Word | LDREX, LDAEX |
| Doubleword | LDREXD. LDAEXD |
| Store-Exclusive | |
| Byte | STREXB, STLEXB |
| Halfword | STREXH, STLEXH |
| Word | STREX, STLEX |
| Doubleword | STREXD, STLEXD |
| Clear-Exclusive | CLREX |

The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address *x* is:

- The Load-Exclusive instruction reads a value from memory address *x*.
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address *x* only if no other observer, process, or thread has performed a more recent store to address *x*. The Store-Exclusive instruction returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see *Marking and the size of the marked memory block* on page E2-2401. A Store-Exclusive instruction to any address in the marked block clears the marking.

**Note**

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- *Exclusive access instructions and Non-shareable memory locations* on page E2-2396.
- *Exclusive access instructions and shareable memory locations* on page E2-2397.
- *Marking and the size of the marked memory block* on page E2-2401.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Context switch support* on page E2-2401.
- *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page E2-2402.
- *Use of WFE and SEV instructions by spin-locks* on page E2-2403.

### E2.10.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the Shareability attribute is Non-shareable, the exclusive access instructions rely on a *local monitor* that marks any address from which the PE executes a Load-Exclusive instruction. Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

**If the local monitor is in the Exclusive Access state**

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - If the store took place the status value is 0.
  - Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

**If the local monitor is in the Open Access state**

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

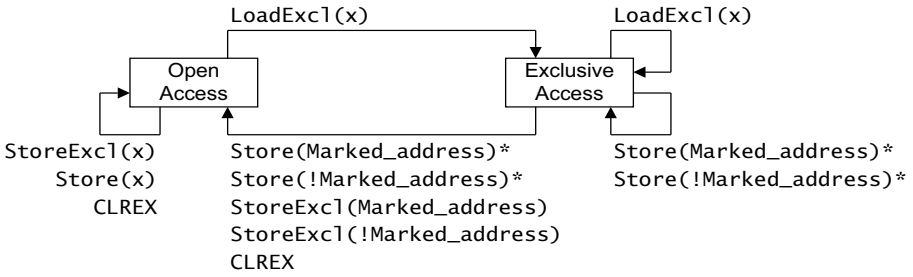
When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a physical address that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a physical address that is marked as Exclusive Access by its local monitor it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked physical address causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the physical address to be marked.

Figure E2-4 on page E2-2397 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address x used for the operation.

**Figure E2-4 Local monitor state machine diagram**

For more information about marking see *Marking and the size of the marked memory block* on page E2-2401.

**Note**

For the local monitor state machine, as shown in Figure E2-4:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any physical address, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction, by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

### Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure E2-4.

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure E2-4.
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure E2-4 must not indefinitely delay forward progress of execution.

## E2.10.2 Exclusive access instructions and shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For shareable memory locations, exclusive access instructions rely on:

- A *local monitor* for each PE in the system, that marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in *Exclusive access instructions and Non-shareable memory locations* on page E2-2396, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:
  - Updating memory.
  - Returning a status value of 0.

  The local monitor can ignore accesses from other PEs in the system.
- A *global monitor* that marks a physical address as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the Shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:
  - Can hold at least one marked block.
  - Maintains a state machine for each marked block it can hold.

  **Note**

  For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is UNPREDICTABLE, see *Load-Exclusive and Store-Exclusive instruction usage restrictions* on page E2-2402.

**Note**

The global monitor can either reside in a block that is part of the hardware on which the PE executes or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

**Note**

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport's Bakery algorithm to achieve mutual exclusion.

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global exclusive monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hint and Write allocation hint and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hint and Write allocation hints and not transient.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The set of memory types that support atomic instructions must include all of the memory types for which a global monitor is implemented.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive/Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an external abort.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Fault Status code of:
  - DFSR.STATUS = `0b110101` when using the Long-descriptor translation table format. The fault can also be reported in the HSR.ISS[5:0] field for exceptions to Hyp mode.
  - DFSR.FS = `0b10101` when using the Short-descriptor translation table format.

  If the IMPLEMENTATION DEFINED MMU fault is generated for the Non-secure PL1&0 translation regime then:
  - If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.
  - Otherwise, the fault is a second stage fault and the exception is taken to EL2.

  The priority of this fault is IMPLEMENTATION DEFINED.
- The instruction is treated as a `NOP`.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by an ARM PE is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

### Operation of the global monitor

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the physical address of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other physical address that has been marked by the requesting PE.

**Note**

The global monitor only supports a single outstanding exclusive access to shareable memory for each PE.

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the physical address accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:
  - A status value of 0 is returned to a register to acknowledge the successful store.
  - The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.
  - If the address accessed is marked for exclusive access in the global monitor state machine for any other PE then that state machine transitions to Open Access state.
- If no address is marked as exclusive access for the requesting PE, the store does not succeed:
  - A status value of 1 is returned to a register to indicate that the store failed.
  - The global monitor is not affected and remains in Open Access state for the requesting PE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If a different physical address is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  - If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.
  - If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means it responds to:

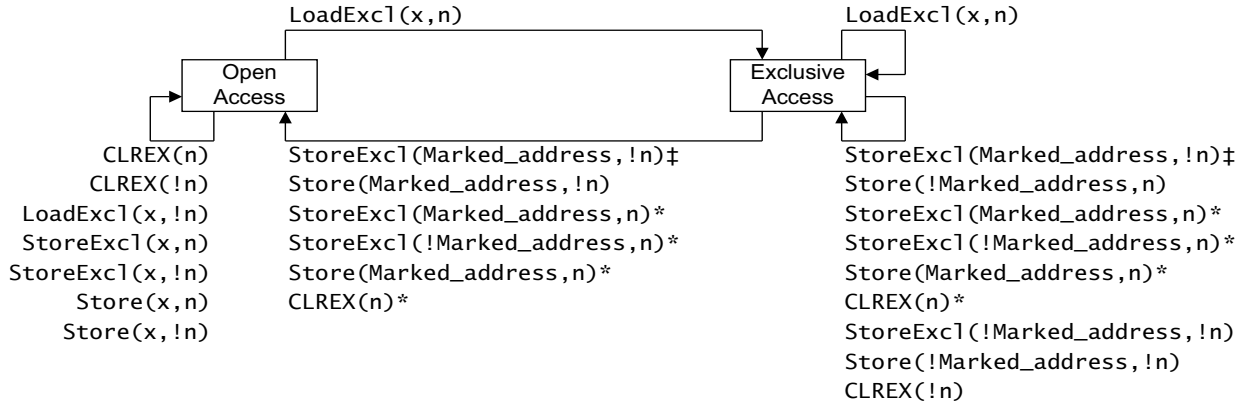
- Accesses generated by PE(n).
- Accesses generated by the other observers in the Shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

### *Clear global monitor event*

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see *Wait For Event and Send Event* on page G1-3834.

Figure E2-5 shows the state machine for PE(n) in a global monitor.




‡StoreExcl(Marked_address,!n) clears the monitor only if the StoreExcl updates memory
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any other store instruction.

Any LoadExcl operation updates the marked address to the most significant bits of the address x used for the operation.

**Figure E2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system**

For more information about marking see *Marking and the size of the marked memory block* on page E2-2401.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Note**

For the global monitor state machine, as shown in Figure E2-5 on page E2-2400:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.
- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure E2-5 on page E2-2400 only shows how the operations by (!n) cause state transitions of the state machine for PE(n).
- A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
- When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a `CLREX` instruction causes the global monitor to transition from Exclusive Access to Open Access state.
- It is IMPLEMENTATION DEFINED:
  - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

### E2.10.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size $2^a$ bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the *Exclusives reservation granule*. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4 - 512 words.

**Note**

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where $a$ is 4.
- 512 words in an implementation where $a$ is 11.

For example, in an implementation where $a$ is 4, a successful `LDREXB` of address `0x341B4` defines a marked block using bits[47:4] of the address. This means that the four words of memory from `0x341B0` to `0x341BF` are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

### E2.10.4 Context switch support

An exception return clears the local monitor. As a result, performing a `CLREX` instruction as part of a context switch is not required in most situations.

**Note**

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### E2.10.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a `LDREX`/`STREX` pair or a `LDREXB`/`STREXB` pair. To support different implementations of these functions, software must follow the notes and restrictions given in this subsection.

The following notes describe use of a Load-Exclusive/ Store-Exclusive pair, `LoadExcl`/`StoreExcl`, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table E2-6 on page E2-2395:

- The exclusives support a single outstanding exclusive access for each software thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target virtual address of a `StoreExcl` is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution, behavior can be UNPREDICTABLE. As a result, a `LoadExcl`/`StoreExcl` pair can only be relied upon to eventually succeed if the `LoadExcl` and the `StoreExcl` are executed with the same virtual address.
- If two `StoreExcl` instructions are executed without an intervening `LoadExcl` instruction the second `StoreExcl` instruction returns a status value of 1. This means that:
  — ARM recommends that, in a given thread of execution, every `StoreExcl` instruction has a preceding `LoadExcl` instruction associated with it.

  It is not necessary for every `LoadExcl` instruction to have a subsequent `StoreExcl` instruction.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a Store-Exclusive instruction is the same as the transaction size of the preceding Load-Exclusive instruction executed in that thread. If the transaction size of a Store-Exclusive instruction is different from the preceding Load-Exclusive instruction in the same thread of execution, behavior can be UNPREDICTABLE. As a result, software can rely on an `LoadExcl`/`StoreExcl` pair to eventually succeed only if they have the same size.
- An implementation might clear an exclusive monitor between the `LoadExcl` instruction and the `StoreExcl`, instruction without any application-related cause. For example, this might happen because of cache evictions. Software must, in any single thread of execution, avoid having any explicit memory accesses, System register updates, or cache maintenance instructions between the `LoadExcl` instruction and the associated `StoreExcl` instruction.
- Implementations can benefit from keeping the `LoadExcl` and `StoreExcl` operations close together in a single thread of execution. This minimizes the likelihood of the exclusive monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, ARM strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.
- The architecture sets an upper limit of 2048 bytes on the exclusive reservation granule that can be marked as exclusive. For performance reasons, ARM recommends that objects that are accessed by exclusive accesses are separated by the size of the exclusive reservations granule. This is a performance guideline rather than a functional requirement.
- After taking a Data Abort exception, the state of the exclusive monitors is UNKNOWN.
- For the memory location accessed by a `LoadExcl`/`StoreExcl` pair, if the memory attributes for the `LoadExcl` instruction differ from the memory attributes for the `StoreExcl` instruction, behavior is UNPREDICTABLE.

  This can occur either:

  — Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.

  — As a result of using different virtual addresses, with different attributes, that point to the same physical address. This case is covered by another bullet point in this list.
- The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global exclusive monitor that is in the Exclusive Access state is UNPREDICTABLE. The instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same Shareability domain as the PE executing the cache maintenance instruction, as determined by the Shareability domain of the address being maintained.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

ARM strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in the Non-secure state cannot cause a denial of service on a PE in the Secure state.

- If the mapping of the virtual to physical address is changed between the LDREX instruction and the STREX instruction, and the change is performed using a break-before-make sequence as described in *Using break-before-make when updating translation table entries* on page G4-4062, if the STREX is performed after another write to the same physical address as the STREX, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the STREX will not pass its monitor check.

  **Note**

  ARM expects that, in many implementations, either:

  — The TLB invalidation will clear either the local or global monitor.

  — The physical address will be checked between the LDREX and STREX.

**Note**

In the event of repeatedly-contending Load-Exclusive/Store-Exclusive instruction sequences from multiple PEs, an implementation must ensure that forward progress is made by at least one PE.

### E2.10.6 Use of WFE and SEV instructions by spin-locks

ARMv8 provides Wait For Event, Send Event, and Send Event Local instructions, WFE, SEV, SEVL, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in *Wait For Event and Send Event* on page G1-3834. However, in ARMv8, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

**Note**

This is equivalent to issuing an SEV instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an SEV instruction after clearing a spinlock.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Part F
## The AArch32 Instruction Sets

# Chapter F1
# The AArch32 Instruction Sets Overview

This chapter describes the T32 and A32 instruction sets. It contains the following sections:


- *Support for instructions in different versions of the ARM architecture* on page F1-2408.
- *Unified Assembler Language* on page F1-2409.
- *Branch instructions* on page F1-2411.
- *Data-processing instructions* on page F1-2412.
- *PSTATE and banked register access instructions* on page F1-2420.
- *Load/store instructions* on page F1-2421.
- *Load/store multiple instructions* on page F1-2424.
- *Miscellaneous instructions* on page F1-2425.
- *Exception-generating and exception-handling instructions* on page F1-2426.
- *Coprocessor instructions* on page F1-2427.
- *Advanced SIMD and floating-point load/store instructions* on page F1-2428.
- *Advanced SIMD and floating-point register transfer instructions* on page F1-2430.
- *Advanced SIMD data-processing instructions* on page F1-2431.
- *Floating-point data-processing instructions* on page F1-2439.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F1.1 Support for instructions in different versions of the ARM architecture

This manual describes the ARM AArch32 instruction sets, T32 and A32, for the ARMv8 architecture. Therefore, it indicates how any options or extensions in the ARMv8 architecture affect the available instructions. Also, Appendix K5 *ARMv8 Changes to the T32 and A32 Instruction Sets* provides information for those migrating from earlier versions of the architecture.

This manual does not provide any information about which T32 and A32 instructions were supported in specific earlier versions of the architecture. For this information, see the *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.2 Unified Assembler Language

This manual uses the ARM *Unified Assembler Language* (UAL). This assembly language syntax provides a canonical form for all T32 and A32 instructions.

UAL describes the syntax for the mnemonic and the operands of each instruction. In addition, it assumes that instructions and data items can be given labels. It does not specify the syntax to be used for labels, nor what assembler directives and options are available. See your assembler documentation for these details.

Most earlier ARM assembly language mnemonics are still supported as synonyms, as described in the instruction details.

**Note**

Most earlier T32 assembly language mnemonics are *not* supported.

UAL includes *instruction selection* rules that specify which instruction encoding is selected when more than one can provide the required functionality. For example, both 16-bit and 32-bit encodings exist for an `ADD R0, R1, R2` instruction. The most common instruction selection rule is that when both a 16-bit encoding and a 32-bit encoding are available, the 16-bit encoding is selected, to optimize code density.

Syntax options exist to override the normal instruction selection rules and ensure that a particular encoding is selected. These are useful when disassembling code, to ensure that subsequent assembly produces the original code, and in some other situations.

### F1.2.1 Conditional instructions

For maximum portability of UAL assembly language between the T32 and A32 instruction sets, ARM recommends that:

- `IT` instructions are written before conditional instructions in the correct way for the T32 instruction set.
- When assembling to the A32 instruction set, assemblers check that any `IT` instructions are correct, but do not generate any code for them.

Although other T32 instructions are unconditional, all instructions that are made conditional by an `IT` instruction must be written with a condition. These conditions must match the conditions imposed by the `IT` instruction. For example, an `ITTEE EQ` instruction imposes the `EQ` condition on the first two following instructions, and the `NE` condition on the next two. Those four instructions must be written with `EQ`, `EQ`, `NE` and `NE` conditions respectively.

Some instructions cannot be made conditional by an `IT` instruction. Some instructions can be conditional if they are the last instruction in the IT block, but not otherwise.

The branch instruction encodings that include a condition code field cannot be made conditional by an `IT` instruction. If the assembler syntax indicates a conditional branch that correctly matches a preceding `IT` instruction, it is assembled using a branch instruction encoding that does not include a condition code field.

### F1.2.2 Use of labels in UAL instruction syntax

The UAL syntax for some instructions includes the label of an instruction or a literal data item that is at a fixed offset from the instruction being specified. The assembler must:

1. Calculate the `PC` or `Align(PC, 4)` value of the instruction. The `PC` value of an instruction is its address plus 4 for a T32 instruction, or plus 8 for an A32 instruction. The `Align(PC, 4)` value of an instruction is its `PC` value ANDed with `0xFFFFFFFC` to force it to be word-aligned. There is no difference between the `PC` and `Align(PC, 4)` values for an A32 instruction, but there can be for a T32 instruction.
2. Calculate the offset from the `PC` or `Align(PC, 4)` value of the instruction to the address of the labeled instruction or literal data item.
3. Assemble a *PC-relative* encoding of the instruction, that is, one that reads its `PC` or `Align(PC, 4)` value and adds the calculated offset to form the required address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

For instructions that can encode a subtraction operation, if the instruction cannot encode the calculated offset but can encode minus the calculated offset, the instruction encoding specifies a subtraction of minus the calculated offset.

The syntax of the following instructions includes a label:

- `B`, `BL`, and `BLX` (immediate). The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a sign-extended immediate offset that is added to the `PC` value of the instruction to form the target address of the branch.
- `CBNZ` and `CBZ`. The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a zero-extended immediate offset that is added to the `PC` value of the instruction to form the target address of the branch. They do not support backward branches.
- `LDC`, `LDC2`, `LDR`, `LDRB`, `LDRD`, `LDRH`, `LDRSB`, `LDRSH`, `PLD`, `PLDW`, `PLI`, and `VLDR`. The normal assembler syntax of these load instructions can specify the label of a literal data item that is to be loaded. The encodings of these instructions specify a zero-extended immediate offset that is either added to or subtracted from the `Align(PC, 4)` value of the instruction to form the address of the data item. A few such encodings perform a fixed addition or a fixed subtraction and must only be used when that operation is required, but most contain a bit that specifies whether the offset is to be added or subtracted.

  When the assembler calculates an offset of 0 for the normal syntax of these instructions, it must assemble an encoding that adds 0 to the `Align(PC, 4)` value of the instruction. Encodings that subtract 0 from the `Align(PC, 4)` value cannot be specified by the normal syntax.

  There is an alternative syntax for these instructions that specifies the addition or subtraction and the immediate offset explicitly. In this syntax, the label is replaced by `[PC, #+/-<imm>]`, where:

  `+/-` Is `+` or omitted to specify that the immediate offset is to be added to the `Align(PC, 4)` value, or `-` if it is to be subtracted.

  `<imm>` Is the immediate offset.

  This alternative syntax makes it possible to assemble the encodings that subtract 0 from the `Align(PC, 4)` value, and to disassemble them to a syntax that can be re-assembled correctly.
- `ADR`. The normal assembler syntax for this instruction can specify the label of an instruction or literal data item whose address is to be calculated. Its encoding specifies a zero-extended immediate offset that is either added to or subtracted from the `Align(PC, 4)` value of the instruction to form the address of the data item, and some opcode bits that determine whether it is an addition or subtraction.

  When the assembler calculates an offset of 0 for the normal syntax of this instruction, it must assemble the encoding that adds 0 to the `Align(PC, 4)` value of the instruction. The encoding that subtracts 0 from the `Align(PC, 4)` value cannot be specified by the normal syntax.

  There is an alternative syntax for this instruction that specifies the addition or subtraction and the immediate value explicitly, by writing them as additions `ADD <Rd>, PC, #<imm>` or subtractions `SUB <Rd>, PC, #<imm>`. This alternative syntax makes it possible to assemble the encoding that subtracts 0 from the `Align(PC, 4)` value, and to disassemble it to a syntax that can be re-assembled correctly.

**Note**

ARM recommends that where possible, software avoids using:

- The alternative syntax for the `ADR`, `LDC`, `LDC2`, `LDR`, `LDRB`, `LDRD`, `LDRH`, `LDRSB`, `LDRSH`, `PLD`, `PLI`, `PLDW`, and `VLDR` instructions.
- The encodings of these instructions that subtract 0 from the `Align(PC, 4)` value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.3 Branch instructions

Table F1-1 summarizes the branch instructions in the T32 and A32 instruction sets. In addition to providing for changes in the flow of execution, some branch instructions can change instruction set.

**Table F1-1 Branch instructions**

| Instruction | See | Range, T32 | Range, A32 |
|---|---|---|---|
| Branch to target address | *B* on page F6-2623 | ±16MB | ±32MB |
| Compare and Branch on Nonzero,<br>Compare and Branch on Zero | *CBNZ, CBZ* on page F6-2646 | 0-126 bytes | [a] |
| Call a subroutine<br>Call a subroutine, change instruction set[b] | *BL, BLX (immediate)* on page F6-2639 | ±16MB<br>±16MB | ±32MB<br>±32MB |
| Call a subroutine, optionally change instruction set | *BLX (register)* on page F6-2641 | Any | Any |
| Branch to target address, change instruction set | *BX* on page F6-2643 | Any | Any |
| Change to Jazelle state | *BXJ* on page F6-2645 | - | - |
| Table Branch (byte offsets)<br>Table Branch (halfword offsets) | *TBB, TBH* on page F6-3117 | 0-510 bytes<br>0-131070 bytes | [a] |

a. These instructions do not exist in the A32 instruction set.

b. The range is determined by the instruction set of the `BLX` instruction, not of the instruction it branches to.

Branches to loaded and calculated addresses can be performed by `LDR`, `LDM` and data-processing instructions. For details see *Load/store instructions* on page F1-2421, *Load/store multiple instructions* on page F1-2424, *Standard data-processing instructions* on page F1-2412, and *Shift instructions* on page F1-2414.

In addition to the branch instructions shown in Table F1-1:

- In the A32 instruction set, a data-processing instruction that targets the PC behaves as a branch instruction. For more information, see *Data-processing instructions* on page F1-2412.
- In the T32 and A32 instruction sets, a load instruction that targets the PC behaves as a branch instruction. For more information, see *Load/store instructions* on page F1-2421.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.4 Data-processing instructions

Core data-processing instructions belong to one of the following groups:

- *Standard data-processing instructions*.

  These instructions perform basic data-processing operations, and share a common format with some variations.
- *Shift instructions* on page F1-2414.
- *Multiply instructions* on page F1-2414.
- *Saturating instructions* on page F1-2416.
- *Saturating addition and subtraction instructions* on page F1-2416.
- *Packing and unpacking instructions* on page F1-2417.
- *Parallel addition and subtraction instructions* on page F1-2418.
- *Divide instructions* on page F1-2419.
- *Miscellaneous data-processing instructions* on page F1-2419.

For related Advanced SIMD and floating-point instructions see *Advanced SIMD data-processing instructions* on page F1-2431 and *Floating-point data-processing instructions* on page F1-2439.

### F1.4.1 Standard data-processing instructions

These instructions generally have a destination register Rd, a first operand register Rn, and a second operand. The second operand can be another register Rm, or an immediate constant.

If the second operand is an immediate constant, it can be:

- Encoded directly in the instruction.
- A *modified immediate constant* that uses 12 bits of the instruction to encode a range of constants. T32 and A32 instructions have slightly different ranges of modified immediate constants. For more information, see *Modified immediate constants in T32 instructions* on page F2-2459 and *Modified immediate constants in A32 instructions* on page F2-2456.

If the second operand is another register, it can optionally be shifted in any of the following ways:

`LSL` Logical Shift Left by 1-31 bits.

`LSR` Logical Shift Right by 1-32 bits.

`ASR` Arithmetic Shift Right by 1-32 bits.

`ROR` Rotate Right by 1-31 bits.

`RRX` Rotate Right with Extend. For details see *Shift and rotate operations* on page E1-2312.

In T32 code, the amount to shift by is always a constant encoded in the instruction. In A32 code, the amount to shift by is either a constant encoded in the instruction, or the value of a register, Rs.

For instructions other than `CMN`, `CMP`, `TEQ`, and `TST`, the result of the data-processing operation is placed in the destination register. In the A32 instruction set, the destination register can be the PC, causing the result to be treated as a branch address. In the T32 instruction set, this is only permitted for some 16-bit forms of the `ADD` and `MOV` instructions.

These instructions can optionally set the condition flags, according to the result of the operation. If they do not set the flags, existing flag settings from a previous instruction are preserved.

Table F1-2 on page F1-2413 summarizes the main data-processing instructions in the T32 and A32 instruction sets. Generally, each of these instructions is described in three sections in Chapter F2 *About the T32 and A32 Instruction Descriptions*, one section for each of the following:

- `INSTRUCTION` (immediate) where the second operand is a modified immediate constant.
- `INSTRUCTION` (register) where the second operand is a register, or a register shifted by a constant.
- `INSTRUCTION` (register-shifted register) where the second operand is a register shifted by a value obtained from another register. These are only available in the A32 instruction set.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F1-2 Standard data-processing instructions**

| Instruction | Mnemonic | Notes |
|---|---|---|
| Add with Carry | ADC | - |
| Add | ADD | T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant. |
| Form PC-relative Address | ADR | First operand is the PC. Second operand is an immediate constant. T32 instruction set uses a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction. |
| Bitwise AND | AND | - |
| Bitwise Bit Clear | BIC | - |
| Compare Negative | CMN | Sets flags. Like ADD but with no destination register. |
| Compare | CMP | Sets flags. Like SUB but with no destination register. |
| Bitwise Exclusive OR | EOR | - |
| Copy operand to destination | MOV | Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. For details see *Shift instructions* on page F1-2414.<br>The T32 and A32 instruction sets permit use of a modified immediate constant or a zero-extended 16-bit immediate constant. |
| Bitwise NOT | MVN | Has only one operand, with the same options as the second operand in most of these instructions. |
| Bitwise OR NOT | ORN | Not available in the A32 instruction set. |
| Bitwise OR | ORR | - |
| Reverse Subtract | RSB | Subtracts first operand from second operand. This permits subtraction from constants and shifted registers. |
| Reverse Subtract with Carry | RSC | Not available in the T32 instruction set. |
| Subtract with Carry | SBC | - |
| Subtract | SUB | T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant. |
| Test Equivalence | TEQ | Sets flags. Like EOR but with no destination register. |
| Test | TST | Sets flags. Like AND but with no destination register. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F1.4.2 Shift instructions

Table F1-3 lists the shift instructions in the T32 and A32 instruction sets.

**Table F1-3 Shift instructions**

| Instruction | See |
|---|---|
| Arithmetic Shift Right | *ASR (immediate)* on page F6-2615<br>*ASR (register)* on page F6-2617<br>*ASRS (immediate)* on page F6-2619<br>*ASRS (register)* on page F6-2621 |
| Logical Shift Left | *LSL (immediate)* on page F6-2785<br>*LSL (register)* on page F6-2787<br>*LSLS (immediate)* on page F6-2789<br>*LSLS (register)* on page F6-2791 |
| Logical Shift Right | *LSR (immediate)* on page F6-2793<br>*LSR (register)* on page F6-2795<br>*LSRS (immediate)* on page F6-2797<br>*LSRS (register)* on page F6-2799 |
| Rotate Right | *ROR (immediate)* on page F6-2917<br>*ROR (register)* on page F6-2919<br>*RORS (immediate)* on page F6-2921<br>*RORS (register)* on page F6-2923 |
| Rotate Right with Extend | *RRX* on page F6-2925<br>*RRXS* on page F6-2927 |

In the A32 instruction set only, the destination register of these instructions can be the PC, causing the result to be treated as an address to branch to.

### F1.4.3 Multiply instructions

These instructions can operate on signed or unsigned quantities. In some types of operation, the results are the same whether the operands are signed or unsigned.

- Table F1-4 summarizes the multiply instructions where there is no distinction between signed and unsigned quantities.

  The least significant 32 bits of the result are used. More significant bits are discarded.
- Table F1-5 on page F1-2415 summarizes the signed multiply instructions.
- Table F1-6 on page F1-2415 summarizes the unsigned multiply instructions.

**Table F1-4 General multiply instructions**

| Instruction | See | Operation (number of bits) |
|---|---|---|
| Multiply Accumulate | *MLA, MLAS* on page F6-2806 | 32 = 32 + 32 × 32 |
| Multiply and Subtract | *MLS* on page F6-2808 | 32 = 32 – 32 × 32 |
| Multiply | *MUL, MULS* on page F6-2842 | 32 = 32 × 32 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F1-5 Signed multiply instructions**

| Instruction | See | Operation (number of bits) |
|---|---|---|
| Signed Multiply Accumulate (halfwords) | *SMLABB, SMLABT, SMLATB, SMLATT* on page F6-2981 | 32 = 32 + 16 × 16 |
| Signed Multiply Accumulate Dual | *SMLAD, SMLADX* on page F6-2983 | 32 = 32 + 16 × 16 + 16 × 16 |
| Signed Multiply Accumulate Long | *SMLAL, SMLALS* on page F6-2985 | 64 = 64 + 32 × 32 |
| Signed Multiply Accumulate Long (halfwords) | *SMLALBB, SMLALBT, SMLALTB, SMLALTT* on page F6-2987 | 64 = 64 + 16 × 16 |
| Signed Multiply Accumulate Long Dual | *SMLALD, SMLALDX* on page F6-2989 | 64 = 64 + 16 × 16 + 16 × 16 |
| Signed Multiply Accumulate (word by halfword) | *SMLAWB, SMLAWT* on page F6-2991 | 32 = 32 + 32 × 16 [a] |
| Signed Multiply Subtract Dual | *SMLSD, SMLSDX* on page F6-2993 | 32 = 32 + 16 × 16 – 16 × 16 |
| Signed Multiply Subtract Long Dual | *SMLSLD, SMLSLDX* on page F6-2995 | 64 = 64 + 16 × 16 – 16 × 16 |
| Signed Most Significant Word Multiply Accumulate | *SMMLA, SMMLAR* on page F6-2997 | 32 = 32 + 32 × 32 [b] |
| Signed Most Significant Word Multiply Subtract | *SMMLS, SMMLSR* on page F6-2999 | 32 = 32 – 32 × 32 [b] |
| Signed Most Significant Word Multiply | *SMMUL, SMMULR* on page F6-3001 | 32 = 32 × 32 [b] |
| Signed Dual Multiply Add | *SMUAD, SMUADX* on page F6-3003 | 32 = 16 × 16 + 16 × 16 |
| Signed Multiply (halfwords) | *SMULBB, SMULBT, SMULTB, SMULTT* on page F6-3005 | 32 = 16 × 16 |
| Signed Multiply Long | *SMULL, SMULLS* on page F6-3007 | 64 = 32 × 32 |
| Signed Multiply (word by halfword) | *SMULWB, SMULWT* on page F6-3009 | 32 = 32 × 16 [a] |
| Signed Dual Multiply Subtract | *SMUSD, SMUSDX* on page F6-3011 | 32 = 16 × 16 – 16 × 16 |

a. The most significant 32 bits of the 48-bit product are used. Less significant bits are discarded.
b. The most significant 32 bits of the 64-bit product are used. Less significant bits are discarded.

**Table F1-6 Unsigned multiply instructions**

| Instruction | See | Operation (number of bits) |
|---|---|---|
| Unsigned Multiply Accumulate Accumulate Long | *UMAAL* on page F6-3152 | 64 = 32 + 32 + 32 × 32 |
| Unsigned Multiply Accumulate Long | *UMLAL, UMLALS* on page F6-3154 | 64 = 64 + 32 × 32 |
| Unsigned Multiply Long | *UMULL, UMULLS* on page F6-3156 | 64 = 32 × 32 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F1.4.4 Saturating instructions

Table F1-7 lists the saturating instructions in the T32 and A32 instruction sets. For more information, see *Pseudocode description of saturation* on page E1-2315.

**Table F1-7 Saturating instructions**

| Instruction | See | Operation |
|---|---|---|
| Signed Saturate | *SSAT* on page F6-3016 | Saturates optionally shifted 32-bit value to selected range |
| Signed Saturate 16 | *SSAT16* on page F6-3018 | Saturates two 16-bit values to selected range |
| Unsigned Saturate | *USAT* on page F6-3174 | Saturates optionally shifted 32-bit value to selected range |
| Unsigned Saturate 16 | *USAT16* on page F6-3176 | Saturates two 16-bit values to selected range |

### F1.4.5 Saturating addition and subtraction instructions

Table F1-8 lists the saturating addition and subtraction instructions in the T32 and A32 instruction sets. For more information, see *Pseudocode description of saturation* on page E1-2315.

**Table F1-8 Saturating addition and subtraction instructions**

| Instruction | See | Operation |
|---|---|---|
| Saturating Add | *QADD* on page F6-2887 | Add, saturating result to the 32-bit signed integer range |
| Saturating Subtract | *QSUB* on page F6-2901 | Subtract, saturating result to the 32-bit signed integer range |
| Saturating Double and Add | *QADD* on page F6-2887 | Doubles one value and adds a second value, saturating the doubling and the addition to the 32-bit signed integer range |
| Saturating Double and Subtract | *QDSUB* on page F6-2897 | Doubles one value and subtracts the result from a second value, saturating the doubling and the subtraction to the 32-bit signed integer range |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F1.4.6 Packing and unpacking instructions

Table F1-9 lists the packing and unpacking instructions in the T32 and A32 instruction sets.

**Table F1-9 Packing and unpacking instructions**

| Instruction | See | Operation |
|---|---|---|
| Pack Halfword | *PKHBT, PKHTB* on page F6-2864 | Combine halfwords |
| Signed Extend and Add Byte | *SXTAB* on page F6-3105 | Extend 8 bits to 32 and add |
| Signed Extend and Add Byte 16 | *SXTAB16* on page F6-3107 | Dual extend 8 bits to 16 and add |
| Signed Extend and Add Halfword | *SXTAH* on page F6-3109 | Extend 16 bits to 32 and add |
| Signed Extend Byte | *SXTB* on page F6-3111 | Extend 8 bits to 32 |
| Signed Extend Byte 16 | *SXTB16* on page F6-3113 | Dual extend 8 bits to 16 |
| Signed Extend Halfword | *SXTH* on page F6-3115 | Extend 16 bits to 32 |
| Unsigned Extend and Add Byte | *UXTAB* on page F6-3184 | Extend 8 bits to 32 and add |
| Unsigned Extend and Add Byte 16 | *UXTAB16* on page F6-3186 | Dual extend 8 bits to 16 and add |
| Unsigned Extend and Add Halfword | *UXTAH* on page F6-3188 | Extend 16 bits to 32 and add |
| Unsigned Extend Byte | *UXTB* on page F6-3190 | Extend 8 bits to 32 |
| Unsigned Extend Byte 16 | *UXTB16* on page F6-3192 | Dual extend 8 bits to 16 |
| Unsigned Extend Halfword | *UXTH* on page F6-3194 | Extend 16 bits to 32 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F1.4.7 Parallel addition and subtraction instructions

These instructions perform additions and subtractions on the values of two registers and write the result to a destination register, treating the register values as sets of two halfwords or four bytes. That is, they perform SIMD additions or subtractions on the general-purpose registers.

These instructions consist of a prefix followed by a main instruction mnemonic. The prefixes are as follows:

`S` Signed arithmetic modulo $2^8$ or $2^{16}$.

`Q` Signed saturating arithmetic.

`SH` Signed arithmetic, halving the results.

`U` Unsigned arithmetic modulo $2^8$ or $2^{16}$.

`UQ` Unsigned saturating arithmetic.

`UH` Unsigned arithmetic, halving the results.

The main instruction mnemonics are as follows:

`ADD16` Adds the top halfwords of two operands to form the top halfword of the result, and the bottom halfwords of the same two operands to form the bottom halfword of the result.

`ASX` Exchanges halfwords of the second operand, and then adds top halfwords and subtracts bottom halfwords.

`SAX` Exchanges halfwords of the second operand, and then subtracts top halfwords and adds bottom halfwords.

`SUB16` Subtracts each halfword of the second operand from the corresponding halfword of the first operand to form the corresponding halfword of the result.

`ADD8` Adds each byte of the second operand to the corresponding byte of the first operand to form the corresponding byte of the result.

`SUB8` Subtracts each byte of the second operand from the corresponding byte of the first operand to form the corresponding byte of the result.

The instruction set permits all 36 combinations of prefix and main instruction operand, as Table F1-10 shows.

See also *Advanced SIMD parallel addition and subtraction* on page F1-2432.

**Table F1-10 Parallel addition and subtraction instructions**

| Main instruction | Signed | Saturating | Signed halving | Unsigned | Unsigned saturating | Unsigned halving |
|---|---|---|---|---|---|---|
| `ADD16`, add, two halfwords | `SADD16` | `QADD16` | `SHADD16` | `UADD16` | `UQADD16` | `UHADD16` |
| `ASX`, add and subtract with exchange | `SASX` | `QASX` | `SHASX` | `UASX` | `UQASX` | `UHASX` |
| `SAX`, subtract and add with exchange | `SSAX` | `QSAX` | `SHSAX` | `USAX` | `UQSAX` | `UHSAX` |
| `SUB16`, subtract, two halfwords | `SSUB16` | `QSUB16` | `SHSUB16` | `USUB16` | `UQSUB16` | `UHSUB16` |
| `ADD8`, add, four bytes | `SADD8` | `QADD8` | `SHADD8` | `UADD8` | `UQADD8` | `UHADD8` |
| `SUB8`, subtract, four bytes | `SSUB8` | `QSUB8` | `SHSUB8` | `USUB8` | `UQSUB8` | `UHSUB8` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F1.4.8 Divide instructions

In ARMv8, signed and unsigned integer divide instructions are included in both the T32 instruction set and the A32 instruction set. For more information about their implementation in previous versions of the ARM architecture see the *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

For descriptions of the instructions see:

- *SDIV* on page F6-2958.
- *UDIV* on page F6-3138.

For the `SDIV` and `UDIV` instructions, divide-by-zero always returns a zero result.

The ID_ISAR0.Divide_instrs field indicates the level of support for these instructions. The field value of `0b0010` indicates they are implemented in both the T32 and A32 instruction sets.

### F1.4.9 Miscellaneous data-processing instructions

Table F1-11 lists the miscellaneous data-processing instructions in the T32 and A32 instruction sets. Immediate values in these instructions are simple binary numbers.

**Table F1-11 Miscellaneous data-processing instructions**

| Instruction | See | Notes |
|---|---|---|
| BitField Clear | *BFC* on page F6-2626 | - |
| BitField Insert | *BFI* on page F6-2628 | - |
| Count Leading Zeros | *CLZ* on page F6-2650 | - |
| Move Top | *MOVT* on page F6-2823 | Moves 16-bit immediate value to top halfword. Bottom halfword unchanged. |
| Reverse Bits | *RBIT* on page F6-2907 | - |
| Byte-Reverse Word | *REV* on page F6-2909 | - |
| Byte-Reverse Packed Halfword | *REV16* on page F6-2911 | - |
| Byte-Reverse Signed Halfword | *REVSH* on page F6-2913 | - |
| Signed BitField Extract | *SBFX* on page F6-2956 | - |
| Select Bytes using GE flags | *SEL* on page F6-2960 | - |
| Unsigned BitField Extract | *UBFX* on page F6-3134 | - |
| Unsigned Sum of Absolute Differences | *USAD8* on page F6-3170 | - |
| Unsigned Sum of Absolute Differences and Accumulate | *USADA8* on page F6-3172 | - |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.5 PSTATE and banked register access instructions

These instructions transfer PE state information to or from a general-purpose register.

### F1.5.1 PSTATE access instructions

PSTATE holds process state information, see *Process state, PSTATE* on page E1-2319. In AArch32 state:

- At EL1 or higher, PSTATE is accessible using the *Current Program Status Register* (CPSR).
- At EL0, a subset of the CPSR is accessible as the *Application Program Status Register* (APSR).
- On taking an exception, the contents of the CPSR are copied to the *Saved Program Status Register* (SPSR) of the mode from which the exception is taken.

The `MRS` and `MSR` instructions move the contents of the CPSR, APSR, or the SPSR of the current mode to or from a general-purpose register, see:

- *MRS* on page F6-2830.
- *MSR (immediate)* on page F6-2838.
- *MSR (register)* on page F6-2840.

When executed at EL0, `MRS` and `MSR` instructions can only access the APSR.

The PSTATE condition flags, PSTATE.{N, Z, C, V} are set by the execution of data-processing instructions, and can control the execution of conditional instructions. However, software can set the condition flags explicitly using the `MSR` instruction, and can read the current state of the condition flags explicitly using the `MRS` instruction.

In addition, at EL1 or higher, software can use the `CPS` instruction to change the PSTATE.M field and the PSTATE.{A, I, F} interrupt mask bits, see *CPS, CPSID, CPSIE* on page F6-2662.

### F1.5.2 Banked register access instructions

At EL1 or higher, the `MRS` (Banked register) and `MSR` (Banked register) instructions move the contents of a Banked general-purpose register, the SPSR, or the ELR_hyp, to or from a general-purpose register. See:

- *MRS (Banked register)* on page F6-2832.
- *MSR (Banked register)* on page F6-2835.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.6 Load/store instructions

Table F1-12 summarizes the general-purpose register load/store instructions in the T32 and A32 instruction sets. Some of these instructions can also operate on the PC. See also:

- *Load/store multiple instructions* on page F1-2424.
- *Synchronization and semaphores* on page E2-2395, for more information about the Load-Exclusive and Store-Exclusive instructions.
- *Load-Acquire, Store-Release* on page E2-2378, for more information about the Load-Acquire/Store-Release and Load-Acquire Exclusive/Store-Release Exclusive instructions.
- *Advanced SIMD and floating-point load/store instructions* on page F1-2428.

Load/store instructions have several options for addressing memory. For more information, see *Addressing modes* on page F1-2422.

**Table F1-12 Load/store instructions**

| Data type | Load | Store | Unprivileged | | Exclusive | | Load-Acquire | Store-Release | Exclusive | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Load | Store | Load | Store | | | Load-Acquire | Store-Release |
| 32-bit word | LDR | STR | LDRT | STRT | LDREX | STREX | LDA | STL | LDAEX | STLEX |
| 16-bit halfword | - | STRH | - | STRHT | - | STREXH | LDAH | STLH | LDAEXH | STLEXH |
| 16-bit unsigned halfword | LDRH | - | LDRHT | - | LDREXH | - | - | - | - | - |
| 16-bit signed halfword | LDRSH | - | LDRSHT | - | - | - | - | - | - | - |
| 8-bit byte | - | STRB | - | STRBT | - | STREXB | LDAB | STLB | LDAEXB | STLEXB |
| 8-bit unsigned byte | LDRB | - | LDRBT | - | LDREXB | - | - | - | - | - |
| 8-bit signed byte | LDRSB | - | LDRSBT | - | - | - | - | - | - | - |
| Two 32-bit words | LDRD | STRD | - | - | - | - | - | - | - | - |
| 64-bit doubleword | - | - | - | - | LDREXD | STREXD | - | - | LDAEXD | STLEXD |

### F1.6.1 Loads to the PC

The LDR instruction can load a value into the PC. The value loaded is treated as an interworking address, as described by the LoadWritePC() pseudocode function in *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

### F1.6.2 Halfword and byte loads and stores

Halfword and byte stores store the least significant halfword or byte from the register, to 16 or 8 bits of memory respectively. There is no distinction between signed and unsigned stores.

Halfword and byte loads load 16 or 8 bits from memory into the least significant halfword or byte of a register. Unsigned loads zero-extend the loaded value to 32 bits, and signed loads sign-extend the value to 32 bits.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F1.6.3 Load unprivileged and Store unprivileged

When executing at EL0, a Load unprivileged or Store unprivileged instruction operates in exactly the same way as the corresponding ordinary load or store instruction. For example, an LDRT instruction executes in exactly the same way as the equivalent LDR instruction. When executed at PL1, Load unprivileged and Store unprivileged instructions behave as they would if they were executed at EL0. For example, an LDRT instruction executes in exactly the way that the equivalent LDR instruction would execute at EL0. In particular, the instructions make unprivileged memory accesses.

**Note**

As described in *Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990, execution at PL1 describes all of the following:

- Execution at Non-secure EL1 using AArch32.
- Execution at Secure EL1 using AArch32 when EL3 is not implemented.
- Execution at Secure EL1 using AArch32 when EL3 is implemented and is using AArch64.
- Execution at Secure EL3 when EL3 is implemented and is using AArch32.

The Load unprivileged and Store unprivileged instructions are UNPREDICTABLE if executed at EL2.

For more information, see *Access permissions* on page G4-4037.

### F1.6.4 Load-Exclusive and Store-Exclusive

Load-Exclusive and Store-Exclusive instructions provide shared memory synchronization. For more information, see *Synchronization and semaphores* on page E2-2395.

### F1.6.5 Load-Acquire and Store-Release

Load-Acquire and Store-Release instructions provide memory barriers. Load-Acquire Exclusive and Store-Release Exclusive instructions provide memory barriers with shared memory synchronization. For more information, see *Load-Acquire, Store-Release* on page E2-2378.

### F1.6.6 Addressing modes

The address for a load or store is formed from two parts: a value from a base register, and an offset.

The base register can be any one of the general-purpose registers R0-R12, SP, or LR.

For loads, the base register can be the PC. This provides PC-relative addressing for position-independent code. Instructions marked (literal) in their title in Chapter F2 *About the T32 and A32 Instruction Descriptions* are PC-relative loads.

The offset takes one of three formats:

**Immediate** The offset is an unsigned number that can be added to or subtracted from the base register value. Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets and input/output registers.

**Register** The offset is a value from a general-purpose register. The value can be added to, or subtracted from, the base register value. Register offsets are useful for accessing arrays or blocks of data.

**Scaled register** The offset is a general-purpose register, shifted by an immediate value, then added to or subtracted from the base register. This means an array index can be scaled by the size of each array element.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The offset and base register can be used in three different ways to form the memory address. The addressing modes are described as follows:

**Offset** The offset is added to or subtracted from the base register to form the memory address.

**Pre-indexed** The offset is added to or subtracted from the base register to form the memory address. The base register is then updated with this new address, to permit automatic indexing through an array or memory block.

**Post-indexed** The value of the base register alone is used as the memory address. The offset is then added to or subtracted from the base register. The result is stored back in the base register, to permit automatic indexing through an array or memory block.

**Note**

Not every variant is available for every instruction, and the range of permitted immediate values and the options for scaled registers vary from instruction to instruction. See Chapter F2 *About the T32 and A32 Instruction Descriptions* for full details for each instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.7 Load/store multiple instructions

Load Multiple instructions load from memory a subset, or possibly all, of the general-purpose registers and the PC.

Store Multiple instructions store to memory a subset, or possibly all, of the general-purpose registers.

The memory locations are consecutive word-aligned words. The addresses used are obtained from a base register, and can be either above or below the value in the base register. The base register can optionally be updated by the total size of the data transferred.

Table F1-13 summarizes the load/store multiple instructions in the T32 and A32 instruction sets.

**Table F1-13 Load/store multiple instructions**

| Instruction | See |
|---|---|
| Load Multiple, Increment After or Full Descending | *LDM, LDMIA, LDMFD* on page F6-2711 |
| Load Multiple, Decrement After or Full Ascending [a] | *LDMDA, LDMFA* on page F6-2718 |
| Load Multiple, Decrement Before or Empty Ascending | *LDMDB, LDMEA* on page F6-2720 |
| Load Multiple, Increment Before or Empty Descending [a] | *LDMIB, LDMED* on page F6-2722 |
| Pop multiple registers off the stack [b] | *POP* on page F6-2877 |
| Push multiple registers onto the stack [c] | *PUSH* on page F6-2883 |
| Store Multiple, Increment After or Empty Ascending | *STM, STMIA, STMEA* on page F6-3041 |
| Store Multiple, Decrement After or Empty Descending [a] | *STMDA, STMED* on page F6-3045 |
| Store Multiple, Decrement Before or Full Descending | *STMDB, STMFD* on page F6-3047 |
| Store Multiple, Increment Before or Full Ascending [a] | *STMIB, STMFA* on page F6-3049 |

a. Not available in the T32 instruction set.
b. This instruction is equivalent to an `LDM` instruction with the SP as base register, and base register updating.
c. This instruction is equivalent to an `STMDB` instruction with the SP as base register, and base register updating.

When executing at EL1, variants of the `LDM` and `STM` instructions load and store User mode registers. Another system level variant of the `LDM` instruction performs an exception return.

### F1.7.1 Loads to the PC

The `LDM`, `LDMDA`, `LDMDB`, `LDMIB`, and `POP` instructions can load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.8 Miscellaneous instructions

Table F1-14 summarizes the miscellaneous instructions in the T32 and A32 instruction sets.

**Table F1-14 Miscellaneous instructions**

| Instruction | See |
|---|---|
| Clear-Exclusive | *CLREX* on page F6-2649 |
| Debug Hint | *DBG* on page F6-2669 |
| Data Memory Barrier | *DMB* on page F6-2671 |
| Data Synchronization Barrier | *DSB* on page F6-2673 |
| Instruction Synchronization Barrier | *ISB* on page F6-2688 |
| If-Then | *IT* on page F6-2690 |
| No Operation | *NOP* on page F6-2851 |
| Preload Data | *PLD, PLDW (immediate)* on page F6-2866<br>*PLD (literal)* on page F6-2868<br>*PLD, PLDW (register)* on page F6-2870 |
| Preload Instruction | *PLI (immediate, literal)* on page F6-2872<br>*PLI (register)* on page F6-2875 |
| Set Endianness | *SETEND* on page F6-2962 |
| Send Event | *SEV* on page F6-2963 |
| Send Event Local | *SEVL* on page F6-2965 |
| Wait For Event | *WFE* on page F6-3196 |
| Wait For Interrupt | *WFI* on page F6-3198 |
| Yield | *YIELD* on page F6-3200 |

### F1.8.1 The Yield instruction

In a *Symmetric Multi-Threading* (SMT) design, a thread can use the `YIELD` instruction to give a hint to the PE that it is running on. The `YIELD` hint indicates that whatever the thread is currently doing is of low importance, and so could yield. For example, the thread might be sitting in a spin-lock. A similar use might be in modifying the arbitration priority of the snoop bus in a multiprocessor (MP) system. Defining such an instruction permits binary compatibility between SMT and SMP systems.

AArch32 state defines a `YIELD` instruction as a specific NOP (No Operation) hint instruction.

The `YIELD` instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use on migration to a multiprocessor or multithreading system. Operating systems can use `YIELD` in places where a yield hint is wanted, knowing that it will be treated as a `NOP` if there is no implementation benefit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.9 Exception-generating and exception-handling instructions

The following instructions are intended specifically to cause a synchronous exception to occur:

- The SVC instruction generates a Supervisor Call exception. For more information, see *Supervisor Call (SVC) exception* on page G1-3815.
- The Breakpoint instruction BKPT provides software breakpoints. For more information, see *Software Breakpoint Instruction exceptions* on page G2-3895.
- In an implementation that includes EL3, when executing at EL1 or higher, the SMC instruction generates a Secure Monitor Call exception. For more information, see *Secure Monitor Call (SMC) exception* on page G1-3816.
- In an implementation that includes EL2, in software executing in a Non-secure EL1 mode, the HVC instruction generates a Hypervisor Call exception. For more information, see *Hypervisor Call (HVC) exception* on page G1-3817.

For an exception taken to an EL1 mode:

- The system level variants of the SUBS and LDM instructions can perform a return from an exception.

  **Note**

  The variants of SUBS include MOVS. See the references to SUBS PC, LR in Table F1-15 for more information.

- The SRS instruction can be used near the start of the handler, to store return information. The RFE instruction can then perform a return from the exception using the stored return information.

In an implementation that includes EL2, the ERET instruction performs a return from an exception taken to Hyp mode.

For more information, see *Exception return to an Exception level using AArch32* on page G1-3796.

Table F1-15 summarizes the instructions, in the T32 and A32 instruction sets, for generating or handling an exception. Except for BKPT and SVC, these are system level instructions.

**Table F1-15 Exception-generating and exception-handling instructions**

| Instruction | See |
|---|---|
| Supervisor Call | *SVC* on page F6-3103 |
| Breakpoint | *BKPT* on page F6-2637 |
| Secure Monitor Call | *SMC* on page F6-2979 |
| Return From Exception | *RFE, RFEDA, RFEDB, RFEIA, RFEIB* on page F6-2915 |
| Subtract (exception return)[a] | *SUB, SUBS (immediate)* on page F6-3088[a] |
| Move (exception return)[a] | *MOV, MOVS (register)* on page F6-2813[a] |
| Hypervisor Call | *HVC* on page F6-2686 |
| Exception Return | *ERET* on page F6-2683 |
| Load Multiple (exception return) | *LDM (exception return)* on page F6-2714 |
| Store Return State | *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F6-3013 |

a. The A32 instruction set includes other instruction forms that can be used for an exception return, that have previously been described as variants of SUBS PC, LR. ARM deprecates any use of these instruction forms.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.10 Coprocessor instructions

There are three types of instruction for communicating with conceptual coprocessors. These permit the PE to:

- Initiate a coprocessor data-processing operation. For details see *CDP, CDP2* on page F6-2647.
- Transfer general-purpose registers to and from coprocessor registers. For details, see:
  - *MCR, MCR2* on page F6-2801.
  - *MCRR, MCRR2* on page F6-2803.
  - *MRC, MRC2* on page F6-2825.
  - *MRRC, MRRC2* on page F6-2828.
- Load or store the values of coprocessor registers. For details, see:
  - *LDC, LDC2 (immediate)* on page F6-2704.
  - *LDC, LDC2 (literal)* on page F6-2708.
  - *STC, STC2* on page F6-3026.

The instruction set encodings provide supports up to 16 coprocessors, CP0 to CP15, with a 4-bit field in each coprocessor instruction to identify the coprocessor number. In ARMv8 the only supported coprocessors are CP10, CP11, CP14, and CP15, and these are supported only in AArch32 state.

**Note**

Multiple coprocessors can be used together to provide a larger block of coprocessor instructions. CP10 and CP11 are used in this way.

CP10 and CP11 are used, together, for floating-point and some Advanced SIMD functionality. There are different instructions for accessing these coprocessors, of similar types to the instructions for CP14 and CP15, that is, to:

- Initiate a coprocessor data-processing operation, see *Floating-point data-processing instructions* on page F1-2439.
- Transfer general-purpose registers to and from coprocessor registers, see *Advanced SIMD and floating-point register transfer instructions* on page F1-2430.
- Load or store the values of coprocessor registers, see *Advanced SIMD and floating-point load/store instructions* on page F1-2428.

Coprocessor instructions are part of the instruction stream executed by the PE. Any coprocessor instruction that cannot be executed by the implemented conceptual coprocessors causes an Undefined Instruction exception. This means that, in ARMv8 AArch32 state, all coprocessor access instruction encodings for coprocessors other than CP10, CP11, CP14, and CP15 are unallocated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.11 Advanced SIMD and floating-point load/store instructions

Table F1-16 summarizes the SIMD and floating-point register file load/store instructions in the Advanced SIMD and floating-point instruction sets.

Advanced SIMD also provides instructions for loading and storing multiple elements, or structures of elements, see *Element and structure load/store instructions*.

**Table F1-16 SIMD and floating-point register file load/store instructions**

| Instruction | See | Operation |
|---|---|---|
| Vector Load Multiple | *VLDM, VLDMDB, VLDMIA* on page F7-3430 | Load 1-16 consecutive 64-bit registers, Advanced SIMD and floating-point.<br>Load 1-16 consecutive 32-bit registers, floating-point only. |
| Vector Load Register | *VLDR* on page F7-3433 | Load one 64-bit register, Advanced SIMD and floating-point.<br>Load one 32-bit register, floating-point only. |
| Vector Store Multiple | *VSTM, VSTMDB, VSTMIA* on page F7-3711 | Store 1-16 consecutive 64-bit registers, Advanced SIMD and floating-point.<br>Store 1-16 consecutive 32-bit registers, floating-point only. |
| Vector Store Register | *VSTR* on page F7-3714 | Store one 64-bit register, Advanced SIMD and floating-point.<br>Store one 32-bit register, floating-point only. |

### F1.11.1 Element and structure load/store instructions

Table F1-17 shows the element and structure load/store instructions available in the Advanced SIMD instruction set. Loading and storing structures of more than one element automatically de-interleaves or interleaves the elements, see Figure F1-1 on page F1-2429 for an example of de-interleaving. Interleaving is the inverse process.

**Table F1-17 Element and structure load/store instructions**

| Instruction | See |
|---|---|
| Load single element | |
| Multiple elements | *VLD1 (multiple single elements)* on page F7-3400 |
| To one lane | *VLD1 (single element to one lane)* on page F7-3394 |
| To all lanes | *VLD1 (single element to all lanes)* on page F7-3397 |
| Load 2-element structure | |
| Multiple structures | *VLD2 (multiple 2-element structures)* on page F7-3409 |
| To one lane | *VLD2 (single 2-element structure to one lane)* on page F7-3403 |
| To all lanes | *VLD2 (single 2-element structure to all lanes)* on page F7-3406 |
| Load 3-element structure | |
| Multiple structures | *VLD3 (multiple 3-element structures)* on page F7-3418 |
| To one lane | *VLD3 (single 3-element structure to one lane)* on page F7-3412 |
| To all lanes | *VLD3 (single 3-element structure to all lanes)* on page F7-3415 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F1-17 Element and structure load/store instructions (continued)**

| Instruction | See |
|---|---|
| Load 4-element structure | |
| Multiple structures | *VLD4 (multiple 4-element structures)* on page F7-3427 |
| To one lane | *VLD4 (single 4-element structure to one lane)* on page F7-3421 |
| To all lanes | *VLD4 (single 4-element structure to all lanes)* on page F7-3424 |
| Store single element | |
| Multiple elements | *VST1 (multiple single elements)* on page F7-3690 |
| From one lane | *VST1 (single element from one lane)* on page F7-3687 |
| Store 2-element structure | |
| Multiple structures | *VST2 (multiple 2-element structures)* on page F7-3696 |
| From one lane | *VST2 (single 2-element structure from one lane)* on page F7-3693 |
| Store 3-element structure | |
| Multiple structures | *VST3 (multiple 3-element structures)* on page F7-3702 |
| From one lane | *VST3 (single 3-element structure from one lane)* on page F7-3699 |
| Store 4-element structure | |
| Multiple structures | *VST4 (multiple 4-element structures)* on page F7-3708 |
| From one lane | *VST4 (single 4-element structure from one lane)* on page F7-3705 |

Figure F1-1 shows the de-interleaving of a `VLD3.16` (multiple 3-element structures) instruction:

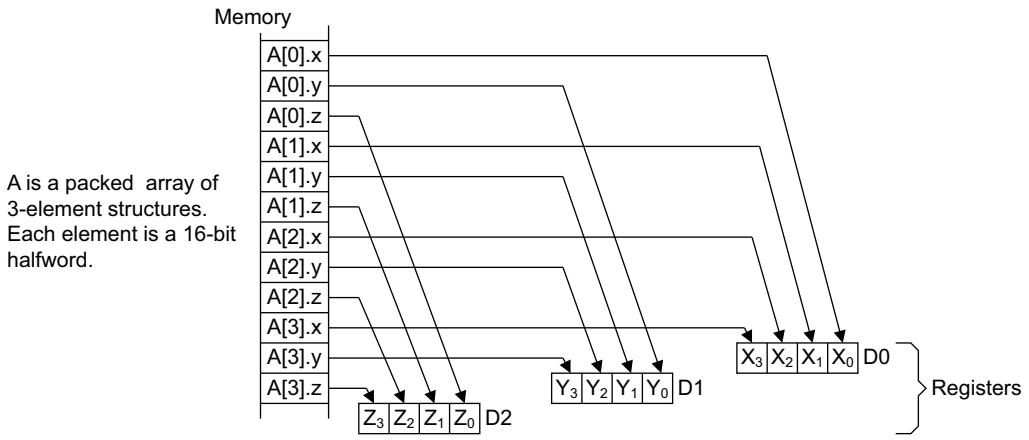



**Figure F1-1 De-interleaving an array of 3-element structures**

Figure F1-1 shows the `VLD3.16` instruction operating to three 64-bit registers that comprise four 16-bit elements:

- Different instructions in this group would produce similar figures, but operate on different numbers of registers. For example, `VLD4` and `VST4` instructions operate on four registers.
- Different element sizes would produce similar figures but with 8-bit or 32-bit elements.
- These instructions operate only on doubleword (64-bit) registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.12 Advanced SIMD and floating-point register transfer instructions

Table F1-18 summarizes the SIMD and floating-point register file transfer instructions in the Advanced SIMD and floating-point instruction sets. These instructions transfer data between the general-purpose registers and the registers in the SIMD and floating-point register file.

Advanced SIMD vectors, and single-precision and double-precision floating-point registers, are all views of the same register file. For details see *The SIMD and floating-point register file* on page E1-2325.

**Table F1-18 SIMD and floating-point register file transfer instructions**

| Instruction | See |
|---|---|
| Copy element from general-purpose register to every element of an Advanced SIMD vector | *VDUP (general-purpose register)* on page F7-3370 |
| Copy byte, halfword, or word from general-purpose register to a register in the SIMD and floating-point register file | *VMOV (general-purpose register to scalar)* on page F7-3483 |
| Copy byte, halfword, or word from a register in the SIMD and floating-point register file to a general-purpose register | *VMOV (scalar to general-purpose register)* on page F7-3487 |
| Copy from single-precision floating-point register to general-purpose register, or from general-purpose register to single-precision floating-point register | *VMOV (between general-purpose register and single-precision register)* on page F7-3485 |
| Copy two words from general-purpose registers to consecutive single-precision floating-point registers, or from consecutive single-precision floating-point registers to general-purpose registers | *VMOV (between two general-purpose registers and two single-precision registers)* on page F7-3489 |
| Copy two words from general-purpose registers to a doubleword register in the SIMD and floating-point register file, or from a doubleword register in the SIMD and floating-point register file to general-purpose registers | *VMOV (between two general-purpose registers and a doubleword floating-point register)* on page F7-3474 |
| Copy from an Advanced SIMD and floating-point System Register to a general-purpose register | *VMRS* on page F7-3495 |
| Copy from a general-purpose register to an Advanced SIMD and floating-point System Register | *VMSR* on page F7-3497 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.13 Advanced SIMD data-processing instructions

Advanced SIMD data-processing instructions process registers containing vectors of elements of the same type packed together, enabling the same operation to be performed on multiple items in parallel.

Instructions operate on vectors held in 64-bit or 128-bit registers. Figure F1-2 shows an operation on two 64-bit operand vectors, generating a 64-bit vector result.

**Note**

Figure F1-2 and other similar figures show 64-bit vectors that consist of four 16-bit elements, and 128-bit vectors that consist of four 32-bit elements. Other element sizes produce similar figures, but with one, two, eight, or sixteen operations performed in parallel instead of four.

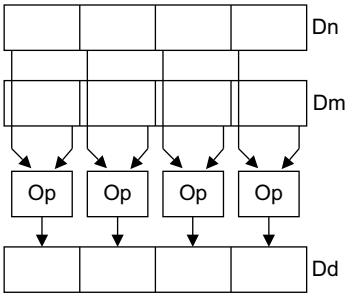



**Figure F1-2 Advanced SIMD instruction operating on 64-bit registers**

Many Advanced SIMD instructions have variants that produce vectors of elements double the size of the inputs. In this case, the number of elements in the result vector is the same as the number of elements in the operand vectors, but each element, and the whole vector, is double the size.

Figure F1-3 shows an example of an Advanced SIMD instruction operating on 64-bit registers, and generating a 128-bit result.

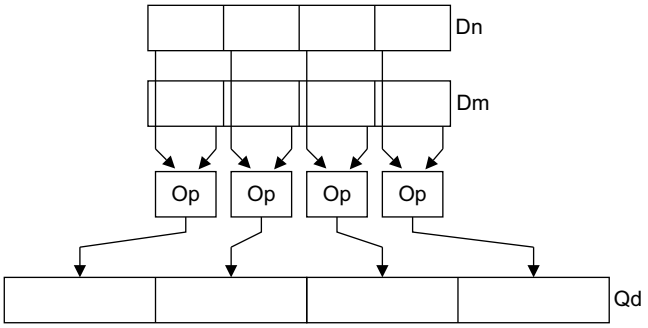



**Figure F1-3 Advanced SIMD instruction producing wider result**

There are also Advanced SIMD instructions that have variants that produce vectors containing elements half the size of the inputs. Figure F1-4 on page F1-2432 shows an example of an Advanced SIMD instruction operating on one 128-bit register, and generating a 64-bit result.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

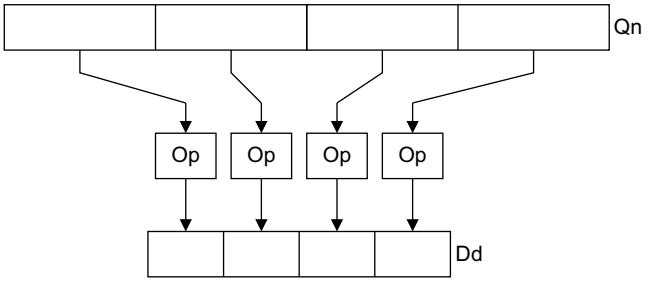



**Figure F1-4 Advanced SIMD instruction producing narrower result**

Some Advanced SIMD instructions do not conform to these standard patterns. Their operation patterns are described in the individual instruction descriptions.

Advanced SIMD instructions that perform floating-point arithmetic use the ARM standard floating-point arithmetic defined in *Floating-point and Advanced SIMD support* on page A1-46.

### F1.13.1 Advanced SIMD parallel addition and subtraction

Table F1-19 shows the Advanced SIMD parallel add and subtract instructions.

**Table F1-19 Advanced SIMD parallel add and subtract instructions**

| Instruction | See |
|---|---|
| Vector Add | *VADD (integer)* on page F7-3265<br>*VADD (floating-point)* on page F7-3262 |
| Vector Add and Narrow, returning High Half | *VADDHN* on page F7-3267 |
| Vector Add Long | *VADDL* on page F7-3269 |
| Vector Add Wide | *VADDW* on page F7-3271 |
| Vector Halving Add | *VHADD* on page F7-3390 |
| Vector Halving Subtract | *VHSUB* on page F7-3392 |
| Vector Pairwise Add and Accumulate Long | *VPADAL* on page F7-3533 |
| Vector Pairwise Add | *VPADD (integer)* on page F7-3537<br>*VPADD (floating-point)* on page F7-3535 |
| Vector Pairwise Add Long | *VPADDL* on page F7-3539 |
| Vector Rounding Add and Narrow, returning High Half | *VRADDHN* on page F7-3600 |
| Vector Rounding Halving Add | *VRHADD* on page F7-3615 |
| Vector Rounding Subtract and Narrow, returning High Half | *VRSUBHN* on page F7-3659 |
| Vector Saturating Add | *VQADD* on page F7-3555 |
| Vector Saturating Subtract | *VQSUB* on page F7-3598 |
| Vector Subtract | *VSUB (integer)* on page F7-3719<br>*VSUB (floating-point)* on page F7-3716 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F1-19 Advanced SIMD parallel add and subtract instructions (continued)**

| Instruction | See |
|---|---|
| Vector Subtract and Narrow, returning High Half | *VSUBHN* on page F7-3721 |
| Vector Subtract Long | *VSUBL* on page F7-3723 |
| Vector Subtract Wide | *VSUBW* on page F7-3725 |

### F1.13.2 Bitwise Advanced SIMD data-processing instructions

Table F1-20 shows bitwise Advanced SIMD data-processing instructions. These operate on the doubleword (64-bit) or quadword (128-bit) registers in the SIMD and floating-point register file, and there is no division into vector elements.

**Table F1-20 Bitwise Advanced SIMD data-processing instructions**

| Instruction | See |
|---|---|
| Vector Bitwise AND | *VAND (register)* on page F7-3275 |
| Vector Bitwise Bit Clear (AND complement) | *VBIC (immediate)* on page F7-3277<br>*VBIC (register)* on page F7-3279 |
| Vector Bitwise Exclusive OR | *VEOR* on page F7-3374 |
| Vector Bitwise Insert if False | *VBIF* on page F7-3281 |
| Vector Bitwise Insert if True | *VBIT* on page F7-3283 |
| Vector Bitwise Move | *VMOV (immediate)* on page F7-3476<br>*VMOV (register)* on page F7-3479 |
| Vector Bitwise NOT | *VMVN (immediate)* on page F7-3512<br>*VMVN (register)* on page F7-3514 |
| Vector Bitwise OR | *VORR (immediate)* on page F7-3529<br>*VORR (register)* on page F7-3531 |
| Vector Bitwise OR NOT | *VORN (register)* on page F7-3527 |
| Vector Bitwise Select | *VBSL* on page F7-3285 |

### F1.13.3 Advanced SIMD comparison instructions

Table F1-21 shows Advanced SIMD comparison instructions.

**Table F1-21 Advanced SIMD comparison instructions**

| Instruction | See |
|---|---|
| Vector Absolute Compare Greater Than or Equal | *VACGE* on page F7-3254 |
| Vector Absolute Compare Greater Than | *VACGT* on page F7-3258 |
| Vector Compare Equal | *VCEQ (register)* on page F7-3289 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F1-21 Advanced SIMD comparison instructions (continued)**

| Instruction | See |
|---|---|
| Vector Compare Equal to Zero | *VCEQ (immediate #0)* on page F7-3287 |
| Vector Compare Greater Than or Equal | *VCGE (register)* on page F7-3294 |
| Vector Compare Greater Than or Equal to Zero | *VCGE (immediate #0)* on page F7-3292 |
| Vector Compare Greater Than | *VCGT (register)* on page F7-3300 |
| Vector Compare Greater Than Zero | *VCGT (immediate #0)* on page F7-3298 |
| Vector Compare Less Than or Equal to Zero | *VCLE (immediate #0)* on page F7-3304 |
| Vector Compare Less Than Zero | *VCLT (immediate #0)* on page F7-3311 |
| Vector Test Bits | *VTST* on page F7-3734 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F1.13.4 Advanced SIMD shift instructions

Table F1-22 lists the shift instructions in the Advanced SIMD instruction set.

**Table F1-22 Advanced SIMD shift instructions**

| Instruction | See |
|---|---|
| Vector Saturating Rounding Shift Left | *VQRSHL* on page F7-3577 |
| Vector Saturating Rounding Shift Right and Narrow | *VQRSHRN, VQRSHRUN* on page F7-3581 |
| Vector Saturating Shift Left | *VQSHL (register)* on page F7-3589<br>*VQSHL, VQSHLU (immediate)* on page F7-3586 |
| Vector Saturating Shift Right and Narrow | *VQSHRN, VQSHRUN* on page F7-3593 |
| Vector Rounding Shift Left | *VRSHL* on page F7-3643 |
| Vector Rounding Shift Right | *VRSHR* on page F7-3645 |
| Vector Rounding Shift Right and Accumulate | *VRSRA* on page F7-3657 |
| Vector Rounding Shift Right and Narrow | *VRSHRN* on page F7-3649 |
| Vector Shift Left | *VSHL (immediate)* on page F7-3664<br>*VSHL (register)* on page F7-3666 |
| Vector Shift Left Long | *VSHLL* on page F7-3668 |
| Vector Shift Right | *VSHR* on page F7-3671 |
| Vector Shift Right and Narrow | *VSHRN* on page F7-3675 |
| Vector Shift Left and Insert | *VSLI* on page F7-3679 |
| Vector Shift Right and Accumulate | *VSRA* on page F7-3683 |
| Vector Shift Right and Insert | *VSRI* on page F7-3685 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F1.13.5 Advanced SIMD multiply instructions

Table F1-23 summarizes the Advanced SIMD multiply instructions.

**Table F1-23 Advanced SIMD multiply instructions**

| Instruction | See |
|---|---|
| Vector Multiply Accumulate | *VMLA (integer)* on page F7-3453<br>*VMLA (floating-point)* on page F7-3450<br>*VMLA (by scalar)* on page F7-3455 |
| Vector Multiply Accumulate Long | *VMLAL (integer)* on page F7-3458<br>*VMLAL (by scalar)* on page F7-3460 |
| Vector Multiply Subtract | *VMLS (integer)* on page F7-3465<br>*VMLS (floating-point)* on page F7-3462<br>*VMLS (by scalar)* on page F7-3467 |
| Vector Multiply Subtract Long | *VMLSL (integer)* on page F7-3470<br>*VMLSL (by scalar)* on page F7-3472 |
| Vector Multiply | *VMUL (integer and polynomial)* on page F7-3502<br>*VMUL (floating-point)* on page F7-3499<br>*VMUL (by scalar)* on page F7-3505 |
| Vector Multiply Long | *VMULL (integer and polynomial)* on page F7-3508<br>*VMULL (by scalar)* on page F7-3510 |
| Vector Fused Multiply Accumulate | *VFMA* on page F7-3380 |
| Vector Fused Multiply Subtract | *VFMS* on page F7-3383 |
| Vector Saturating Doubling Multiply Accumulate Long | *VQDMLAL* on page F7-3557 |
| Vector Saturating Doubling Multiply Subtract Long | *VQDMLSL* on page F7-3560 |
| Vector Saturating Doubling Multiply Returning High Half | *VQDMULH* on page F7-3563 |
| Vector Saturating Rounding Doubling Multiply Returning High Half | *VQRDMULH* on page F7-3574 |
| Vector Saturating Doubling Multiply Long | *VQDMULL* on page F7-3566 |

Advanced SIMD multiply instructions can operate on vectors of:

- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit polynomials over {0, 1}. `VMUL` and `VMULL` are the only instructions that operate on polynomials. `VMULL` produces a 16-bit polynomial over {0, 1}.
- Single-precision (32-bit) floating-point numbers.

They can also act on one vector and one scalar.

Long instructions have doubleword (64-bit) operands, and produce quadword (128-bit) results. Other Advanced SIMD multiply instructions can have either doubleword or quadword operands, and produce results of the same size.

Floating-point multiply instructions can operate on:

- Single-precision (32-bit) floating-point numbers.
- Double-precision (64-bit) floating-point numbers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F1.13.6 Miscellaneous Advanced SIMD data-processing instructions

Table F1-24 shows miscellaneous Advanced SIMD data-processing instructions.

**Table F1-24 Miscellaneous Advanced SIMD data-processing instructions**

| Instruction | See |
|---|---|
| Vector Absolute Difference and Accumulate | *VABA* on page F7-3241 |
| Vector Absolute Difference and Accumulate Long | *VABAL* on page F7-3243 |
| Vector Absolute Difference | *VABD (integer)* on page F7-3247<br>*VABD (floating-point)* on page F7-3245 |
| Vector Absolute Difference Long | *VABDL (integer)* on page F7-3249 |
| Vector Absolute | *VABS* on page F7-3251 |
| Vector Convert between floating-point and fixed point | *VCVT (between floating-point and fixed-point, Advanced SIMD)* on page F7-3337 |
| Vector Convert between floating-point and integer | *VCVT (between floating-point and integer, Advanced SIMD)* on page F7-3330 |
| Vector Convert between half-precision and single-precision | *VCVT (between half-precision and single-precision, Advanced SIMD)* on page F7-3328 |
| Vector Count Leading Sign Bits | *VCLS* on page F7-3309 |
| Vector Count Leading Zeros | *VCLZ* on page F7-3316 |
| Vector Count Set Bits | *VCNT* on page F7-3324 |
| Vector Duplicate scalar | *VDUP (scalar)* on page F7-3372 |
| Vector Extract | *VEXT (byte elements)* on page F7-3376 |
| Vector Move and Narrow | *VMOVN* on page F7-3493 |
| Vector Move Long | *VMOVL* on page F7-3491 |
| Vector Maximum | *VMAX (integer)* on page F7-3438<br>*VMAX (floating-point)* on page F7-3436 |
| Vector Minimum | *VMIN (integer)* on page F7-3445<br>*VMIN (floating-point)* on page F7-3443 |
| Vector Negate | *VNEG* on page F7-3516 |
| Vector Pairwise Maximum | *VPMAX (integer)* on page F7-3543<br>*VPMAX (floating-point)* on page F7-3541 |
| Vector Pairwise Minimum | *VPMIN (integer)* on page F7-3547<br>*VPMIN (floating-point)* on page F7-3545 |
| Vector Reciprocal Estimate | *VRECPE* on page F7-3602 |
| Vector Reciprocal Step | *VRECPS* on page F7-3604 |
| Vector Reciprocal Square Root Estimate | *VRSQRTE* on page F7-3653 |
| Vector Reciprocal Square Root Step | *VRSQRTS* on page F7-3655 |
| Vector Reverse in halfwords | *VREV16* on page F7-3606 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F1-24 Miscellaneous Advanced SIMD data-processing instructions (continued)**

| Instruction | See |
|---|---|
| Vector Reverse in words | *VREV32* on page F7-3609 |
| Vector Reverse in doublewords | *VREV64* on page F7-3612 |
| Vector Saturating Absolute | *VQABS* on page F7-3553 |
| Vector Saturating Move and Narrow | *VQMOVN, VQMOVUN* on page F7-3569 |
| Vector Saturating Negate | *VQNEG* on page F7-3572 |
| Vector Swap | *VSWP* on page F7-3727 |
| Vector Table Lookup | *VTBL, VTBX* on page F7-3729 |
| Vector Transpose | *VTRN* on page F7-3731 |
| Vector Unzip | *VUZP* on page F7-3736 |
| Vector Zip | *VZIP* on page F7-3739 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F1.14 Floating-point data-processing instructions

Table F1-25 summarizes the data-processing instructions in the floating-point instruction set. In this table, *floating-point register* means a register in the SIMD and floating-point register file.

For details of the floating-point arithmetic used by floating-point instructions, see *Floating-point and Advanced SIMD support* on page A1-46.

**Table F1-25 Floating-point data-processing instructions**

| Instruction | See |
|---|---|
| Convert between double-precision and single-precision | *VCVT (between double-precision and single-precision)* on page F7-3326 |
| Convert between floating-point and fixed-point | *VCVT (between floating-point and fixed-point, floating-point)* on page F7-3340 |
| Convert between half-precision and single-precision, writing to bottom half of single-precision register | *VCVTB* on page F7-3347 |
| Convert between half-precision and single-precision, writing to top half of single-precision register | *VCVTT* on page F7-3365 |
| Convert from floating-point to integer | *VCVT (floating-point to integer, floating-point)* on page F7-3332 |
| Convert from floating-point to integer using FPSCR rounding mode | *VCVTR* on page F7-3362 |
| Convert from integer to floating-point | *VCVT (integer to floating-point, floating-point)* on page F7-3335 |
| Copy from one floating-point register to another | *VMOV (register)* on page F7-3479 |
| Divide | *VDIV* on page F7-3368 |
| Move immediate value to a floating-point register | *VMOV (immediate)* on page F7-3476 |
| Square Root | *VSQRT* on page F7-3681 |
| Vector Absolute value | *VABS* on page F7-3251 |
| Vector Add | *VADD (floating-point)* on page F7-3262 |
| Vector Compare with exceptions disabled | *VCMPE* on page F7-3321 |
| Vector Compare with exceptions enabled | *VCMP* on page F7-3318 |
| Vector Fused Multiply Accumulate | *VFMA* on page F7-3380 |
| Vector Fused Multiply Subtract | *VFMS* on page F7-3383 |
| Vector Fused Negate Multiply Accumulate | *VFNMA* on page F7-3386 |
| Vector Fused Negate Multiply Subtract | *VFNMS* on page F7-3388 |
| Vector Multiply | *VMUL (floating-point)* on page F7-3499 |
| Vector Multiply Accumulate | *VMLA (floating-point)* on page F7-3450 |
| Vector Multiply Subtract | *VMLS (floating-point)* on page F7-3462 |
| Vector Negate Multiply | *VNMUL* on page F7-3523 |
| Vector Negate Multiply Accumulate | *VNMLA* on page F7-3519 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F1-25 Floating-point data-processing instructions (continued)**

| Instruction | See |
|---|---|
| Vector Negate Multiply Subtract | *VNMLS* on page F7-3521 |
| Vector Negate, by inverting the sign bit | *VNEG* on page F7-3516 |
| Vector Subtract | *VSUB (floating-point)* on page F7-3716 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter F2
# About the T32 and A32 Instruction Descriptions

This chapter describes each instruction. It contains the following sections:


- *Format of instruction descriptions* on page F2-2442.
- *Standard assembler syntax fields* on page F2-2446.
- *Conditional execution* on page F2-2447.
- *Shifts applied to a register* on page F2-2449.
- *Memory accesses* on page F2-2452.
- *Encoding of lists of general-purpose registers and the PC* on page F2-2453.
- *About the T32 and A32 instruction encodings* on page F2-2454.
- *Additional pseudocode support for instruction descriptions* on page F2-2461.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F2.1 Format of instruction descriptions

The instruction descriptions in Chapter F6 *T32 and A32 Base Instruction Set Instruction Descriptions* and Chapter F7 *T32 and A32 Advanced SIMD and floating-point Instruction Descriptions* normally use the following format:

- Instruction section title.
- Introduction to the instruction.
- A description of each encoding of the instruction.
- Assembler syntax.
- Pseudocode describing how the instruction operates.
- Notes, if applicable.

Each of these items is described in more detail in the following subsections.

### F2.1.1 Instruction section title

The instruction section title gives the base mnemonic for the instruction or instructions described in the section. When one mnemonic has multiple forms described in separate instruction sections, this is followed by a short description of the form in parentheses. The most common use of this is to distinguish between forms of an instruction in which one of the operands is an immediate value and forms in which it is a register.

### F2.1.2 Introduction to the instruction

The introduction to the instruction briefly describes the main features of the instruction. This description is not necessarily complete and is not definitive. If there is any conflict between it and the more detailed information that follows, the latter takes priority.

### F2.1.3 Instruction encodings

This is a list of one or more instruction encodings. Each instruction encoding is labelled as:

- A1, A2, A3 … for the first, second, third and any additional A32 encodings.
- T1, T2, T3 … for the first, second, third and any additional T32 encodings.

Each instruction encoding description consists of:

- An assembly syntax that ensures that the assembler selects the encoding in preference to any other encoding. In some cases, multiple syntax variants are given. These are written in a `typewriter` font using the conventions described in *Assembler syntax prototype line conventions* on page F2-2444. The correct one to use can be indicated by:
  - A subheading that identifies the encodings that correspond to the syntax. See, for example, the subheading *Flag setting, rotate right with extend variant* in the description of the A1 encoding of the `ADC`, `ADCS` (register) instructions in *A1* on page F6-2579.
  - An annotation to the syntax, such as *Inside IT block* or *Outside IT block*. See, for example, the syntax descriptions of the T1 encoding of the `ADC`, `ADCS` (register) instructions in *T1* on page F6-2580.

  In other cases, the correct one to use can be determined by looking at the assembler syntax description and using it to determine which syntax corresponds to the instruction being disassembled.

  There is usually more than one syntax variant that ensures re-assembly to any particular encoding, and the exact set of syntaxes that do so usually depends on the register numbers, immediate constants and other operands to the instruction. For example, when assembling to the T32 instruction set, the syntax `AND R0, R0, R8` ensures selection of a 32-bit encoding but `AND R0, R0, R1` selects a 16-bit encoding.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For each instruction encoding belonging to a target instruction set, an assembler can use this information to determine whether it can use that encoding to encode the instruction requested by the UAL source. If multiple encodings can encode the instruction then:

— If both a 16-bit encoding and a 32-bit encoding can encode the instruction, the architecture *prefers* the 16-bit encoding. This means the assembler must use the 16-bit encoding rather than the 32-bit encoding.

  Software can use the `.W` and `.N` qualifiers to specify the required encoding width, see *Standard assembler syntax fields* on page F2-2446.

— If multiple encodings of the same length can encode the instruction, the *Assembler syntax* subsection says which encoding is preferred, and how software can, instead, select the other encodings.

  Each encoding also documents UAL syntax that selects it in preference to any other encoding.

If no encodings of the target instruction set can encode the instruction requested by the UAL source, normally the assembler generates an error saying that the instruction is not available in that instruction set.

**Note**

In some cases, an instruction is available in one instruction set but not in another. The *Assembler syntax* subsection identifies many of these cases. For example, the A32 instructions with bits<31:28> == `0b1111` described in *Branch, branch with link, and block data transfer* on page F4-2535, *Coprocessor instructions, and Supervisor Call* on page F4-2537, and *Unconditional instructions* on page F4-2541 cannot have a condition code, but the equivalent T32 instructions often can, and this usually appears in the *Assembler syntax* subsection as a statement that the A32 instruction cannot be conditional.

However, some such cases are too complex to describe in the available space, so the definitive test of whether an instruction is available in a given instruction set is whether there is an available encoding for it in that instruction set.

The assembly syntax given for an encoding is therefore a suitable one for a disassembler to disassemble that encoding to. However, disassemblers might wish to use simpler syntaxes when they are suitable for the operand combination, in order to produce more readable disassembled code.

- An encoding diagram. This is half-width for 16-bit T32 encodings and full-width for 32-bit T32 and A32 encodings. The 32-bit A32 encoding diagrams number the bits from 31 to 0, while the 32-bit T32 encoding diagrams number the bits from 15 to 0 for each halfword, to distinguish them from A32 encodings and to act as a reminder that a 32-bit T32 instruction consists of two consecutive halfwords rather than a word.

  In particular, if instructions are stored using the standard little-endian instruction endianness, the encoding diagram for an A32 instruction at address A shows the bytes at addresses A+3, A+2, A+1, A from left to right, but the encoding diagram for a 32-bit T32 instruction shows them in the order A+1, A for the first halfword, followed by A+3, A+2 for the second halfword.

- Encoding-specific pseudocode. This is pseudocode that translates the encoding-specific instruction fields into inputs to the encoding-independent pseudocode in the *Operation* subsection, and that picks out any special cases in the encoding. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix K11 *ARM Pseudocode Definition*.

### F2.1.4 Assembler symbols

The *Assembly symbols* describes the standard UAL syntax for the instruction.

Each syntax description consists of the following elements:

- Descriptions of all of the variable or optional fields of the syntax.

  Some syntax fields are standardized across all or most instructions. *Standard assembler syntax fields* on page F2-2446 describes these fields.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

By default, syntax fields that specify registers, such as <Rd>, <Rn>, or <Rt>, can be any of R0-R12 or LR in T32 instructions, and any of R0-R12, SP or LR in A32 instructions. These require that the encoding-specific pseudocode set the corresponding integer variable (such as d, n, or t) to the corresponding register number, using 0-12 for R0-R12, 13 for SP, or 14 for LR:

— Normally, software can do this by setting the corresponding field in the instruction, typically named Rd, Rn, Rt, to the binary encoding of that number.

— In the case of 16-bit T32 encodings, the field is normally of length 3, and so the encoding is only available when the assembler syntax specifies one of R0-R7. Such encodings often use a register field name like Rdn. This indicates that the encoding is only available if <Rd> and <Rn> specify the same register, and that the register number of that register is encoded in the field if they do.

The description of a syntax field that specifies a register sometimes extends or restricts the permitted range of registers or documents other differences from the default rules for such fields. Examples of extensions are permitting the use of the SP in a T32 instruction, or permitting the use of the PC, identified using register number 15.

- Where appropriate, text that briefly describes changes from the pre-UAL assembler syntax. Where present, this usually consists of an alternative pre-UAL form of the assembler mnemonic. The pre-UAL assembler syntax does not conflict with UAL. ARM recommends that it is supported, as an optional extension to UAL, so that pre-UAL assembler source files can be assembled.

### Assembler syntax prototype line conventions

The following conventions are used in assembler syntax prototype lines and their subfields:

**< >** Any item bracketed by < and > is a short description of a type of value to be supplied by the user in that position. A longer description of the item is normally supplied by subsequent text. Such items often correspond to a similarly named field in an encoding diagram for an instruction. When the correspondence only requires the binary encoding of an integer value or register number to be substituted into the instruction encoding, it is not described explicitly. For example, if the assembler syntax for an instruction contains an item <Rn> and the instruction encoding diagram contains a 4-bit field named Rn, the number of the register specified in the assembler syntax is encoded in binary in the instruction field.

If the correspondence between the assembler syntax item and the instruction encoding is more complex than simple binary encoding of an integer or register number, the item description indicates how it is encoded. This is often done by specifying a required output from the encoding-specific pseudocode, such as add = TRUE. The assembler must only use encodings that produce that output.

**{ }** Any item bracketed by { and } is optional. A description of the item and of how its presence or absence is encoded in the instruction is normally supplied by subsequent text.

Many instructions have an optional destination register. Unless otherwise stated, if such a destination register is omitted, it is the same as the immediately following source register in the instruction syntax.

**#** In the assembler syntax, numeric constants are normally preceded by a #. Some UAL instruction syntax descriptions explicitly show this # as optional. Any UAL assembler:

- Must treat the # as optional where an instruction syntax description shows it as optional.
- Can treat the # either as mandatory or as optional where an instruction syntax description does not show it as optional.

**Note**

ARM recommends that UAL assemblers treat all uses of # shown in this manual as optional.

**spaces** Single spaces are used for clarity, to separate items. When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.

**+/-** This indicates an optional + or - sign. If neither is coded, + is assumed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other characters must be encoded precisely as they appear in the assembler syntax. Apart from { and }, the special characters described above do not appear in the basic forms of assembler instructions documented in this manual. In a few places, the { and } characters must be encoded as part of a variable item. When this happens, the long description of the variable item indicates how they must be used.

### F2.1.5 Pseudocode describing how the instruction operates

The *Operation for all classes* subsection contains encoding-independent pseudocode that describes the main operation of the instruction. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix K11 *ARM Pseudocode Definition*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F2.2 Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

<c> Is an optional field. It specifies the condition under which the instruction is executed. See *Conditional execution* on page F2-2447 for the range of available conditions and their encoding. If <c> is omitted, it defaults to *always* (AL).

<q> Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.
- .W Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither .W nor .N is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description. The assembler syntax includes a mandatory .W qualifier, along with a note describing the cases in which it applies, where this qualifier is required to select a particular encoding for an instruction. Additional assembler syntax will describe the syntax when the conditions are not met.

**Note**

When assembling to the A32 instruction set, the .N qualifier produces an assembler error and the .W qualifier has no effect.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F2.3 Conditional execution

Most T32 and A32 instructions can be executed conditionally, based on the values of the APSR condition flags. Table F2-1 lists the available conditions.

**Table F2-1 Condition codes**

| cond | Mnemonic extension | Meaning (integer) | Meaning (floating-point) [a] | Condition flags |
|---|---|---|---|---|
| 0000 | EQ | Equal | Equal | Z == 1 |
| 0001 | NE | Not equal | Not equal, or unordered | Z == 0 |
| 0010 | CS [b] | Carry set | Greater than, equal, or unordered | C == 1 |
| 0011 | CC [c] | Carry clear | Less than | C == 0 |
| 0100 | MI | Minus, negative | Less than | N == 1 |
| 0101 | PL | Plus, positive or zero | Greater than, equal, or unordered | N == 0 |
| 0110 | VS | Overflow | Unordered | V == 1 |
| 0111 | VC | No overflow | Not unordered | V == 0 |
| 1000 | HI | Unsigned higher | Greater than, or unordered | C == 1 and Z == 0 |
| 1001 | LS | Unsigned lower or same | Less than or equal | C == 0 or Z == 1 |
| 1010 | GE | Signed greater than or equal | Greater than or equal | N == V |
| 1011 | LT | Signed less than | Less than, or unordered | N != V |
| 1100 | GT | Signed greater than | Greater than | Z == 0 and N == V |
| 1101 | LE | Signed less than or equal | Less than, equal, or unordered | Z == 1 or N != V |
| 1110 | None (AL) [d] | Always (unconditional) | Always (unconditional) | Any |

a. Unordered means at least one NaN operand.
b. HS (unsigned higher or same) is a synonym for CS.
c. LO (unsigned lower) is a synonym for CC.
d. AL is an optional mnemonic extension for always, except in IT instructions. For details see *IT* on page F6-2690.

In T32 instructions, the condition, if it is not AL, is normally encoded in a preceding IT instruction. For more information see *Conditional instructions* on page F1-2409 and *IT* on page F6-2690. Some conditional branch instructions do not require a preceding IT instruction, because they include a condition code in their encoding.

In A32 instructions, bits[31:28] of the instruction contain either:

- The condition code, see *The condition code field in A32 instruction encodings* on page F2-2448.
- 0b1111 for some A32 instructions that can only be executed unconditionally.

ARM deprecates the conditional execution of any instruction encoding provided by Advanced SIMD that is not also provided by floating-point, and strongly recommends that:

- For A32 instructions, any such Advanced SIMD instruction that can be conditionally executed is executed with the <c> field omitted or set to AL.

  **Note**

  This applies only to VDUP, see *VDUP (general-purpose register)* on page F7-3370. The other A32 instructions do not permit conditional execution.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For T32 instructions, such Advanced SIMD instructions are never included in an IT block. This means they must be specified with the <c> field omitted or set to AL.

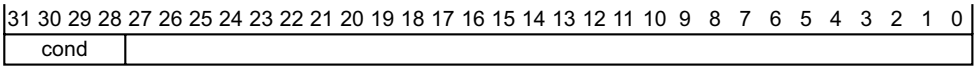
This deprecation does not apply to Advanced SIMD instruction encodings that are also available as floating-point instruction encodings. That is, it does not apply to the Advanced SIMD encodings of the instructions described in the following sections:

- *VLDM, VLDMDB, VLDMIA* on page F7-3430.
- *VLDR* on page F7-3433.
- *VMOV (general-purpose register to scalar)* on page F7-3483.
- *VMOV (between two general-purpose registers and a doubleword floating-point register)* on page F7-3474.
- *VMRS* on page F7-3495.
- *VMSR* on page F7-3497.
- *VPOP* on page F7-3549.
- *VPUSH* on page F7-3551.
- *VSTM, VSTMDB, VSTMIA* on page F7-3711.
- *VSTR* on page F7-3714.

See also *Conditional execution of undefined instructions* on page G1-3813.

### F2.3.1 The condition code field in A32 instruction encodings

Every conditional A32 instruction contains a 4-bit condition code field, the cond field, in bits 31 to 28:




This field contains one of the values 0b0000-0b1110, as shown in Table F2-1 on page F2-2447. Most instruction mnemonics can be extended with the letters defined in the *Mnemonic extension* on page F2-2447 column of that table.

If the *always* (AL) condition is specified, the instruction is executed irrespective of the value of the condition flags. The absence of a condition code on an instruction mnemonic implies the AL condition code.

### F2.3.2 Pseudocode description of conditional execution

The AArch32.CurrentCond() function returns a 4-bit condition specifier as follows:

- For A32 instructions, it returns bits[31:28] of the instruction.
- For the T1 and T3 encodings of the Branch instruction (see *B* on page F6-2623), it returns the 4-bit cond field of the encoding.
- For all other T32 instructions:
  - If PSTATE.IT<3:0> != '0000' it returns PSTATE.IT<7:4>.
  - If PSTATE.IT<7:0> == '00000000' it returns '1110'.
  - Otherwise, execution of the instruction is UNPREDICTABLE.

  For more information, see *Process state, PSTATE* on page E1-2319.

The ConditionPassed() function uses this condition specifier and the condition flags to determine whether the instruction must be executed, by calling the ConditionHolds() function.

Chapter J1 *ARMv8 Pseudocode* includes the definitions of these functions.

*Undefined Instruction exception* on page G1-3811 describes the handling of conditional instructions that are UNDEFINED or UNPREDICTABLE. The pseudocode in the manual, as a sequential description of the instructions, has limitations in this respect. For more information, see *Limitations of the instruction pseudocode* on page K11-5650.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F2.4 Shifts applied to a register

A32 register offset load/store word and unsigned byte instructions can apply a wide range of different constant shifts to the offset register. Both T32 and A32 data-processing instructions can apply the same range of different constant shifts to the second operand register. For details see *Constant shifts*.

A32 data-processing instructions can apply a register-controlled shift to the second operand register.

### F2.4.1 Constant shifts

These are the same in T32 and A32 instructions, except that the input bits come from different positions.

`<shift>` is an optional shift to be applied to `<Rm>`. It can be any one of:

**(omitted)** No shift.

`LSL #<n>` Logical shift left `<n>` bits. 1 <= `<n>` <= 31.

`LSR #<n>` Logical shift right `<n>` bits. 1 <= `<n>` <= 32.

`ASR #<n>` Arithmetic shift right `<n>` bits. 1 <= `<n>` <= 32.

`ROR #<n>` Rotate right `<n>` bits. 1 <= `<n>` <= 31.

`RRX` Rotate right one bit, with extend. Bit[0] is written to `shifter_carry_out`, bits[31:1] are shifted right one bit, and the Carry flag is shifted into bit[31].

**Note**

Assemblers can permit the use of some or all of `ASR #0`, `LSL #0`, `LSR #0`, and `ROR #0` to specify that no shift is to be performed. This is not standard UAL, and the encoding selected for T32 instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must omit the shift specifier when the instruction specifies no shift.

Similarly, assemblers can permit the use of `#0` in the immediate forms of `ASR`, `LSL`, `LSR`, and `ROR` instructions to specify that no shift is to be performed, that is, that a `MOV` (register) instruction is wanted. Again, this is not standard UAL, and the encoding selected for T32 instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must use the `MOV` (register) syntax when the instruction specifies no shift.

#### Encoding

The assembler encodes `<shift>` into two type bits and five immediate bits, as follows:

**(omitted)** type = `0b00`, immediate = 0.

`LSL #<n>` type = `0b00`, immediate = `<n>`.

`LSR #<n>` type = `0b01`.
If `<n>` < 32, immediate = `<n>`.
If `<n>` == 32, immediate = 0.

`ASR #<n>` type = `0b10`.
If `<n>` < 32, immediate = `<n>`.
If `<n>` == 32, immediate = 0.

`ROR #<n>` type = `0b11`, immediate = `<n>`.

`RRX` type = `0b11`, immediate = 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F2.4.2 Register controlled shifts

These are only available in A32 instructions.

<type> is the type of shift to apply to the value read from <Rm>. It must be one of:

| | |
|---|---|
| ASR | Arithmetic shift right, encoded as type = 0b10. |
| LSL | Logical shift left, encoded as type = 0b00. |
| LSR | Logical shift right, encoded as type = 0b01. |
| ROR | Rotate right, encoded as type = 0b11. |

The bottom byte of <Rs> contains the shift amount.

### F2.4.3 Pseudocode description of instruction-specified shifts and rotates

```
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};

// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);

// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) type)
    case type of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;

// Shift()
// =======

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, type, amount, carry_in);
    return result;

// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when SRType_LSR
            (result, carry_out) = LSR_C(value, amount);
        when SRType_ASR
            (result, carry_out) = ASR_C(value, amount);
        when SRType_ROR
            (result, carry_out) = ROR_C(value, amount);
        when SRType_RRX
            (result, carry_out) = RRX_C(value, carry_in);

return (result, carry_out);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F2.5 Memory accesses

Commonly, the following addressing modes are permitted for memory access instructions:

**Offset addressing**

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The value of the base register is unchanged.

The assembly language syntax for this mode is:

`[<Rn>, <offset>]`

**Pre-indexed addressing**

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register.

The assembly language syntax for this mode is:

`[<Rn>, <offset>]!`

**Post-indexed addressing**

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register

The assembly language syntax for this mode is:

`[<Rn>], <offset>`

In each case, `<Rn>` is the base register. `<offset>` can be:

- An immediate constant, such as `<imm8>` or `<imm12>`.
- An index register, `<Rm>`.
- A shifted index register, such as `<Rm>, LSL #<shift>`.

For information about unaligned access, endianness, and exclusive access, see:

- *Alignment support* on page E2-2363.
- *Endian support* on page E2-2365.
- *Synchronization and semaphores* on page E2-2395.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F2.6 Encoding of lists of general-purpose registers and the PC

A number of instructions operate on lists of general-purpose registers. For some load instructions, the list of registers to be loaded can include the PC. For these instructions, the assembler syntax includes a `<registers>` field, that provides a list of the registers to be operated on, with list entries separated by commas.

The registers list is encoded in the instruction encoding. Most often, this is done using an 8-bit, 13-bit, or 16-bit `register_list` field. This section gives more information about these and other possible register list encodings.

In a `register_list` field, each bit corresponds to a single register, and if the `<registers>` field of the assembler instruction includes Rt then `register_list<t>` is set to 1, otherwise it is set to 0.

The full rules for the encoding of lists of general-purpose registers, and possibly the PC, are:

- Except for the cases listed here, 16-bit T32 encodings use an 8-bit register list, and can access only registers R0-R7.

  The exceptions to this rule are:

  — The T1 encoding of `POP` uses an 8-bit register list, and an additional bit, `P`, that corresponds to the PC. This means it can access any of R0-R7 and the PC.

  — The T1 encoding of `PUSH` uses an 8-bit register list, and an additional bit, `M`, that corresponds to the LR. This means it can access any of R0-R7 and the LR.

- 32-bit T32 encodings of load operations use a 13-bit register list, and two additional bits, `M`, corresponding to the LR, and `P`, corresponding to the PC. This means these instructions can access any of R0-R12 and the LR and PC.

- 32-bit T32 encodings of store operations use a 13-bit register list, and one additional bit, `M`, corresponding to the LR. This means these instructions can access any of R0-R12 and the LR.

- Except for the case listed here, A32 encodings use a 16-bit register list. This means these instructions can access any of R0-R12 and the SP, LR, and PC.

  The exception to this rule is:

  — The system instructions `LDM` (exception return) and `LDM` (User registers) use a 15-bit register list. This means these instructions can access any of R0-R12 and the SP and LR.

- The T3 and A2 encodings of `POP`, and the T3 and A2 encodings of `PUSH`, access a single register from the set of registers {R0-R12, LR, PC} and encode the register number in the `Rt` field.

**Note**

`POP` is a load operation, and `PUSH` is a store operation.

In every case, the encoding-specific pseudocode converts the register list into a 32-bit variable, `registers`, with a bit corresponding to each of the registers R0-R12, SP, LR, and PC.

**Note**

Some floating-point and Advanced SIMD instructions operate on lists of SIMD and floating-point registers. The assembler syntax of these instructions includes a `<list>` field that specifies the registers to be operated on, and the description of the instruction in *Alphabetical list of T32 and A32 base instruction set instructions* on page F6-2576 defines the use and encoding of this field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F2.7 About the T32 and A32 instruction encodings

Chapter F3 *T32 Base Instruction Set Encoding* describes the T32 instruction encodings, and Chapter F4 *A32 Base Instruction Set Encoding* describes the A32 instruction encodings. The following subsections give more information about these encodings:

- *UNDEFINED and UNPREDICTABLE instruction set space*.
- *The PC and the use of 0b1111 as a register specifier* on page F2-2455.
- *The SP and the use of 0b1101 as a register specifier* on page F2-2456.
- *Modified immediate constants in A32 instructions* on page F2-2456.
- *Modified immediate constants in T32 instructions* on page F2-2459.

### F2.7.1 UNDEFINED and UNPREDICTABLE instruction set space

An attempt to execute an unallocated instruction results in either:

- Unpredictable behavior. The instruction is described as UNPREDICTABLE.

  ARMv8-A greatly reduces the architecturally UNPREDICTABLE behavior in AArch32 state. Many cases that earlier versions of the architecture describe as unpredictable become either:

  — CONSTRAINED UNPREDICTABLE, meaning the architecture defines a limited range of permitted behaviors.

  — Fully predictable.

  For more information see Appendix K1 *Architectural Constraints on UNPREDICTABLE behaviors*.

  The AArch32 parts of this manual might sometimes describe as UNPREDICTABLE behavior that ARMv8-A makes CONSTRAINED UNPREDICTABLE.

- An Undefined Instruction exception. The instruction is described as UNDEFINED.

An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description, or in Chapter F3 *T32 Base Instruction Set Encoding* or Chapter F4 *A32 Base Instruction Set Encoding*.

An instruction is UNPREDICTABLE if:

- It is declared as UNPREDICTABLE in an instruction description or in Chapter F3 or Chapter F4, and Appendix K1 does not redefine the behavior as CONSTRAINED UNPREDICTABLE.
- The pseudocode for that encoding does not indicate that a different special case applies, and a bit marked (0) or (1) in the encoding diagram of an instruction is not 0 or 1 respectively. In most cases, ARMv8 makes these cases CONSTRAINED UNPREDICTABLE, as described in *SBZ or SBO fields in instructions* on page K1-5426.

Unless otherwise specified, T32 and A32 instructions provided as part of an architectural extension, or by an optional feature of the architecture, are UNDEFINED in an implementation that does not include that extension or feature.

**Note**

Examples of where this rule applies are:

- The instructions provided by the Cryptographic Extension.
- The system instructions that provide access to the System registers of the OPTIONAL Performance Monitors Extension.
- The Advanced SIMD and floating-point instructions.

For more information about UNDEFINED and UNPREDICTABLE instruction behavior, see *Undefined Instruction exception* on page G1-3811.

For more information about the behavior of T32 and A32 instructions in earlier versions of the architecture see the *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F2.7.2 The PC and the use of 0b1111 as a register specifier

Restrictions on the use of PC or 0b1111 as a register specifier differ between the T32 and the A32 instruction sets, as described in:

- *T32 restrictions on the use of the PC, and use of 0b1111 as a register specifier*.
- *A32 restrictions on the use of PC or 0b1111 as a register specifier* on page F2-2456.

#### T32 restrictions on the use of the PC, and use of 0b1111 as a register specifier

The use of 0b1111 as a register specifier is not normally permitted in T32 instructions. When a value of 0b1111 is permitted, a variety of meanings is possible. For register reads, these meanings include:

- Read the PC value, that is, the address of the current instruction + 4. The base register of the table branch instructions TBB and TBH can be the PC. This means branch tables can be placed in memory immediately after the instruction.

  **Note**

  ARM deprecates use of the PC as the base register in the STC instruction.

- Read the word-aligned PC value, that is, the address of the current instruction + 4, with bits[1:0] forced to zero. The base register of LDC, LDR, LDRB, LDRD (pre-indexed, no writeback), LDRH, LDRSB, and LDRSH instructions can be the word-aligned PC. This provides PC-relative data addressing. In addition, some encodings of the ADD and SUB instructions permit their source registers to be 0b1111 for the same purpose.

- Read zero. This is done in some cases when one instruction is a special case of another, more general instruction, but with one operand zero. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

For register writes, these meanings include:

- The PC can be specified as the destination register of an LDR instruction. This is done by encoding Rt as 0b1111. The loaded value is treated as an address, and the effect of execution is a branch to that address. Bit[0] of the loaded value selects whether to execute A32 or T32 instructions after the branch.

- Some other instructions write the PC in similar ways. An instruction can specify that the PC is written:
  - Implicitly, for example, branch instructions.
  - Explicitly by a register specifier of 0b1111, for example 16-bit MOV (register) instructions.
  - Explicitly by using a register mask, for example LDM instructions.

  The address to branch to can be:
  - A loaded value, for example, RFE.
  - A register value, for example, BX.
  - The result of a calculation, for example, TBB or TBH.

  The method of choosing the instruction set used after the branch can be:
  - Similar to the LDR case, for example, LDM or BX.
  - A fixed instruction set other than the one currently being used, for example, the immediate form of BLX.
  - Unchanged, for example, branch instructions or 16-bit MOV (register) instructions.
  - Set from the SPSR.T bit, for RFE and SUBS PC, LR, #imm8.

- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

- If the destination register specifier of an LDRB, LDRH, LDRSB, or LDRSH instruction is 0b1111, the instruction is a memory hint instead of a load operation.

- If the destination register specifier of an MRC instruction is 0b1111, bits[31:28] of the value transferred from the coprocessor are written to the N, Z, C, and V condition flags in the APSR, and bits[27:0] are discarded.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### A32 restrictions on the use of PC or 0b1111 as a register specifier

In A32 instructions, the use of 0b1111 as a register specifier specifies the PC.

Many instructions are UNPREDICTABLE if they use 0b1111 as a register specifier. This is specified by pseudocode in the instruction description. ARMv8-A constrains the resulting UNPREDICTABLE behavior, see *Using R15* on page K1-5423.

**Note**

ARM deprecates use of the PC as the base register in any store instruction.

## F2.7.3 The SP and the use of 0b1101 as a register specifier

In the T32 and A32 instruction sets, ARM strongly recommends that the use of 0b1101 as a register specifier specifies the SP, and deprecates using SP (R13) for any purpose other than as a stack pointer.

**Note**

- The recommendation that register specifier 0b1101 is used only to specify the SP applies to both the T32 and the A32 instruction sets.
- Despite this recommendation, in ARMv8, most T32 uses of R13 as a general-purpose register behave predictably. This differs from ARMv7, where many uses of R13 are UNPREDICTABLE. For more information, see the *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

## F2.7.4 Modified immediate constants in A32 instructions

The encoding of a modified immediate constant in an A32 instruction is:

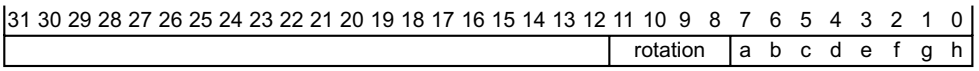



Table F2-2 shows the range of modified immediate constants available in A32 data-processing instructions, and their encoding in the a, b, c, d, e, f, g, and h bits and the rotation field in the instruction.

**Table F2-2 Encoding of modified immediates in A32 processing instructions**

| rotation | <const> [a] |
|---|---|
| 0000 | 00000000 00000000 00000000 abcdefgh |
| 0001 | gh000000 00000000 00000000 00abcdef |
| 0010 | efgh0000 00000000 00000000 0000abcd |
| 0011 | cdefgh00 00000000 00000000 000000ab |
| 0100 | abcdefgh 00000000 00000000 00000000 |
| . | . |
| . | . 8-bit values shifted to other even-numbered positions |
| . | . |
| 1001 | 00000000 00abcdef gh000000 00000000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F2-2 Encoding of modified immediates in A32 processing instructions (continued)**

| rotation | <const> [a] |
|---|---|
| . | . |
| . | . 8-bit values shifted to other even-numbered positions |
| . | . |
| 1110 | 00000000 00000000 0000abcd efgh0000 |
| 1111 | 00000000 00000000 000000ab cdefgh00 |

a. This table shows the immediate constant value in binary form, to relate abcdefgh to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

**Note**

The range of values available in A32 modified immediate constants is slightly different from the range of values available in 32-bit T32 instructions. See *Modified immediate constants in T32 instructions* on page F2-2459.

### Carry out

A logical instruction with the rotation field set to `0b0000` does not affect APSR.C. Otherwise, a logical flag-setting instruction sets APSR.C to the value of bit[31] of the modified immediate constant.

### Constants with multiple encodings

Some constant values have multiple possible encodings. In this case, a UAL assembler must select the encoding with the lowest unsigned value of the rotation field. This is the encoding that appears first in Table F2-2 on page F2-2456. For example, the constant `#3` must be encoded with (rotation, abcdefgh) == (`0b0000`, `0b00000011`), not (`0b0001`, `0b00001100`), (`0b0010`, `0b00110000`), or (`0b0011`, `0b11000000`).

In particular, this means that all constants in the range 0-255 are encoded with rotation == `0b0000`, and permitted constants outside that range are encoded with rotation != `0b0000`. A flag-setting logical instruction with a modified immediate constant therefore leaves APSR.C unchanged if the constant is in the range 0-255 and sets it to the most significant bit of the constant otherwise. This matches the behavior of T32 modified immediate constants for all constants that are permitted in both the A32 and T32 instruction sets.

An alternative syntax is available for a modified immediate constant that permits the programmer to specify the encoding directly. In this syntax, `#<const>` is instead written as `#<byte>, #<rot>`, where:

`<byte>` Is the numeric value of abcdefgh, in the range 0-255.

`<rot>` Is twice the numeric value of rotation, an even number in the range 0-30.

This syntax permits all A32 data-processing instructions with modified immediate constants to be disassembled to assembler syntax that assembles to the original instruction.

This syntax also makes it possible to write variants of some flag-setting logical instructions that have different effects on APSR.C to those obtained with the normal `#<const>` syntax. For example, `ANDS R1, R2, #12, #2` has the same behavior as `ANDS R1, R2, #3` except that it sets APSR.C to 0 instead of leaving it unchanged. Such variants of flag-setting logical instructions do not have equivalents in the T32 instruction set, and ARM deprecates their use.

### Operation of modified immediate constants, A32 instructions

```
// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    (imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

    return imm32;

// A32ExpandImm_C()
// ================

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRType_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F2.7.5 Modified immediate constants in T32 instructions

The encoding of a modified immediate constant in a 32-bit T32 instruction is:

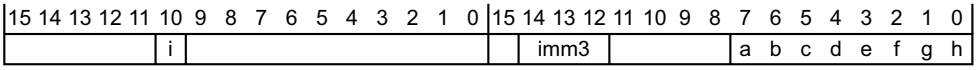



Table F2-3 shows the range of modified immediate constants available in T32 data-processing instructions, and their encoding in the a, b, c, d, e, f, g, h, and i bits, and the imm3 field, in the instruction.

**Table F2-3 Encoding of modified immediates in T32 data-processing instructions**

| i:imm3:a | <const> [a] |
|---|---|
| 0000x | `00000000 00000000 00000000 abcdefgh` |
| 0001x | `00000000 abcdefgh 00000000 abcdefgh` [b] |
| 0010x | `abcdefgh 00000000 abcdefgh 00000000` [b] |
| 0011x | `abcdefgh abcdefgh abcdefgh abcdefgh` [b] |
| 01000 | `1bcdefgh 00000000 00000000 00000000` |
| 01001 | `01bcdefg h0000000 00000000 00000000` [c] |
| 01010 | `001bcdef gh000000 00000000 00000000` |
| 01011 | `0001bcde fgh00000 00000000 00000000` [c] |
| . . . | . . . 8-bit values shifted to other positions |
| 11101 | `00000000 00000000 000001bc defgh000` [c] |
| 11110 | `00000000 00000000 0000001b cdefgh00` |
| 11111 | `00000000 00000000 00000001 bcdefgh0` [c] |

a. This table shows the immediate constant value in binary form, to relate `abcdefgh` to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).
b. ARM deprecates using a modified immediate with abcdefgh == 00000000.
c. Not available in A32 instructions if `h` == 1.

**Note**

As the footnotes to Table F2-3 show, the range of values available in T32 modified immediate constants is slightly different from the range of values available in A32 instructions. See *Modified immediate constants in A32 instructions* on page F2-2456 for the A32 values.

#### Carry out

A logical instruction with i:imm3:a == '00xxx' does not affect the Carry flag. Otherwise, a logical flag-setting instruction sets the Carry flag to the value of bit[31] of the modified immediate constant.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation of modified immediate constants, T32 instructions

```
// T32ExpandImm()
// ==============

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;

// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F2.8 Additional pseudocode support for instruction descriptions

Earlier sections of this chapter include pseudocode that describes features of the execution of A32 and T32 instructions, see:

- *Pseudocode description of conditional execution* on page F2-2448.
- *Pseudocode description of instruction-specified shifts and rotates* on page F2-2450

The following subsection gives additional pseudocode support functions for some of the instructions described in *Alphabetical list of T32 and A32 base instruction set instructions* on page F6-2576. See also *Pseudocode support for the Banked register transfer instructions* on page F6-3205.

### F2.8.1 Pseudocode description of coprocessor operations

The `Coproc_CheckInstr()` pseudocode function determines whether a coprocessor instruction is accepted for execution, see *Pseudocode description of checking accesses to the conceptual coprocessors CP14 and CP15* on page G1-3840.

The `Coproc_DoneLoading()` pseudocode function determines, for an `LDC` instruction, whether enough words have been loaded:

```
boolean Coproc_DoneLoading(integer cp_num, bits(32) instr);
```

The `Coproc_DoneStoring()` function determines for an `STC` instruction whether enough words have been stored:

```
boolean Coproc_DoneStoring(integer cp_num, bits(32) instr);
```

The `Coproc_GetOneWord()` function obtains the word for an `MRC` instruction from the coprocessor:

```
bits(32) Coproc_GetOneWord(integer cp_num, bits(32) instr);
```

The `Coproc_GetTwoWords()` function obtains the two words for an `MRRC` instruction from the coprocessor:

```
(bits(32), bits(32)) Coproc_GetTwoWords(integer cp_num, bits(32) instr);
```

**Note**

The relative significance of the two words returned is IMPLEMENTATION DEFINED, but all uses within this manual present the two words in the order (most significant, least significant).

The `Coproc_GetWordToStore()` function obtains the next word to store for an `STC` instruction from the coprocessor:

```
bits(32) Coproc_GetWordToStore(integer cp_num, bits(32) instr);
```

The `Coproc_InternalOperation()` procedure instructs a coprocessor to perform the internal operation requested by a `CDP` instruction:

```
Coproc_InternalOperation(integer cp_num, bits(32) instr);
```

The `Coproc_SendLoadedWord()` procedure sends a loaded word for an `LDC` instruction to the coprocessor:

```
Coproc_SendLoadedWord(bits(32) word, integer cp_num, bits(32) instr);
```

The `Coproc_SendOneWord()` procedure sends the word for an `MCR` instruction to the coprocessor:

```
Coproc_SendOneWord(bits(32) word, integer cp_num, bits(32) instr);
```

The `Coproc_SendTwoWords()` procedure sends the two words for an `MCRR` instruction to the coprocessor:

```
Coproc_SendTwoWords(bits(32) word2, bits(32) word1, integer cp_num, bits(32) instr);
```

**Note**

The relative significance of `word2` and `word1` is IMPLEMENTATION DEFINED, but all uses within this manual treat `word2` as more significant than `word1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The CP14DebugInstrDecode() pseudocode function decodes an accepted access to a CP14 debug register:

```
boolean CP15InstrDecode(bits(32) instr);
```

The CP14JazelleInstrDecode() pseudocode function decodes an accepted access to a CP14 Jazelle register:

```
boolean CP14JazelleInstrDecode(bits(32) instr);
```

The CP14TraceInstrDecode() pseudocode function decodes an accepted access to a CP14 Trace register:

```
boolean CP14TraceInstrDecode(bits(32) instr);
```

The CP15InstrDecode() pseudocode function decodes an accepted access to a CP15 register:

```
boolean CP15InstrDecode(bits(32) instr);
```

### F2.8.2 Pseudocode details of system calls

The CallSupervisor() pseudocode function generates a Supervisor Call exception. Valid execution of the SVC instruction calls this function.

```
// AArch32.CallSupervisor()
// ========================
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate)

    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;

    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCException(immediate);
```

The CallHypervisor() pseudocode function generates an HVC exception. Valid execution of the HVC instruction calls this function.

```
// AArch32.CallHypervisor()
// ========================
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCException(immediate);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter F3
# T32 Base Instruction Set Encoding

This chapter introduces the T32 instruction set and describes how it uses the ARM programmers' model. It contains the following sections:

- *Top level T32 instruction set encoding* on page F3-2464.
- *16-bit T32 instruction encoding* on page F3-2465.
- *32-bit T32 instruction encoding* on page F3-2476.

**Note**

In the decode tables in this chapter, an entry of - for a field value means the value of the field does not affect the decoding.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F3.1 Top level T32 instruction set encoding

The T32 instruction stream is a sequence of halfword-aligned halfwords. Each T32 instruction is either a single 16-bit halfword in that stream, or a 32-bit instruction consisting of two consecutive halfwords in that stream.

If the value of bits[15:11] of the halfword being decoded is one of the following, the halfword is the first halfword of a 32-bit instruction:

- `0b11101`.
- `0b11110`.
- `0b11111`.

Otherwise, the halfword is a 16-bit instruction.

For more information on T32 instruction encodings see *About the T32 and A32 instruction encodings* on page F2-2454.

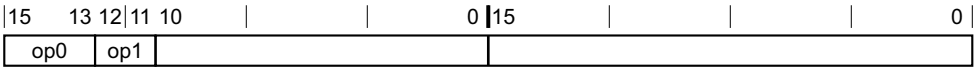



**Table F3-1 Main encoding table for the T32 instruction set**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| **op0** | **op1** | |
| != 111 | - | *16-bit T32 instruction encoding* on page F3-2465 |
| 111 | 00 | B - T2 variant |
| 111 | != 00 | *32-bit T32 instruction encoding* on page F3-2476 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F3.2 16-bit T32 instruction encoding

This section describes the encoding of the 16-bit T32 instruction encoding group. This section is decoded from *Top level T32 instruction set encoding* on page F3-2464.

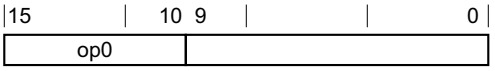



**Table F3-2 Encoding table for the 16-bit group**

| Decode fields | Decode group or instruction page |
|---|---|
| **op0** | |
| 00xxxx | *Shift (immediate), add, subtract, move, and compare* |
| 010000 | *Data-processing (two low registers)* on page F3-2467 |
| 010001 | *Special data instructions and branch and exchange* on page F3-2468 |
| 01001x | LDR (literal) - T1 variant |
| 0101xx | *Load/store (register offset)* on page F3-2469 |
| 011xxx | *Load/store word/byte (immediate offset)* on page F3-2469 |
| 1000xx | *Load/store halfword (immediate offset)* on page F3-2470 |
| 1001xx | *Load/store (SP-relative)* on page F3-2470 |
| 1010xx | *Add PC/SP (immediate)* on page F3-2471 |
| 1011xx | *Miscellaneous 16-bit instructions* on page F3-2471 |
| 1100xx | *Load/store multiple* on page F3-2474 |
| 1101xx | *Conditional branch, and Supervisor Call* on page F3-2474 |

### F3.2.1 Shift (immediate), add, subtract, move, and compare

This section describes the encoding of the Shift (immediate), add, subtract, move, and compare group. This section is decoded from *16-bit T32 instruction encoding*.

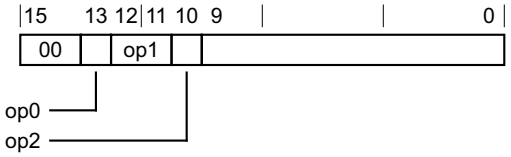



**Table F3-3 Encoding table for the Shift (immediate), add, subtract, move, and compare group**

| Decode fields | | | Decode group or instruction page |
|---|---|---|---|
| **op0** | **op1** | **op2** | |
| 0 | 11 | 0 | *Add, subtract (three low registers)* on page F3-2466 |

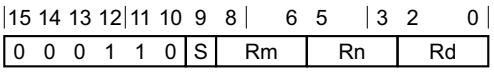

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F3-3 Encoding table for the Shift (immediate), add, subtract, move, and compare group**

| Decode fields | | | Decode group or instruction page |
|---|---|---|---|
| op0 | op1 | op2 | |
| 0 | 11 | 1 | *Add, subtract (two low registers and immediate)* |
| 0 | != 11 | - | MOV, MOVS (register) - T2 variant |
| 1 | - | - | *Add, subtract, compare, move (one low register and immediate)* |

## Add, subtract (three low registers)

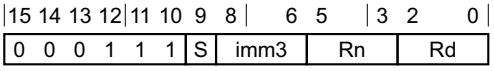
This section describes the encoding of the Add, subtract (three low registers) instruction class. This section is decoded from *Shift (immediate), add, subtract, move, and compare* on page F3-2465.




| Decode fields | Instruction Page |
|---|---|
| S | |
| 0 | ADD, ADDS (register) |
| 1 | SUB, SUBS (register) |

## Add, subtract (two low registers and immediate)

This section describes the encoding of the Add, subtract (two low registers and immediate) instruction class. This section is decoded from *Shift (immediate), add, subtract, move, and compare* on page F3-2465.

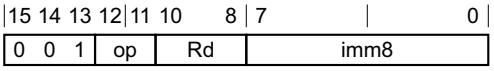



| Decode fields | Instruction Page |
|---|---|
| S | |
| 0 | ADD, ADDS (immediate) |
| 1 | SUB, SUBS (immediate) |

## Add, subtract, compare, move (one low register and immediate)

This section describes the encoding of the Add, subtract, compare, move (one low register and immediate) instruction class. This section is decoded from *Shift (immediate), add, subtract, move, and compare* on page F3-2465.

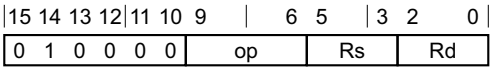

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*




| Decode fields | Instruction Page |
|---|---|
| op | |
| 00 | MOV, MOVS (immediate) |
| 01 | CMP (immediate) |
| 10 | ADD, ADDS (immediate) |
| 11 | SUB, SUBS (immediate) |

### F3.2.2 Data-processing (two low registers)

This section describes the encoding of the Data-processing (two low registers) instruction class. This section is decoded from *16-bit T32 instruction encoding* on page F3-2465.




| Decode fields | Instruction Page |
|---|---|
| op | |
| 0000 | AND, ANDS (register) |
| 0001 | EOR, EORS (register) |
| 0010 | MOV, MOVS (register-shifted register) - Logical shift left variant |
| 0011 | MOV, MOVS (register-shifted register) - Logical shift right variant |
| 0100 | MOV, MOVS (register-shifted register) - Arithmetic shift right variant |
| 0101 | ADC, ADCS (register) |
| 0110 | SBC, SBCS (register) |
| 0111 | MOV, MOVS (register-shifted register) - Rotate right variant |
| 1000 | TST (register) |
| 1001 | RSB, RSBS (immediate) |
| 1010 | CMP (register) |
| 1011 | CMN (register) |
| 1100 | ORR, ORRS (register) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields<br>op | Instruction Page |
|---|---|
| 1101 | MUL, MULS |
| 1110 | BIC, BICS (register) |
| 1111 | MVN, MVNS (register) |

### F3.2.3 Special data instructions and branch and exchange

This section describes the encoding of the Special data instructions and branch and exchange group. This section is decoded from *16-bit T32 instruction encoding* on page F3-2465.

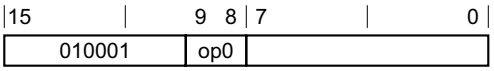
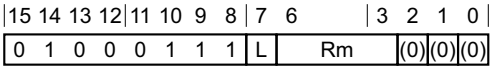



**Table F3-4 Encoding table for the Special data instructions and branch and exchange group**

| Decode fields<br>op0 | Decode group or instruction page |
|---|---|
| 11 | *Branch and exchange* |
| != 11 | *Add, subtract, compare, move (two high registers)* |

#### Branch and exchange

This section describes the encoding of the Branch and exchange instruction class. This section is decoded from *Special data instructions and branch and exchange*.

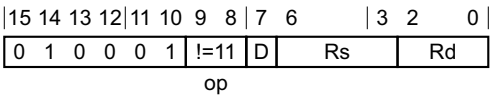



| Decode fields<br>L | Instruction Page |
|---|---|
| 0 | BX |
| 1 | BLX (register) |

#### Add, subtract, compare, move (two high registers)

This section describes the encoding of the Add, subtract, compare, move (two high registers) instruction class. This section is decoded from *Special data instructions and branch and exchange*.

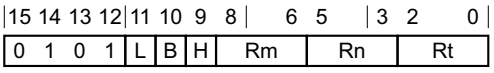

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| op | D:Rd | Rs | |
| 00 | != 1101 | != 1101 | ADD, ADDS (register) |
| 00 | - | 1101 | ADD, ADDS (SP plus register) - T1 variant |
| 00 | 1101 | != 1101 | ADD, ADDS (SP plus register) - T2 variant |
| 01 | - | - | CMP (register) |
| 10 | - | - | MOV, MOVS (register) |

### F3.2.4 Load/store (register offset)

This section describes the encoding of the Load/store (register offset) instruction class. This section is decoded from *16-bit T32 instruction encoding* on page F3-2465.

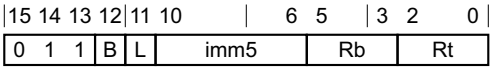



| Decode fields | | | Instruction Page |
|---|---|---|---|
| L | B | H | |
| 0 | 0 | 0 | STR (register) |
| 0 | 0 | 1 | STRH (register) |
| 0 | 1 | 0 | STRB (register) |
| 0 | 1 | 1 | LDRSB (register) |
| 1 | 0 | 0 | LDR (register) |
| 1 | 0 | 1 | LDRH (register) |
| 1 | 1 | 0 | LDRB (register) |
| 1 | 1 | 1 | LDRSH (register) |

### F3.2.5 Load/store word/byte (immediate offset)

This section describes the encoding of the Load/store word/byte (immediate offset) instruction class. This section is decoded from *16-bit T32 instruction encoding* on page F3-2465.

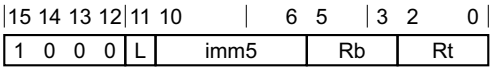

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | Instruction Page |
|---|---|---|
| B | L | |
| 0 | 0 | STR (immediate) |
| 0 | 1 | LDR (immediate) |
| 1 | 0 | STRB (immediate) |
| 1 | 1 | LDRB (immediate) |

### F3.2.6 Load/store halfword (immediate offset)

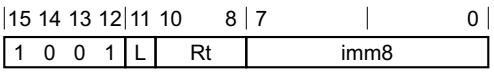
This section describes the encoding of the Load/store halfword (immediate offset) instruction class. This section is decoded from *16-bit T32 instruction encoding* on page F3-2465.




| Decode fields | Instruction Page |
|---|---|
| L | |
| 0 | STRH (immediate) |
| 1 | LDRH (immediate) |

### F3.2.7 Load/store (SP-relative)

This section describes the encoding of the Load/store (SP-relative) instruction class. This section is decoded from *16-bit T32 instruction encoding* on page F3-2465.

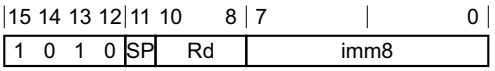



| Decode fields | Instruction Page |
|---|---|
| L | |
| 0 | STR (immediate) |
| 1 | LDR (immediate) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F3.2.8 Add PC/SP (immediate)

This section describes the encoding of the Add PC/SP (immediate) instruction class. This section is decoded from *16-bit T32 instruction encoding* on page F3-2465.




| Decode fields | Instruction Page |
|---|---|
| SP | |
| 0 | ADR |
| 1 | ADD, ADDS (SP plus immediate) |

### F3.2.9 Miscellaneous 16-bit instructions

This section describes the encoding of the Miscellaneous 16-bit instructions group. This section is decoded from *16-bit T32 instruction encoding* on page F3-2465.

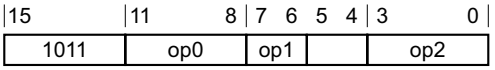


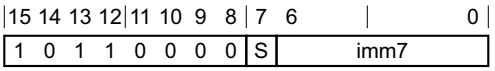

**Table F3-5 Encoding table for the Miscellaneous 16-bit instructions group**

| Decode fields | | | Decode group or instruction page |
|---|---|---|---|
| op0 | op1 | op2 | |
| 0000 | - | - | *Adjust SP (immediate)* on page F3-2472 |
| 0010 | - | - | *Extend* on page F3-2472 |
| 0110 | 00 | - | Unallocated. |
| 0110 | 01 | - | *Change Processor State* on page F3-2472 |
| 0110 | 1x | - | Unallocated. |
| 0111 | - | - | Unallocated. |
| 1000 | - | - | Unallocated. |
| 1010 | 10 | - | HLT |
| 1010 | != 10 | - | *Reverse bytes* on page F3-2473 |
| 1110 | - | - | BKPT |
| 1111 | - | 0000 | *Hints* on page F3-2473 |
| 1111 | - | != 0000 | IT |
| x0x1 | - | - | CBNZ, CBZ |
| x10x | - | - | *Push and Pop* on page F3-2473 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Adjust SP (immediate)

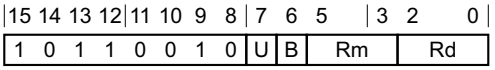
This section describes the encoding of the Adjust SP (immediate) instruction class. This section is decoded from *Miscellaneous 16-bit instructions* on page F3-2471.




| Decode fields | Instruction Page |
|---|---|
| S | |
| 0 | ADD, ADDS (SP plus immediate) |
| 1 | SUB, SUBS (SP minus immediate) |

### Extend

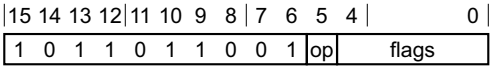
This section describes the encoding of the Extend instruction class. This section is decoded from *Miscellaneous 16-bit instructions* on page F3-2471.




| Decode fields | | Instruction Page |
|---|---|---|
| U | B | |
| 0 | 0 | SXTH |
| 0 | 1 | SXTB |
| 1 | 0 | UXTH |
| 1 | 1 | UXTB |

### Change Processor State

This section describes the encoding of the Change Processor State instruction class. This section is decoded from *Miscellaneous 16-bit instructions* on page F3-2471.

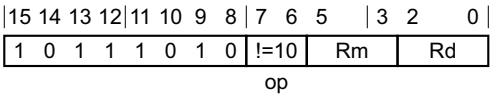



| Decode fields | | Instruction Page |
|---|---|---|
| op | flags | |
| 0 | - | SETEND |
| 1 | - | CPS, CPSID, CPSIE |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Reverse bytes

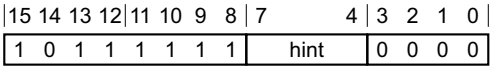
This section describes the encoding of the Reverse bytes instruction class. This section is decoded from *Miscellaneous 16-bit instructions* on page F3-2471.




| Decode fields | Instruction Page |
|---|---|
| op | |
| 00 | REV |
| 01 | REV16 |
| 11 | REVSH |

### Hints

This section describes the encoding of the Hints instruction class. This section is decoded from *Miscellaneous 16-bit instructions* on page F3-2471.

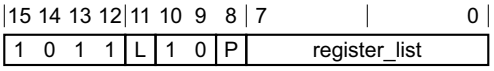



| Decode fields | Instruction Page |
|---|---|
| hint | |
| 0000 | NOP |
| 0001 | YIELD |
| 0010 | WFE |
| 0011 | WFI |
| 0100 | SEV |
| 0101 | SEVL |
| 011x | Unallocated. |
| 1xxx | Unallocated. |

### Push and Pop

This section describes the encoding of the Push and Pop instruction class. This section is decoded from *Miscellaneous 16-bit instructions* on page F3-2471.

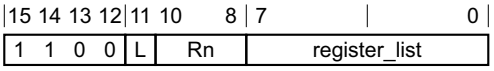

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | Instruction Page |
|---|---|
| L | |
| 0 | PUSH |
| 1 | POP |

### F3.2.10 Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. This section is decoded from *16-bit T32 instruction encoding* on page F3-2465.




| Decode fields | Instruction Page |
|---|---|
| L | |
| 0 | STM, STMIA, STMEA |
| 1 | LDM, LDMIA, LDMFD |

### F3.2.11 Conditional branch, and Supervisor Call

This section describes the encoding of the Conditional branch, and Supervisor Call group. This section is decoded from *16-bit T32 instruction encoding* on page F3-2465.

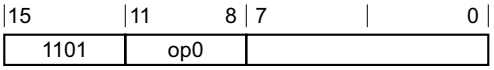


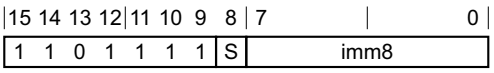

**Table F3-6 Encoding table for the Conditional branch, and Supervisor Call group**

| Decode fields | Decode group or instruction page |
|---|---|
| op0 | |
| 111x | *Exception generation* |
| != 111x | B - T1 variant |

#### Exception generation

This section describes the encoding of the Exception generation instruction class. This section is decoded from *Conditional branch, and Supervisor Call*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | Instruction Page |
|---|---|
| S | |
| 0 | UDF |
| 1 | SVC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F3.3 32-bit T32 instruction encoding

This section describes the encoding of the 32-bit T32 instruction encoding group. This section is decoded from *Top level T32 instruction set encoding* on page F3-2464.

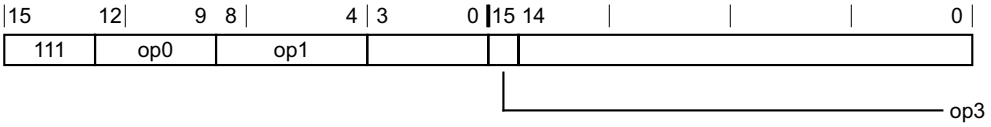



**Table F3-7 Encoding table for the 32-bit group**

| Decode fields | | | Decode group or instruction page |
|---|---|---|---|
| **op0** | **op1** | **op3** | |
| x11x | - | - | *Coprocessor, Advanced SIMD, and floating-point instructions* |
| 0100 | xx0xx | - | *Load/store multiple* on page F3-2479 |
| 0100 | xx1xx | - | *Load/store dual, load/store exclusive, table branch* on page F3-2480 |
| 0101 | - | - | *Data-processing (shifted register)* on page F3-2483 |
| 10xx | - | 1 | *Branches and miscellaneous control* on page F3-2485 |
| 10x0 | - | 0 | *Data-processing (modified immediate)* on page F3-2489 |
| 10x1 | - | 0 | *Data-processing (plain binary immediate)* on page F3-2490 |
| 1100 | 1xxx0 | - | For more information about this encoding see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*. |
| 1100 | != 1xxx0 | - | *Load/store single* on page F3-2492 |
| 1101 | 0xxxx | - | *Data-processing (register)* on page F3-2497 |
| 1101 | 10xxx | - | *Multiply, multiply accumulate, and absolute difference* on page F3-2501 |
| 1101 | 11xxx | - | *Long multiply and divide* on page F3-2503 |

### F3.3.1 Coprocessor, Advanced SIMD, and floating-point instructions

This section describes the encoding of the Coprocessor, Advanced SIMD, and floating-point instructions group. This section is decoded from *32-bit T32 instruction encoding*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

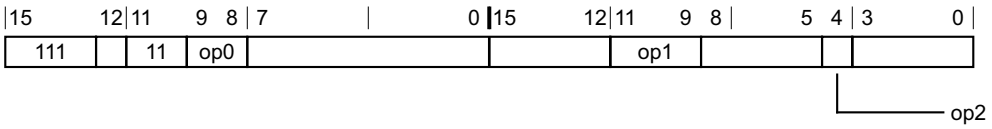



**Table F3-8 Encoding table for the Coprocessor, Advanced SIMD, and floating-point instructions group**

| Decode fields | | | Decode group or instruction page |
|---|---|---|---|
| **op0** | **op1** | **op2** | |
| 0x | 101 | - | For more information about this encoding see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*. |
| 10 | 101 | 0 | For more information about this encoding see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*. |
| 10 | 101 | 1 | For more information about this encoding see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*. |
| 11 | - | - | For more information about this encoding see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*. |
| != 11 | != 101 | - | *Coprocessor* |

## Coprocessor

This section describes the encoding of the Coprocessor group. This section is decoded from *Coprocessor, Advanced SIMD, and floating-point instructions* on page F3-2476.

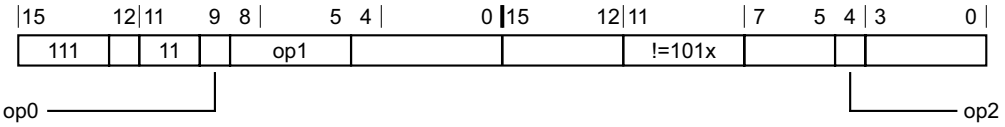


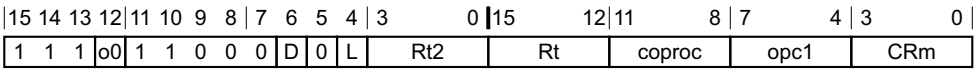

**Table F3-9 Encoding table for the Coprocessor group**

| Decode fields | | | Decode group or instruction page |
|---|---|---|---|
| **op0** | **op1** | **op2** | |
| 0 | 00x0 | - | *Coprocessor 64-bit move* |
| 0 | != 00x0 | - | *Coprocessor load/store registers* on page F3-2478 |
| 1 | 0xxx | 0 | CDP, CDP2 |
| 1 | 0xxx | 1 | *Coprocessor 32-bit move* on page F3-2479 |

### *Coprocessor 64-bit move*

This section describes the encoding of the Coprocessor 64-bit move instruction class. This section is decoded from *Coprocessor*.

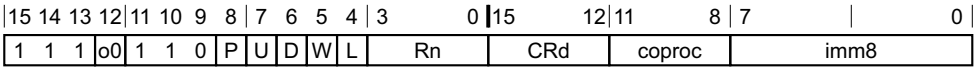

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| o0 | D | L | |
| 0 | 0 | - | Unallocated. |
| 0 | 1 | 0 | MCRR, MCRR2 - T1 variant |
| 0 | 1 | 1 | MRRC, MRRC2 - T1 variant |
| 1 | 0 | - | Unallocated. |
| 1 | 1 | 0 | MCRR, MCRR2 - T2 variant |
| 1 | 1 | 1 | MRRC, MRRC2 - T2 variant |

### *Coprocessor load/store registers*

This section describes the encoding of the Coprocessor load/store registers instruction class. This section is decoded from *Coprocessor* on page F3-2477.



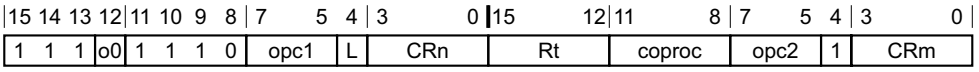

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| o0 | P:U:W | L | Rn | |
| 0 | != 000 | 1 | 1111 | LDC, LDC2 (literal) - T1 variant |
| 0 | 0x1 | 0 | - | STC, STC2 |
| 0 | 0x1 | 1 | != 1111 | LDC, LDC2 (immediate) |
| 0 | 010 | 0 | - | STC, STC2 |
| 0 | 010 | 1 | != 1111 | LDC, LDC2 (immediate) |
| 0 | 1x0 | 0 | - | STC, STC2 |
| 0 | 1x0 | 1 | != 1111 | LDC, LDC2 (immediate) |
| 0 | 1x1 | 0 | - | STC, STC2 |
| 0 | 1x1 | 1 | != 1111 | LDC, LDC2 (immediate) |
| 1 | != 000 | 1 | 1111 | LDC, LDC2 (literal) - T2 variant |
| 1 | 0x1 | 0 | - | STC, STC2 |
| 1 | 0x1 | 1 | != 1111 | LDC, LDC2 (immediate) |
| 1 | 010 | 0 | - | STC, STC2 |
| 1 | 010 | 1 | != 1111 | LDC, LDC2 (immediate) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| o0 | P:U:W | L | Rn | |
| 1 | 1x0 | 0 | - | STC, STC2 |
| 1 | 1x0 | 1 | != 1111 | LDC, LDC2 (immediate) |
| 1 | 1x1 | 0 | - | STC, STC2 |
| 1 | 1x1 | 1 | != 1111 | LDC, LDC2 (immediate) |

#### *Coprocessor 32-bit move*

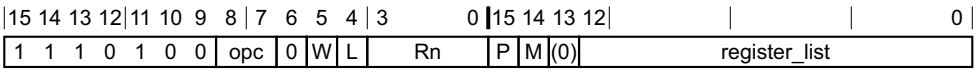
This section describes the encoding of the Coprocessor 32-bit move instruction class. This section is decoded from *Coprocessor* on page F3-2477.




| Decode fields | | Instruction Page |
|---|---|---|
| o0 | L | |
| 0 | 0 | MCR, MCR2 - T1 variant |
| 0 | 1 | MRC, MRC2 - T1 variant |
| 1 | 0 | MCR, MCR2 - T2 variant |
| 1 | 1 | MRC, MRC2 - T2 variant |

### F3.3.2 Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. This section is decoded from *32-bit T32 instruction encoding* on page F3-2476.




| Decode fields | | Instruction Page |
|---|---|---|
| opc | L | |
| 00 | 0 | SRS, SRSDA, SRSDB, SRSIA, SRSIB - T1 variant |
| 00 | 1 | RFE, RFEDA, RFEDB, RFEIA, RFEIB - T1 variant |
| 01 | 0 | STM, STMIA, STMEA |
| 01 | 1 | LDM, LDMIA, LDMFD |
| 10 | 0 | STMDB, STMFD |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | Instruction Page |
|---|---|---|
| opc | L | |
| 10 | 1 | LDMDB, LDMEA |
| 11 | 0 | SRS, SRSDA, SRSDB, SRSIA, SRSIB - T2 variant |
| 11 | 1 | RFE, RFEDA, RFEDB, RFEIA, RFEIB - T2 variant |

### F3.3.3 Load/store dual, load/store exclusive, table branch

This section describes the encoding of the Load/store dual, load/store exclusive, table branch group. This section is decoded from *32-bit T32 instruction encoding* on page F3-2476.

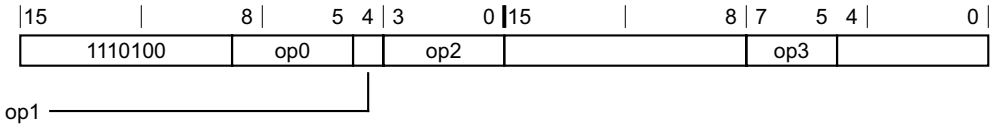


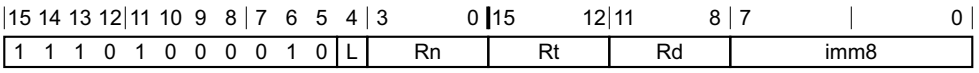

**Table F3-10 Encoding table for the Load/store dual, load/store exclusive, table branch group**

| Decode fields | | | | Decode group or instruction page |
|---|---|---|---|---|
| op0 | op1 | op2 | op3 | |
| 0010 | - | - | - | *Load/store exclusive* |
| 0110 | 0 | - | 000 | Unallocated. |
| 0110 | 1 | - | 000 | TBB, TBH |
| 0110 | - | - | 01x | *Load/store exclusive byte/half/dual* on page F3-2481 |
| 0110 | - | - | 1xx | *Load-acquire / Store-release* on page F3-2481 |
| 0x11 | - | != 1111 | - | *Load/store dual (immediate, post-indexed)* on page F3-2482 |
| 1x10 | - | != 1111 | - | *Load/store dual (immediate)* on page F3-2483 |
| 1x11 | - | != 1111 | - | *Load/store dual (immediate, pre-indexed)* on page F3-2483 |
| != 0xx0 | - | 1111 | - | LDRD (literal) |

#### Load/store exclusive

This section describes the encoding of the Load/store exclusive instruction class. This section is decoded from *Load/store dual, load/store exclusive, table branch*.

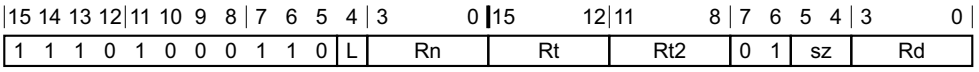

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | Instruction Page |
|---|---|
| L | |
| 0 | STREX |
| 1 | LDREX |

### Load/store exclusive byte/half/dual

This section describes the encoding of the Load/store exclusive byte/half/dual instruction class. This section is decoded from *Load/store dual, load/store exclusive, table branch* on page F3-2480.

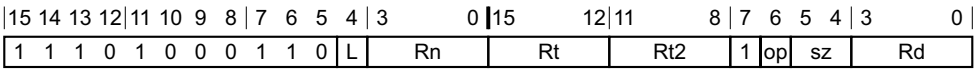



| Decode fields | | Instruction Page |
|---|---|---|
| L | sz | |
| 0 | 00 | STREXB |
| 0 | 01 | STREXH |
| 0 | 10 | Unallocated. |
| 0 | 11 | STREXD |
| 1 | 00 | LDREXB |
| 1 | 01 | LDREXH |
| 1 | 10 | Unallocated. |
| 1 | 11 | LDREXD |

### Load-acquire / Store-release

This section describes the encoding of the Load-acquire / Store-release instruction class. This section is decoded from *Load/store dual, load/store exclusive, table branch* on page F3-2480.

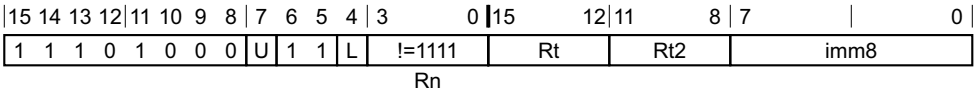

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| op | L | sz | |
| 0 | 0 | 00 | STLB |
| 0 | 0 | 01 | STLH |
| 0 | 0 | 10 | STL |
| 0 | 0 | 11 | Unallocated. |
| 0 | 1 | 00 | LDAB |
| 0 | 1 | 01 | LDAH |
| 0 | 1 | 10 | LDA |
| 0 | 1 | 11 | Unallocated. |
| 1 | 0 | 00 | STLEXB |
| 1 | 0 | 01 | STLEXH |
| 1 | 0 | 10 | STLEX |
| 1 | 0 | 11 | STLEXD |
| 1 | 1 | 00 | LDAEXB |
| 1 | 1 | 01 | LDAEXH |
| 1 | 1 | 10 | LDAEX |
| 1 | 1 | 11 | LDAEXD |

### Load/store dual (immediate, post-indexed)

This section describes the encoding of the Load/store dual (immediate, post-indexed) instruction class. This section is decoded from *Load/store dual, load/store exclusive, table branch* on page F3-2480.

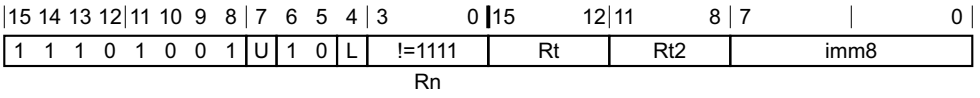



| Decode fields | | Instruction Page |
|---|---|---|
| L | | |
| 0 | | STRD (immediate) |
| 1 | | LDRD (immediate) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Load/store dual (immediate)

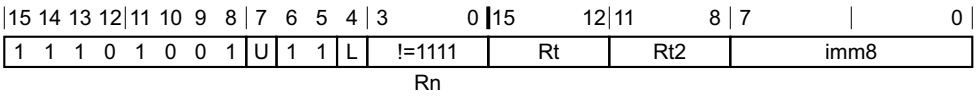
This section describes the encoding of the Load/store dual (immediate) instruction class. This section is decoded from *Load/store dual, load/store exclusive, table branch* on page F3-2480.




| Decode fields | Instruction Page |
|---|---|
| L | |
| 0 | STRD (immediate) |
| 1 | LDRD (immediate) |

### Load/store dual (immediate, pre-indexed)

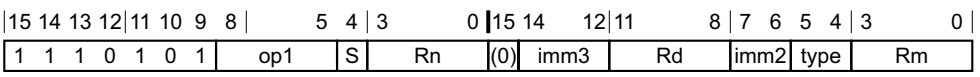
This section describes the encoding of the Load/store dual (immediate, pre-indexed) instruction class. This section is decoded from *Load/store dual, load/store exclusive, table branch* on page F3-2480.




| Decode fields | Instruction Page |
|---|---|
| L | |
| 0 | STRD (immediate) |
| 1 | LDRD (immediate) |

## F3.3.4 Data-processing (shifted register)

This section describes the encoding of the Data-processing (shifted register) instruction class. This section is decoded from *32-bit T32 instruction encoding* on page F3-2476.




| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| op1 | S | Rn | Rd | imm3:imm2:type | |
| 0000 | 0 | - | - | - | AND, ANDS (register) - AND, rotate right with extend variant |
| 0000 | 1 | - | != 1111 | != 0000011 | AND, ANDS (register) - ANDS, shift or rotate by value variant |
| 0000 | 1 | - | != 1111 | 0000011 | AND, ANDS (register) - ANDS, rotate right with extend variant |
| 0000 | 1 | - | 1111 | != 0000011 | TST (register) - Shift or rotate by value variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| op1 | S | Rn | Rd | imm3:imm2:type | |
| 0000 | 1 | - | 1111 | 0000011 | TST (register) - Rotate right with extend variant |
| 0001 | - | - | - | - | BIC, BICS (register) |
| 0010 | 0 | != 1111 | - | - | ORR, ORRS (register) - ORR, rotate right with extend variant |
| 0010 | 0 | 1111 | - | - | MOV, MOVS (register) - MOV, rotate right with extend variant |
| 0010 | 1 | != 1111 | - | - | ORR, ORRS (register) - ORRS, rotate right with extend variant |
| 0010 | 1 | 1111 | - | - | MOV, MOVS (register) - MOVS, rotate right with extend variant |
| 0011 | 0 | != 1111 | - | - | ORN, ORNS (register) - ORN, rotate right with extend variant |
| 0011 | 0 | 1111 | - | - | MVN, MVNS (register) - MVN, rotate right with extend variant |
| 0011 | 1 | != 1111 | - | - | ORN, ORNS (register) - ORNS, rotate right with extend variant |
| 0011 | 1 | 1111 | - | - | MVN, MVNS (register) - MVNS, rotate right with extend variant |
| 0100 | 0 | - | - | - | EOR, EORS (register) - EOR, rotate right with extend variant |
| 0100 | 1 | - | != 1111 | != 0000011 | EOR, EORS (register) - EORS, shift or rotate by value variant |
| 0100 | 1 | - | != 1111 | 0000011 | EOR, EORS (register) - EORS, rotate right with extend variant |
| 0100 | 1 | - | 1111 | != 0000011 | TEQ (register) - Shift or rotate by value variant |
| 0100 | 1 | - | 1111 | 0000011 | TEQ (register) - Rotate right with extend variant |
| 0101 | - | - | - | - | Unallocated. |
| 0110 | 0 | - | - | xxxxx00 | PKHBT, PKHTB - PKHBT variant |
| 0110 | 0 | - | - | xxxxx01 | Unallocated. |
| 0110 | 0 | - | - | xxxxx10 | PKHBT, PKHTB - PKHTB variant |
| 0110 | 0 | - | - | xxxxx11 | Unallocated. |
| 0111 | - | - | - | - | Unallocated. |
| 1000 | 0 | != 1101 | - | - | ADD, ADDS (register) - ADD, rotate right with extend variant |
| 1000 | 0 | 1101 | - | - | ADD, ADDS (SP plus register) - ADD, rotate right with extend variant |
| 1000 | 1 | != 1101 | != 1111 | - | ADD, ADDS (register) - ADDS, rotate right with extend variant |
| 1000 | 1 | 1101 | != 1111 | - | ADD, ADDS (SP plus register) - ADDS, rotate right with extend variant |
| 1000 | 1 | - | 1111 | - | CMN (register) |
| 1001 | - | - | - | - | Unallocated. |
| 1010 | - | - | - | - | ADC, ADCS (register) |
| 1011 | - | - | - | - | SBC, SBCS (register) |
| 1100 | - | - | - | - | Unallocated. |
| 1101 | 0 | != 1101 | - | - | SUB, SUBS (register) - SUB, rotate right with extend variant |
| 1101 | 0 | 1101 | - | - | SUB, SUBS (SP minus register) - SUB, rotate right with extend variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| op1 | S | Rn | Rd | imm3:imm2:type | |
| 1101 | 1 | != 1101 | != 1111 | - | SUB, SUBS (register) - SUBS, rotate right with extend variant |
| 1101 | 1 | 1101 | != 1111 | - | SUB, SUBS (SP minus register) - SUBS, rotate right with extend variant |
| 1101 | 1 | - | 1111 | - | CMP (register) |
| 1110 | - | - | - | - | RSB, RSBS (register) |
| 1111 | - | - | - | - | Unallocated. |

### F3.3.5 Branches and miscellaneous control

This section describes the encoding of the Branches and miscellaneous control group. This section is decoded from *32-bit T32 instruction encoding* on page F3-2476.

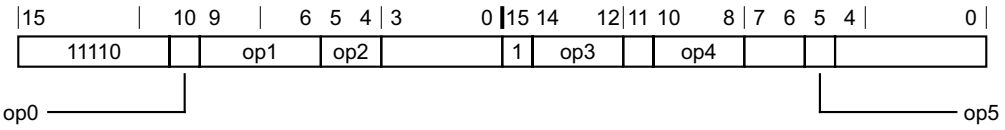



**Table F3-11 Encoding table for the Branches and miscellaneous control group**

| Decode fields | | | | | | Decode group or instruction page |
|---|---|---|---|---|---|---|
| op0 | op1 | op2 | op3 | op4 | op5 | |
| 0 | 1110 | 0x | 0x0 | - | 0 | MSR (register) |
| 0 | 1110 | 0x | 0x0 | - | 1 | MSR (Banked register) |
| 0 | 1110 | 10 | 0x0 | 000 | - | *Hints* on page F3-2486 |
| 0 | 1110 | 10 | 0x0 | != 000 | - | *Change processor state* on page F3-2486 |
| 0 | 1110 | 11 | 0x0 | - | - | *Miscellaneous system* on page F3-2487 |
| 0 | 1111 | 00 | 0x0 | - | - | BXJ |
| 0 | 1111 | 01 | 0x0 | - | - | *Exception return* on page F3-2487 |
| 0 | 1111 | 1x | 0x0 | - | 0 | MRS |
| 0 | 1111 | 1x | 0x0 | - | 1 | MRS (Banked register) |
| 1 | 1110 | 00 | 000 | - | - | *DCPS* on page F3-2488 |
| 1 | 1110 | 00 | 010 | - | - | Unallocated. |
| 1 | 1110 | 01 | 0x0 | - | - | Unallocated. |
| 1 | 1110 | 1x | 0x0 | - | - | Unallocated. |
| 1 | 1111 | 0x | 0x0 | - | - | Unallocated. |
| 1 | 1111 | 1x | 0x0 | - | - | *Exception generation* on page F3-2488 |
| - | != 111x | - | 0x0 | - | - | B - T3 variant |

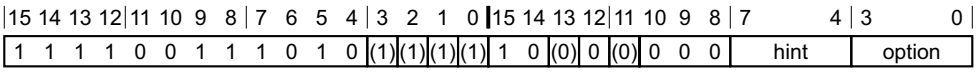

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F3-11 Encoding table for the Branches and miscellaneous control group (continued)**

| Decode fields | | | | | | Decode group or instruction page |
|---|---|---|---|---|---|---|
| op0 | op1 | op2 | op3 | op4 | op5 | |
| - | - | - | 0x1 | - | - | B - T4 variant |
| - | - | - | 1x0 | - | - | BL, BLX (immediate) - T2 variant |
| - | - | - | 1x1 | - | - | BL, BLX (immediate) - T1 variant |

## Hints

This section describes the encoding of the Hints instruction class. This section is decoded from *Branches and miscellaneous control* on page F3-2485.

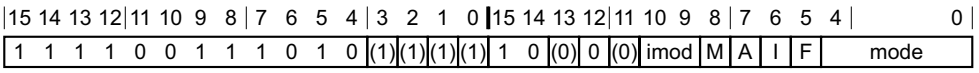



| Decode fields | | Instruction Page |
|---|---|---|
| hint | option | |
| 0000 | 0000 | NOP |
| 0000 | 0001 | YIELD |
| 0000 | 0010 | WFE |
| 0000 | 0011 | WFI |
| 0000 | 0100 | SEV |
| 0000 | 0101 | SEVL |
| 0000 | 011x | Unallocated. |
| 0000 | 1xxx | Unallocated. |
| 0001 | - | Unallocated. |
| 001x | - | Unallocated. |
| 01xx | - | Unallocated. |
| 10xx | - | Unallocated. |
| 110x | - | Unallocated. |
| 1110 | - | Unallocated. |
| 1111 | - | DBG |

## Change processor state

This section describes the encoding of the Change processor state instruction class. This section is decoded from *Branches and miscellaneous control* on page F3-2485.

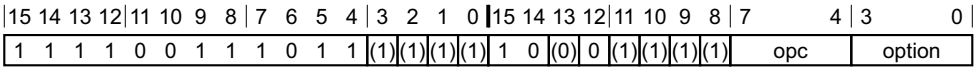

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | Instruction Page |
|---|---|---|
| imod | M | |
| 00 | 1 | CPS, CPSID, CPSIE - CPS variant |
| 01 | - | Unallocated. |
| 10 | - | CPS, CPSID, CPSIE - CPSIE variant |
| 11 | - | CPS, CPSID, CPSIE - CPSID variant |

### Miscellaneous system

This section describes the encoding of the Miscellaneous system instruction class. This section is decoded from *Branches and miscellaneous control* on page F3-2485.

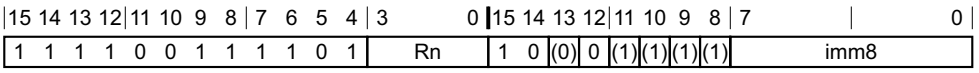



| Decode fields | Instruction Page |
|---|---|
| opc | |
| 000x | Unallocated. |
| 0010 | CLREX |
| 0011 | Unallocated. |
| 0100 | DSB |
| 0101 | DMB |
| 0110 | ISB |
| 0111 | Unallocated. |
| 1xxx | Unallocated. |

### Exception return

This section describes the encoding of the Exception return instruction class. This section is decoded from *Branches and miscellaneous control* on page F3-2485.

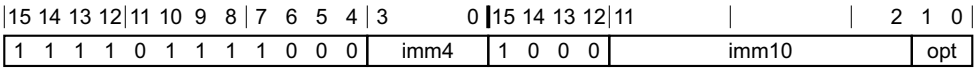

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | Instruction Page |
|---|---|---|
| **Rn** | **imm8** | |
| - | != 00000000 | SUB, SUBS (immediate) |
| 1110 | 00000000 | ERET |

### DCPS

This section describes the encoding of the DCPS instruction class. This section is decoded from *Branches and miscellaneous control* on page F3-2485.

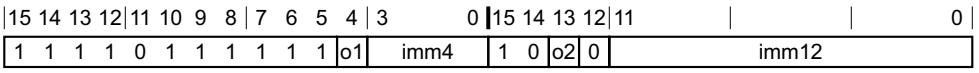



| Decode fields | | | Instruction Page |
|---|---|---|---|
| **imm4** | **imm10** | **opt** | |
| != 1111 | - | - | Unallocated. |
| 1111 | != 0000000000 | - | Unallocated. |
| 1111 | 0000000000 | 00 | Unallocated. |
| 1111 | 0000000000 | 01 | DCPS1, DCPS2, DCPS3 - DCPS1 variant |
| 1111 | 0000000000 | 10 | DCPS1, DCPS2, DCPS3 - DCPS2 variant |
| 1111 | 0000000000 | 11 | DCPS1, DCPS2, DCPS3 - DCPS3 variant |

### Exception generation

This section describes the encoding of the Exception generation instruction class. This section is decoded from *Branches and miscellaneous control* on page F3-2485.

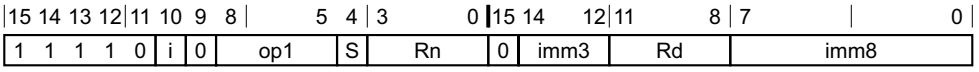

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | Instruction Page |
|---|---|---|
| o1 | o2 | |
| 0 | 0 | HVC |
| 0 | 1 | Unallocated. |
| 1 | 0 | SMC |
| 1 | 1 | UDF |

### F3.3.6 Data-processing (modified immediate)

This section describes the encoding of the Data-processing (modified immediate) instruction class. This section is decoded from *32-bit T32 instruction encoding* on page F3-2476.




| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| op1 | S | Rn | Rd | |
| 0000 | 0 | - | - | AND, ANDS (immediate) - AND variant |
| 0000 | 1 | - | != 1111 | AND, ANDS (immediate) - ANDS variant |
| 0000 | 1 | - | 1111 | TST (immediate) |
| 0001 | - | - | - | BIC, BICS (immediate) |
| 0010 | 0 | != 1111 | - | ORR, ORRS (immediate) - ORR variant |
| 0010 | 0 | 1111 | - | MOV, MOVS (immediate) - MOV variant |
| 0010 | 1 | != 1111 | - | ORR, ORRS (immediate) - ORRS variant |
| 0010 | 1 | 1111 | - | MOV, MOVS (immediate) - MOVS variant |
| 0011 | 0 | != 1111 | - | ORN, ORNS (immediate) - Not flag setting variant |
| 0011 | 0 | 1111 | - | MVN, MVNS (immediate) - MVN variant |
| 0011 | 1 | != 1111 | - | ORN, ORNS (immediate) - Flag setting variant |
| 0011 | 1 | 1111 | - | MVN, MVNS (immediate) - MVNS variant |
| 0100 | 0 | - | - | EOR, EORS (immediate) - EOR variant |
| 0100 | 1 | - | != 1111 | EOR, EORS (immediate) - EORS variant |
| 0100 | 1 | - | 1111 | TEQ (immediate) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| op1 | S | Rn | Rd | |
| 0101 | - | - | - | Unallocated. |
| 011x | - | - | - | Unallocated. |
| 1000 | 0 | != 1101 | - | ADD, ADDS (immediate) - ADD variant |
| 1000 | 0 | 1101 | - | ADD, ADDS (SP plus immediate) - ADD variant |
| 1000 | 1 | != 1101 | != 1111 | ADD, ADDS (immediate) - ADDS variant |
| 1000 | 1 | 1101 | != 1111 | ADD, ADDS (SP plus immediate) - ADDS variant |
| 1000 | 1 | - | 1111 | CMN (immediate) |
| 1001 | - | - | - | Unallocated. |
| 1010 | - | - | - | ADC, ADCS (immediate) |
| 1011 | - | - | - | SBC, SBCS (immediate) |
| 1100 | - | - | - | Unallocated. |
| 1101 | 0 | != 1101 | - | SUB, SUBS (immediate) - SUB variant |
| 1101 | 0 | 1101 | - | SUB, SUBS (SP minus immediate) - SUB variant |
| 1101 | 1 | != 1101 | != 1111 | SUB, SUBS (immediate) - SUBS variant |
| 1101 | 1 | 1101 | != 1111 | SUB, SUBS (SP minus immediate) - SUBS variant |
| 1101 | 1 | - | 1111 | CMP (immediate) |
| 1110 | - | - | - | RSB, RSBS (immediate) |
| 1111 | - | - | - | Unallocated. |

### F3.3.7 Data-processing (plain binary immediate)

This section describes the encoding of the Data-processing (plain binary immediate) group. This section is decoded from *32-bit T32 instruction encoding* on page F3-2476.

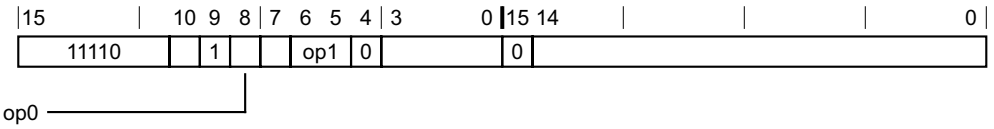



**Table F3-12 Encoding table for the Data-processing (plain binary immediate) group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 0 | 0x | *Data-processing (simple immediate)* on page F3-2491 |

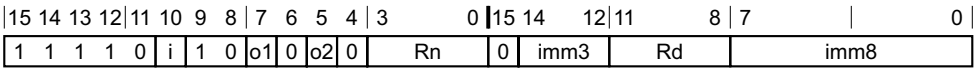

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F3-12 Encoding table for the Data-processing (plain binary immediate) group (continued)**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 0 | 10 | *Move Wide (16-bit immediate)* |
| 0 | 11 | Unallocated. |
| 1 | - | *Saturate, Bitfield* on page F3-2492 |

## Data-processing (simple immediate)

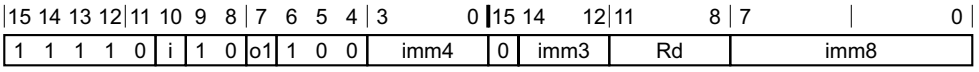
This section describes the encoding of the Data-processing (simple immediate) instruction class. This section is decoded from *Data-processing (plain binary immediate)* on page F3-2490.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| o1 | o2 | Rn | |
| 0 | 0 | != 11x1 | ADD, ADDS (immediate) |
| 0 | 0 | 1101 | ADD, ADDS (SP plus immediate) |
| 0 | 0 | 1111 | ADR - T3 variant |
| 0 | 1 | - | Unallocated. |
| 1 | 0 | - | Unallocated. |
| 1 | 1 | != 11x1 | SUB, SUBS (immediate) |
| 1 | 1 | 1101 | SUB, SUBS (SP minus immediate) |
| 1 | 1 | 1111 | ADR - T2 variant |

## Move Wide (16-bit immediate)

This section describes the encoding of the Move Wide (16-bit immediate) instruction class. This section is decoded from *Data-processing (plain binary immediate)* on page F3-2490.

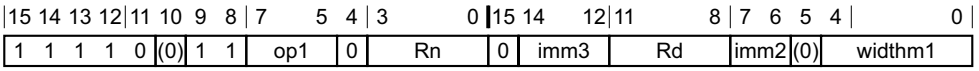



| Decode fields | Instruction Page |
|---|---|
| o1 | |
| 0 | MOV, MOVS (immediate) |
| 1 | MOVT |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Saturate, Bitfield

This section describes the encoding of the Saturate, Bitfield instruction class. This section is decoded from *Data-processing (plain binary immediate)* on page F3-2490.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| op1 | Rn | imm3:imm2 | |
| 000 | - | - | SSAT - Logical shift left variant |
| 001 | - | != 00000 | SSAT - Arithmetic shift right variant |
| 001 | - | 00000 | SSAT16 |
| 010 | - | - | SBFX |
| 011 | != 1111 | - | BFI |
| 011 | 1111 | - | BFC |
| 100 | - | - | USAT - Logical shift left variant |
| 101 | - | != 00000 | USAT - Arithmetic shift right variant |
| 101 | - | 00000 | USAT16 |
| 110 | - | - | UBFX |
| 111 | - | - | Unallocated. |

## F3.3.8 Load/store single

This section describes the encoding of the Load/store single group. This section is decoded from *32-bit T32 instruction encoding* on page F3-2476.

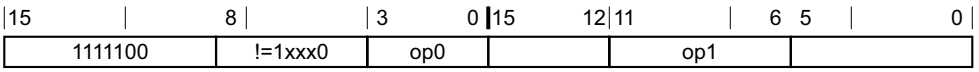



**Table F3-13 Encoding table for the Load/store single group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| != 1111 | 000000 | *Load/store (register offset)* on page F3-2493 |
| != 1111 | 10x0xx | Unallocated. |
| != 1111 | 10x1xx | *Load/store (immediate, post-indexed)* on page F3-2493 |
| != 1111 | 1100xx | *Load/store (negative immediate)* on page F3-2494 |
| != 1111 | 1110xx | *Load/store (unprivileged)* on page F3-2495 |

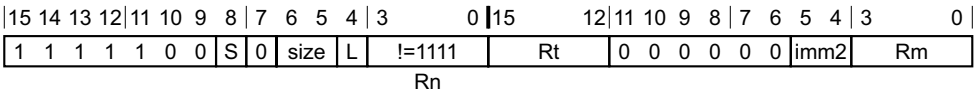

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F3-13 Encoding table for the Load/store single group (continued)**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| != 1111 | 11x1xx | *Load/store (immediate, pre-indexed)* on page F3-2495 |
| != 1111 | - | *Load/store (positive immediate)* on page F3-2496 |
| 1111 | - | *Load literal* on page F3-2497 |

## Load/store (register offset)

This section describes the encoding of the Load/store (register offset) instruction class. This section is decoded from *Load/store single* on page F3-2492.

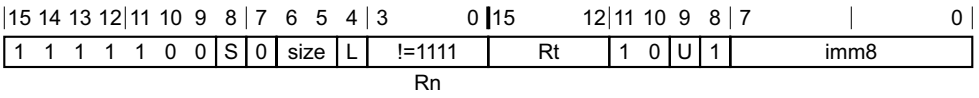



| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| S | size | L | Rt | |
| 0 | 00 | 0 | - | STRB (register) |
| 0 | 00 | 1 | != 1111 | LDRB (register) |
| 0 | 00 | 1 | 1111 | PLD, PLDW (register) - Preload read variant |
| 0 | 01 | 0 | - | STRH (register) |
| 0 | 01 | 1 | != 1111 | LDRH (register) |
| 0 | 01 | 1 | 1111 | PLD, PLDW (register) - Preload write variant |
| 0 | 10 | 0 | - | STR (register) |
| 0 | 10 | 1 | - | LDR (register) |
| 0 | 11 | - | - | Unallocated. |
| 1 | 00 | 1 | != 1111 | LDRSB (register) |
| 1 | 00 | 1 | 1111 | PLI (register) |
| 1 | 01 | 1 | != 1111 | LDRSH (register) |
| 1 | 1x | 1 | - | Unallocated. |

## Load/store (immediate, post-indexed)

This section describes the encoding of the Load/store (immediate, post-indexed) instruction class. This section is decoded from *Load/store single* on page F3-2492.

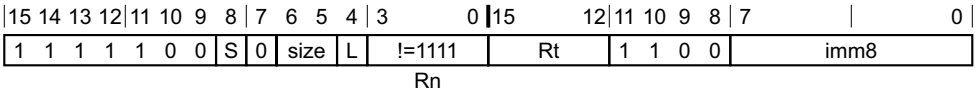

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| S | size | L | |
| 0 | 00 | 0 | STRB (immediate) |
| 0 | 00 | 1 | LDRB (immediate) |
| 0 | 01 | 0 | STRH (immediate) |
| 0 | 01 | 1 | LDRH (immediate) |
| 0 | 10 | 0 | STR (immediate) |
| 0 | 10 | 1 | LDR (immediate) |
| 0 | 11 | - | Unallocated. |
| 1 | 00 | 1 | LDRSB (immediate) |
| 1 | 01 | 1 | LDRSH (immediate) |
| 1 | 1x | 1 | Unallocated. |

### Load/store (negative immediate)

This section describes the encoding of the Load/store (negative immediate) instruction class. This section is decoded from *Load/store single* on page F3-2492.



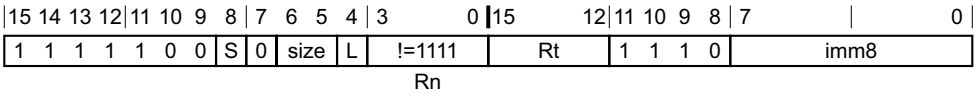

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| S | size | L | Rt | |
| 0 | 00 | 0 | - | STRB (immediate) |
| 0 | 00 | 1 | != 1111 | LDRB (immediate) |
| 0 | 00 | 1 | 1111 | PLD, PLDW (immediate) - Preload read variant |
| 0 | 01 | 0 | - | STRH (immediate) |
| 0 | 01 | 1 | != 1111 | LDRH (immediate) |
| 0 | 01 | 1 | 1111 | PLD, PLDW (immediate) - Preload write variant |
| 0 | 10 | 0 | - | STR (immediate) |
| 0 | 10 | 1 | - | LDR (immediate) |
| 0 | 11 | - | - | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| S | size | L | Rt | |
| 1 | 00 | 1 | - | LDRSB (immediate) |
| 1 | 00 | 1 | 1111 | PLI (immediate, literal) |
| 1 | 01 | 1 | - | LDRSH (immediate) |
| 1 | 1x | 1 | - | Unallocated. |

### Load/store (unprivileged)

This section describes the encoding of the Load/store (unprivileged) instruction class. This section is decoded from *Load/store single* on page F3-2492.

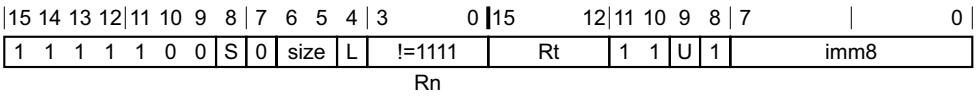



| Decode fields | | | Instruction Page |
|---|---|---|---|
| S | size | L | |
| 0 | 00 | 0 | STRBT |
| 0 | 00 | 1 | LDRBT |
| 0 | 01 | 0 | STRHT |
| 0 | 01 | 1 | LDRHT |
| 0 | 10 | 0 | STRT |
| 0 | 10 | 1 | LDRT |
| 0 | 11 | - | Unallocated. |
| 1 | 00 | 1 | LDRSBT |
| 1 | 01 | 1 | LDRSHT |
| 1 | 1x | 1 | Unallocated. |

### Load/store (immediate, pre-indexed)

This section describes the encoding of the Load/store (immediate, pre-indexed) instruction class. This section is decoded from *Load/store single* on page F3-2492.

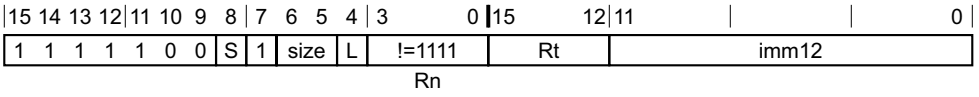

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| S | size | L | |
| 0 | 00 | 0 | STRB (immediate) |
| 0 | 00 | 1 | LDRB (immediate) |
| 0 | 01 | 0 | STRH (immediate) |
| 0 | 01 | 1 | LDRH (immediate) |
| 0 | 10 | 0 | STR (immediate) |
| 0 | 10 | 1 | LDR (immediate) |
| 0 | 11 | - | Unallocated. |
| 1 | 00 | 1 | LDRSB (immediate) |
| 1 | 01 | 1 | LDRSH (immediate) |
| 1 | 1x | 1 | Unallocated. |

### Load/store (positive immediate)

This section describes the encoding of the Load/store (positive immediate) instruction class. This section is decoded from *Load/store single* on page F3-2492.



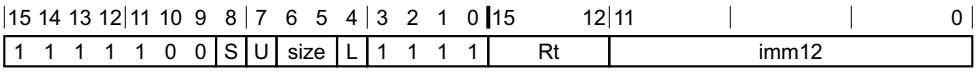

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| S | size | L | Rt | |
| 0 | 00 | 0 | - | STRB (immediate) |
| 0 | 00 | 1 | != 1111 | LDRB (immediate) |
| 0 | 00 | 1 | 1111 | PLD, PLDW (immediate) - Preload read variant |
| 0 | 01 | 0 | - | STRH (immediate) |
| 0 | 01 | 1 | != 1111 | LDRH (immediate) |
| 0 | 01 | 1 | 1111 | PLD, PLDW (immediate) - Preload write variant |
| 0 | 10 | 0 | - | STR (immediate) |
| 0 | 10 | 1 | - | LDR (immediate) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| S | size | L | Rt | |
| 1 | 00 | 1 | != 1111 | LDRSB (immediate) |
| 1 | 00 | 1 | 1111 | PLI (immediate, literal) |
| 1 | 01 | 1 | != 1111 | LDRSH (immediate) |

### Load literal

This section describes the encoding of the Load literal instruction class. This section is decoded from *Load/store single* on page F3-2492.




| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| S | size | L | Rt | |
| 0 | 0x | 1 | 1111 | PLD (literal) |
| 0 | 00 | 1 | != 1111 | LDRB (literal) |
| 0 | 01 | 1 | != 1111 | LDRH (literal) |
| 0 | 10 | 1 | - | LDR (literal) |
| 0 | 11 | - | - | Unallocated. |
| 1 | 00 | 1 | != 1111 | LDRSB (literal) |
| 1 | 00 | 1 | 1111 | PLI (immediate, literal) |
| 1 | 01 | 1 | != 1111 | LDRSH (literal) |
| 1 | 1x | 1 | - | Unallocated. |

## F3.3.9 Data-processing (register)

This section describes the encoding of the Data-processing (register) group. This section is decoded from *32-bit T32 instruction encoding* on page F3-2476.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

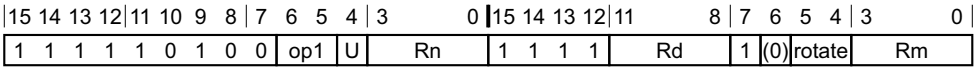



**Table F3-14 Encoding table for the Data-processing (register) group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| **op0** | **op1** | |
| 0 | 0000 | MOV, MOVS (register-shifted register) - Flag setting variant |
| 0 | 0001 | Unallocated. |
| 0 | 001x | Unallocated. |
| 0 | 01xx | Unallocated. |
| 0 | 1xxx | *Register extends* |
| 1 | 0xxx | *Parallel add-subtract* on page F3-2499 |
| 1 | 10xx | *Data-processing (two source registers)* on page F3-2500 |
| 1 | 11xx | Unallocated. |

## Register extends

This section describes the encoding of the Register extends instruction class. This section is decoded from *Data-processing (register)* on page F3-2497.



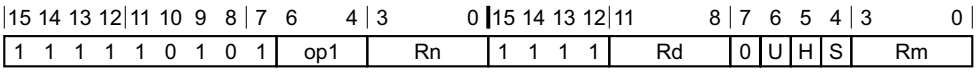

| Decode fields | | | Instruction Page |
|---|---|---|---|
| **op1** | **U** | **Rn** | |
| 00 | 0 | != 1111 | SXTAH |
| 00 | 0 | 1111 | SXTH |
| 00 | 1 | != 1111 | UXTAH |
| 00 | 1 | 1111 | UXTH |
| 01 | 0 | != 1111 | SXTAB16 |
| 01 | 0 | 1111 | SXTB16 |
| 01 | 1 | != 1111 | UXTAB16 |
| 01 | 1 | 1111 | UXTB16 |
| 10 | 0 | != 1111 | SXTAB |
| 10 | 0 | 1111 | SXTB |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| op1 | U | Rn | |
| 10 | 1 | != 1111 | UXTAB |
| 10 | 1 | 1111 | UXTB |
| 11 | - | - | Unallocated. |

### Parallel add-subtract

This section describes the encoding of the Parallel add-subtract instruction class. This section is decoded from *Data-processing (register)* on page F3-2497.




| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| op1 | U | H | S | |
| 000 | 0 | 0 | 0 | SADD8 |
| 000 | 0 | 0 | 1 | QADD8 |
| 000 | 0 | 1 | 0 | SHADD8 |
| 000 | 0 | 1 | 1 | Unallocated. |
| 000 | 1 | 0 | 0 | UADD8 |
| 000 | 1 | 0 | 1 | UQADD8 |
| 000 | 1 | 1 | 0 | UHADD8 |
| 000 | 1 | 1 | 1 | Unallocated. |
| 001 | 0 | 0 | 0 | SADD16 |
| 001 | 0 | 0 | 1 | QADD16 |
| 001 | 0 | 1 | 0 | SHADD16 |
| 001 | 0 | 1 | 1 | Unallocated. |
| 001 | 1 | 0 | 0 | UADD16 |
| 001 | 1 | 0 | 1 | UQADD16 |
| 001 | 1 | 1 | 0 | UHADD16 |
| 001 | 1 | 1 | 1 | Unallocated. |
| 010 | 0 | 0 | 0 | SASX |
| 010 | 0 | 0 | 1 | QASX |
| 010 | 0 | 1 | 0 | SHASX |
| 010 | 0 | 1 | 1 | Unallocated. |

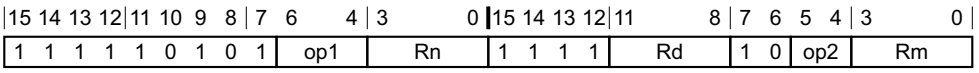

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| op1 | U | H | S | |
| 010 | 1 | 0 | 0 | UASX |
| 010 | 1 | 0 | 1 | UQASX |
| 010 | 1 | 1 | 0 | UHASX |
| 010 | 1 | 1 | 1 | Unallocated. |
| 100 | 0 | 0 | 0 | SSUB8 |
| 100 | 0 | 0 | 1 | QSUB8 |
| 100 | 0 | 1 | 0 | SHSUB8 |
| 100 | 0 | 1 | 1 | Unallocated. |
| 100 | 1 | 0 | 0 | USUB8 |
| 100 | 1 | 0 | 1 | UQSUB8 |
| 100 | 1 | 1 | 0 | UHSUB8 |
| 100 | 1 | 1 | 1 | Unallocated. |
| 101 | 0 | 0 | 0 | SSUB16 |
| 101 | 0 | 0 | 1 | QSUB16 |
| 101 | 0 | 1 | 0 | SHSUB16 |
| 101 | 0 | 1 | 1 | Unallocated. |
| 101 | 1 | 0 | 0 | USUB16 |
| 101 | 1 | 0 | 1 | UQSUB16 |
| 101 | 1 | 1 | 0 | UHSUB16 |
| 101 | 1 | 1 | 1 | Unallocated. |
| 110 | 0 | 0 | 0 | SSAX |
| 110 | 0 | 0 | 1 | QSAX |
| 110 | 0 | 1 | 0 | SHSAX |
| 110 | 0 | 1 | 1 | Unallocated. |
| 110 | 1 | 0 | 0 | USAX |
| 110 | 1 | 0 | 1 | UQSAX |
| 110 | 1 | 1 | 0 | UHSAX |
| 110 | 1 | 1 | 1 | Unallocated. |
| 111 | - | - | - | Unallocated. |

### Data-processing (two source registers)

This section describes the encoding of the Data-processing (two source registers) instruction class. This section is decoded from *Data-processing (register)* on page F3-2497.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | Instruction Page |
|---|---|---|
| op1 | op2 | |
| 000 | 00 | QADD |
| 000 | 01 | QDADD |
| 000 | 10 | QSUB |
| 000 | 11 | QDSUB |
| 001 | 00 | REV |
| 001 | 01 | REV16 |
| 001 | 10 | RBIT |
| 001 | 11 | REVSH |
| 010 | 00 | SEL |
| 010 | 01 | Unallocated. |
| 010 | 1x | Unallocated. |
| 011 | 00 | CLZ |
| 011 | 01 | Unallocated. |
| 011 | 1x | Unallocated. |
| 100 | 00 | CRC32 - CRC32B variant |
| 100 | 01 | CRC32 - CRC32H variant |
| 100 | 10 | CRC32 - CRC32W variant |
| 100 | 11 | UNPREDICTABLE |
| 101 | 00 | CRC32C - CRC32CB variant |
| 101 | 01 | CRC32C - CRC32CH variant |
| 101 | 10 | CRC32C - CRC32CW variant |
| 101 | 11 | UNPREDICTABLE |
| 11x | - | Unallocated. |

### F3.3.10 Multiply, multiply accumulate, and absolute difference

This section describes the encoding of the Multiply, multiply accumulate, and absolute difference group. This section is decoded from *32-bit T32 instruction encoding* on page F3-2476.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

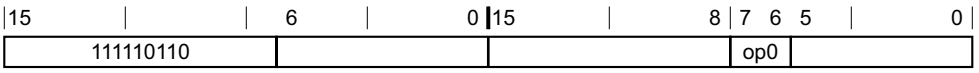
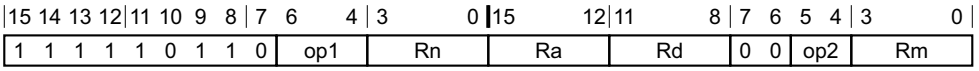



**Table F3-15 Encoding table for the Multiply, multiply accumulate, and absolute difference group**

| Decode fields | Decode group or instruction page |
|---|---|
| **op0** | |
| 00 | *Multiply and absolute difference* |
| 01 | Unallocated. |
| 1x | Unallocated. |

## Multiply and absolute difference

This section describes the encoding of the Multiply and absolute difference instruction class. This section is decoded from *Multiply, multiply accumulate, and absolute difference* on page F3-2501.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| **op1** | **op2** | **Ra** | |
| 000 | 00 | != 1111 | MLA, MLAS |
| 000 | 00 | 1111 | MUL, MULS |
| 000 | 01 | - | MLS |
| 000 | 1x | - | Unallocated. |
| 001 | 00 | != 1111 | SMLABB, SMLABT, SMLATB, SMLATT - SMLABB variant |
| 001 | 00 | 1111 | SMULBB, SMULBT, SMULTB, SMULTT - SMULBB variant |
| 001 | 01 | != 1111 | SMLABB, SMLABT, SMLATB, SMLATT - SMLABT variant |
| 001 | 01 | 1111 | SMULBB, SMULBT, SMULTB, SMULTT - SMULBT variant |
| 001 | 10 | != 1111 | SMLABB, SMLABT, SMLATB, SMLATT - SMLATB variant |
| 001 | 10 | 1111 | SMULBB, SMULBT, SMULTB, SMULTT - SMULTB variant |
| 001 | 11 | != 1111 | SMLABB, SMLABT, SMLATB, SMLATT - SMLATT variant |
| 001 | 11 | 1111 | SMULBB, SMULBT, SMULTB, SMULTT - SMULTT variant |
| 010 | 00 | != 1111 | SMLAD, SMLADX - SMLAD variant |
| 010 | 00 | 1111 | SMUAD, SMUADX - SMUAD variant |
| 010 | 01 | != 1111 | SMLAD, SMLADX - SMLADX variant |
| 010 | 01 | 1111 | SMUAD, SMUADX - SMUADX variant |

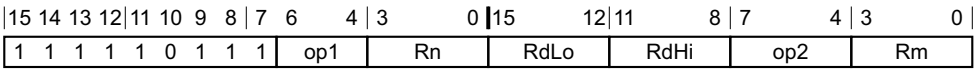

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| op1 | op2 | Ra | |
| 010 | 1x | - | Unallocated. |
| 011 | 00 | != 1111 | SMLAWB, SMLAWT - SMLAWB variant |
| 011 | 00 | 1111 | SMULWB, SMULWT - SMULWB variant |
| 011 | 01 | != 1111 | SMLAWB, SMLAWT - SMLAWT variant |
| 011 | 01 | 1111 | SMULWB, SMULWT - SMULWT variant |
| 011 | 1x | - | Unallocated. |
| 100 | 00 | != 1111 | SMLSD, SMLSDX - SMLSD variant |
| 100 | 00 | 1111 | SMUSD, SMUSDX - SMUSD variant |
| 100 | 01 | != 1111 | SMLSD, SMLSDX - SMLSDX variant |
| 100 | 01 | 1111 | SMUSD, SMUSDX - SMUSDX variant |
| 100 | 1x | - | Unallocated. |
| 101 | 00 | != 1111 | SMMLA, SMMLAR - SMMLA variant |
| 101 | 00 | 1111 | SMMUL, SMMULR - SMMUL variant |
| 101 | 01 | != 1111 | SMMLA, SMMLAR - SMMLAR variant |
| 101 | 01 | 1111 | SMMUL, SMMULR - SMMULR variant |
| 101 | 1x | - | Unallocated. |
| 110 | 00 | - | SMMLS, SMMLSR - SMMLS variant |
| 110 | 01 | - | SMMLS, SMMLSR - SMMLSR variant |
| 110 | 1x | - | Unallocated. |
| 111 | 00 | != 1111 | USADA8 |
| 111 | 00 | 1111 | USAD8 |
| 111 | 01 | - | Unallocated. |
| 111 | 1x | - | Unallocated. |

### F3.3.11 Long multiply and divide

This section describes the encoding of the Long multiply and divide instruction class. This section is decoded from *32-bit T32 instruction encoding* on page F3-2476.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | Instruction Page |
|---|---|---|
| **op1** | **op2** | |
| 000 | != 0000 | Unallocated. |
| 000 | 0000 | SMULL, SMULLS |
| 001 | != 1111 | Unallocated. |
| 001 | 1111 | SDIV |
| 010 | != 0000 | Unallocated. |
| 010 | 0000 | UMULL, UMULLS |
| 011 | != 1111 | Unallocated. |
| 011 | 1111 | UDIV |
| 100 | 0000 | SMLAL, SMLALS |
| 100 | 0001 | Unallocated. |
| 100 | 001x | Unallocated. |
| 100 | 01xx | Unallocated. |
| 100 | 1000 | SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALBB variant |
| 100 | 1001 | SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALBT variant |
| 100 | 1010 | SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALTB variant |
| 100 | 1011 | SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALTT variant |
| 100 | 1100 | SMLALD, SMLALDX - SMLALD variant |
| 100 | 1101 | SMLALD, SMLALDX - SMLALDX variant |
| 100 | 111x | Unallocated. |
| 101 | 0xxx | Unallocated. |
| 101 | 10xx | Unallocated. |
| 101 | 1100 | SMLSLD, SMLSLDX - SMLSLD variant |
| 101 | 1101 | SMLSLD, SMLSLDX - SMLSLDX variant |
| 101 | 111x | Unallocated. |
| 110 | 0000 | UMLAL, UMLALS |
| 110 | 0001 | Unallocated. |
| 110 | 001x | Unallocated. |
| 110 | 010x | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | Instruction Page |
|---|---|---|
| **op1** | **op2** | |
| 110 | 0110 | UMAAL |
| 110 | 0111 | Unallocated. |
| 110 | 1xxx | Unallocated. |
| 111 | - | Unallocated. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Chapter F4
# A32 Base Instruction Set Encoding

This chapter describes the encoding of the A32 instruction set. It contains the following sections:


- *Top level A32 instruction set encoding* on page F4-2508.
- *Data-processing and miscellaneous instructions* on page F4-2509.
- *Load/Store Word, Unsigned Byte (immediate, literal)* on page F4-2525.
- *Load/Store Word, Unsigned Byte (register)* on page F4-2526.
- *Media instructions* on page F4-2527.
- *Branch, branch with link, and block data transfer* on page F4-2535.
- *Coprocessor instructions, and Supervisor Call* on page F4-2537.
- *Unconditional instructions* on page F4-2541.

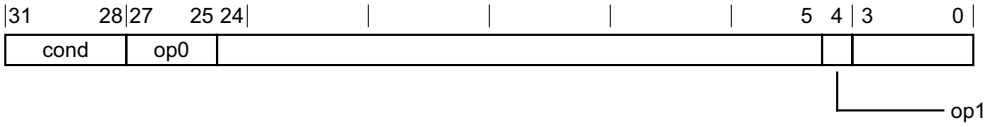

**Note**

In the decode tables in this chapter, an entry of - for a field value means the value of the field does not affect the decoding.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F4.1 Top level A32 instruction set encoding

The A32 instruction stream is a sequence of word-aligned words. Each A32 instruction is a single 32-bit word in that stream. The encoding of an A32 instruction is:




Most A32 instructions can be conditional, with a condition determined by bits[31:28] of the instruction, the `cond` field. For more information see *The condition code field in A32 instruction encodings* on page F2-2448. This applies to all instructions except those with the `cond` field equal to `0b1111`.

Table F4-1 shows the top-level encoding of the A32 instruction set, including the cases where bits[31:28] are defined as `0b1111` rather than as the `cond`.

For more information on A32 instruction encodings see *About the T32 and A32 instruction encodings* on page F2-2454.

**Table F4-1 Main encoding table for the A32 instruction set**

| Decode fields | | | Decode group or instruction page |
|---|---|---|---|
| **cond** | **op0** | **op1** | |
| != 1111 | 00x | - | *Data-processing and miscellaneous instructions* on page F4-2509 |
| != 1111 | 010 | - | *Load/Store Word, Unsigned Byte (immediate, literal)* on page F4-2525 |
| != 1111 | 011 | 0 | *Load/Store Word, Unsigned Byte (register)* on page F4-2526 |
| != 1111 | 011 | 1 | *Media instructions* on page F4-2527 |
| - | 10x | - | *Branch, branch with link, and block data transfer* on page F4-2535 |
| - | 11x | - | *Coprocessor instructions, and Supervisor Call* on page F4-2537 |
| 1111 | 0xx | - | *Unconditional instructions* on page F4-2541 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F4.2 Data-processing and miscellaneous instructions

This section describes the encoding of the Data-processing and miscellaneous instructions group. This section is decoded from *Top level A32 instruction set encoding* on page F4-2508.

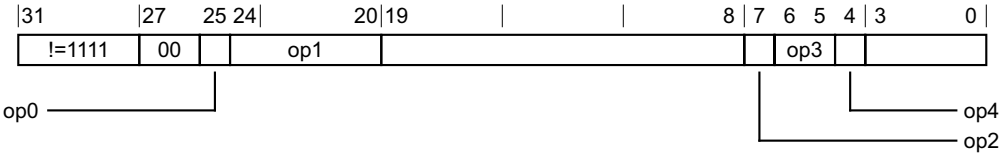



**Table F4-2 Encoding table for the Data-processing and miscellaneous instructions group**

| Decode fields | | | | | Decode group or instruction page |
|---|---|---|---|---|---|
| op0 | op1 | op2 | op3 | op4 | |
| 0 | - | 1 | != 00 | 1 | *Extra load/store* |
| 0 | 0xxxx | 1 | 00 | 1 | *Multiply and Accumulate* on page F4-2512 |
| 0 | 1xxxx | 1 | 00 | 1 | *Synchronization primitives* on page F4-2512 |
| 0 | 10xx0 | 0 | - | - | *Miscellaneous* on page F4-2514 |
| 0 | 10xx0 | 1 | - | 0 | *Halfword Multiply and Accumulate* on page F4-2516 |
| 0 | != 10xx0 | - | - | 0 | *Data-processing register (immediate shift)* on page F4-2517 |
| 0 | != 10xx0 | 0 | - | 1 | *Data-processing register (register shift)* on page F4-2519 |
| 1 | - | - | - | - | *Data-processing immediate* on page F4-2521 |

### F4.2.1 Extra load/store

This section describes the encoding of the Extra load/store group. This section is decoded from *Data-processing and miscellaneous instructions*.

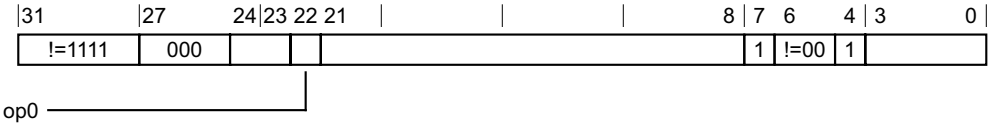


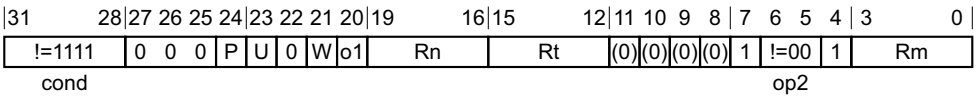

**Table F4-3 Encoding table for the Extra load/store group**

| Decode fields | Decode group or instruction page |
|---|---|
| op0 | |
| 0 | *Load/Store Dual, Half, Signed Byte (register)* |
| 1 | *Load/Store Dual, Half, Signed Byte (immediate, literal)* on page F4-2510 |

#### Load/Store Dual, Half, Signed Byte (register)

This section describes the encoding of the Load/Store Dual, Half, Signed Byte (register) instruction class. This section is decoded from *Extra load/store*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

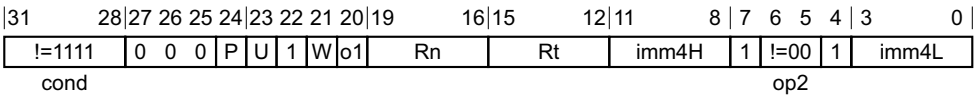



| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| o1 | op2 | P | W | |
| 0 | 01 | 0 | 0 | STRH (register) - Post-indexed variant |
| 0 | 01 | 0 | 1 | STRHT |
| 0 | 01 | 1 | - | STRH (register) - Pre-indexed variant |
| 0 | 10 | 0 | 0 | LDRD (register) - Post-indexed variant |
| 0 | 10 | 0 | 1 | Unallocated. |
| 0 | 10 | 1 | - | LDRD (register) - Pre-indexed variant |
| 0 | 11 | 0 | 0 | STRD (register) - Post-indexed variant |
| 0 | 11 | 0 | 1 | Unallocated. |
| 0 | 11 | 1 | - | STRD (register) - Pre-indexed variant |
| 1 | 01 | 0 | 0 | LDRH (register) - Post-indexed variant |
| 1 | 01 | 0 | 1 | LDRHT |
| 1 | 01 | 1 | - | LDRH (register) - Pre-indexed variant |
| 1 | 10 | 0 | 0 | LDRSB (register) - Post-indexed variant |
| 1 | 10 | 0 | 1 | LDRSBT |
| 1 | 10 | 1 | - | LDRSB (register) - Pre-indexed variant |
| 1 | 11 | 0 | 0 | LDRSH (register) - Post-indexed variant |
| 1 | 11 | 0 | 1 | LDRSHT |
| 1 | 11 | 1 | - | LDRSH (register) - Pre-indexed variant |

### Load/Store Dual, Half, Signed Byte (immediate, literal)

This section describes the encoding of the Load/Store Dual, Half, Signed Byte (immediate, literal) instruction class. This section is decoded from *Extra load/store* on page F4-2509.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

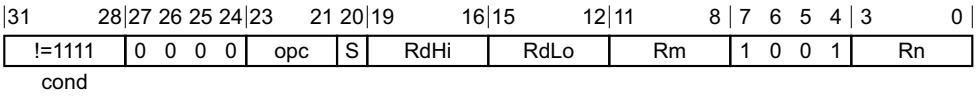



| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| **o1** | **op2** | **P:W** | **Rn** | |
| 0 | 01 | 00 | - | STRH (immediate) - Post-indexed variant |
| 0 | 01 | 01 | - | STRHT |
| 0 | 01 | 10 | - | STRH (immediate) - Offset variant |
| 0 | 01 | 11 | - | STRH (immediate) - Pre-indexed variant |
| 0 | 10 | - | 1111 | LDRD (literal) |
| 0 | 10 | 00 | != 1111 | LDRD (immediate) - Post-indexed variant |
| 0 | 10 | 01 | != 1111 | Unallocated. |
| 0 | 10 | 10 | != 1111 | LDRD (immediate) - Offset variant |
| 0 | 10 | 11 | != 1111 | LDRD (immediate) - Pre-indexed variant |
| 0 | 11 | 00 | - | STRD (immediate) - Post-indexed variant |
| 0 | 11 | 01 | - | Unallocated. |
| 0 | 11 | 10 | - | STRD (immediate) - Offset variant |
| 0 | 11 | 11 | - | STRD (immediate) - Pre-indexed variant |
| 1 | 01 | != 01 | 1111 | LDRH (literal) |
| 1 | 01 | 00 | != 1111 | LDRH (immediate) - Post-indexed variant |
| 1 | 01 | 01 | - | LDRHT |
| 1 | 01 | 10 | != 1111 | LDRH (immediate) - Offset variant |
| 1 | 01 | 11 | != 1111 | LDRH (immediate) - Pre-indexed variant |
| 1 | 10 | != 01 | 1111 | LDRSB (literal) |
| 1 | 10 | 00 | != 1111 | LDRSB (immediate) - Post-indexed variant |
| 1 | 10 | 01 | - | LDRSBT |
| 1 | 10 | 10 | != 1111 | LDRSB (immediate) - Offset variant |
| 1 | 10 | 11 | != 1111 | LDRSB (immediate) - Pre-indexed variant |
| 1 | 11 | != 01 | 1111 | LDRSH (literal) |
| 1 | 11 | 00 | != 1111 | LDRSH (immediate) - Post-indexed variant |
| 1 | 11 | 01 | - | LDRSHT |
| 1 | 11 | 10 | != 1111 | LDRSH (immediate) - Offset variant |
| 1 | 11 | 11 | != 1111 | LDRSH (immediate) - Pre-indexed variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F4.2.2 Multiply and Accumulate

This section describes the encoding of the Multiply and Accumulate instruction class. This section is decoded from *Data-processing and miscellaneous instructions* on page F4-2509.




| Decode fields | | Instruction Page |
|---|---|---|
| opc | S | |
| 000 | - | MUL, MULS |
| 001 | - | MLA, MLAS |
| 010 | 0 | UMAAL |
| 010 | 1 | Unallocated. |
| 011 | 0 | MLS |
| 011 | 1 | Unallocated. |
| 100 | - | UMULL, UMULLS |
| 101 | - | UMLAL, UMLALS |
| 110 | - | SMULL, SMULLS |
| 111 | - | SMLAL, SMLALS |

### F4.2.3 Synchronization primitives

This section describes the encoding of the Synchronization primitives group. This section is decoded from *Data-processing and miscellaneous instructions* on page F4-2509.

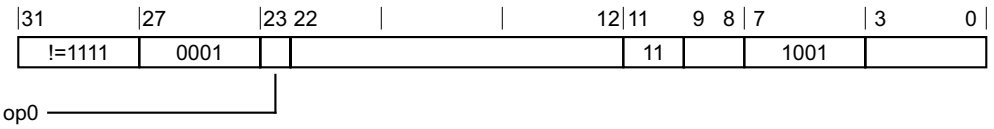


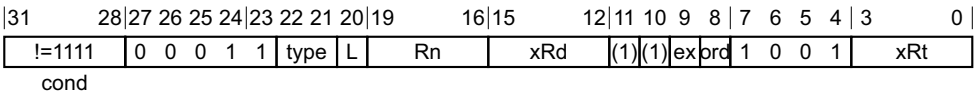

**Table F4-4 Encoding table for the Synchronization primitives group**

| Decode fields | Decode group or instruction page |
|---|---|
| op0 | |
| 0 | Unallocated. |
| 1 | *Load/Store Exclusive* |

#### Load/Store Exclusive

This section describes the encoding of the Load/Store Exclusive instruction class. This section is decoded from *Synchronization primitives*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| type | L | ex | ord | |
| 00 | 0 | 0 | 0 | STL |
| 00 | 0 | 0 | 1 | Unallocated. |
| 00 | 0 | 1 | 0 | STLEX |
| 00 | 0 | 1 | 1 | STREX |
| 00 | 1 | 0 | 0 | LDA |
| 00 | 1 | 0 | 1 | Unallocated. |
| 00 | 1 | 1 | 0 | LDAEX |
| 00 | 1 | 1 | 1 | LDREX |
| 01 | 0 | 0 | - | Unallocated. |
| 01 | 0 | 1 | 0 | STLEXD |
| 01 | 0 | 1 | 1 | STREXD |
| 01 | 1 | 0 | - | Unallocated. |
| 01 | 1 | 1 | 0 | LDAEXD |
| 01 | 1 | 1 | 1 | LDREXD |
| 10 | 0 | 0 | 0 | STLB |
| 10 | 0 | 0 | 1 | Unallocated. |
| 10 | 0 | 1 | 0 | STLEXB |
| 10 | 0 | 1 | 1 | STREXB |
| 10 | 1 | 0 | 0 | LDAB |
| 10 | 1 | 0 | 1 | Unallocated. |
| 10 | 1 | 1 | 0 | LDAEXB |
| 10 | 1 | 1 | 1 | LDREXB |
| 11 | 0 | 0 | 0 | STLH |
| 11 | 0 | 0 | 1 | Unallocated. |
| 11 | 0 | 1 | 0 | STLEXH |
| 11 | 0 | 1 | 1 | STREXH |
| 11 | 1 | 0 | 0 | LDAH |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| type | L | ex | ord | |
| 11 | 1 | 0 | 1 | Unallocated. |
| 11 | 1 | 1 | 0 | LDAEXH |
| 11 | 1 | 1 | 1 | LDREXH |

### F4.2.4 Miscellaneous

This section describes the encoding of the Miscellaneous group. This section is decoded from *Data-processing and miscellaneous instructions* on page F4-2509.

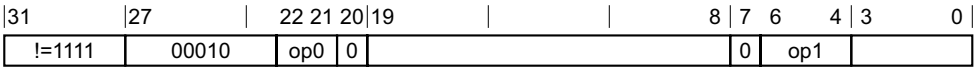



**Table F4-5 Encoding table for the Miscellaneous group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 00 | 001 | Unallocated. |
| 00 | 010 | Unallocated. |
| 00 | 011 | Unallocated. |
| 00 | 110 | Unallocated. |
| 01 | 001 | BX |
| 01 | 010 | BXJ |
| 01 | 011 | BLX (register) |
| 01 | 110 | Unallocated. |
| 10 | 001 | Unallocated. |
| 10 | 010 | Unallocated. |
| 10 | 011 | Unallocated. |
| 10 | 110 | Unallocated. |
| 11 | 001 | CLZ |
| 11 | 010 | Unallocated. |
| 11 | 011 | Unallocated. |
| 11 | 110 | ERET |
| - | 111 | *Exception Generation* on page F4-2515 |

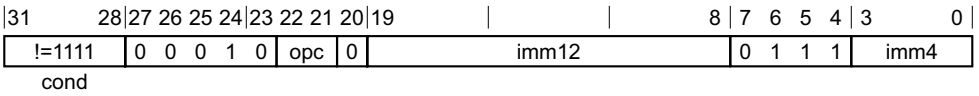

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F4-5 Encoding table for the Miscellaneous group (continued)**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| - | 000 | *Move special register (register)* |
| - | 100 | *Cyclic Redundancy Check* on page F4-2516 |
| - | 101 | *Integer Saturating Arithmetic* on page F4-2516 |

## Exception Generation

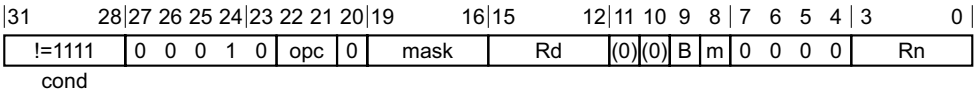
This section describes the encoding of the Exception Generation instruction class. This section is decoded from *Miscellaneous* on page F4-2514.




| Decode fields | Instruction Page |
|---|---|
| opc | |
| 00 | HLT |
| 01 | BKPT |
| 10 | HVC |
| 11 | SMC |

## Move special register (register)

This section describes the encoding of the Move special register (register) instruction class. This section is decoded from *Miscellaneous* on page F4-2514.

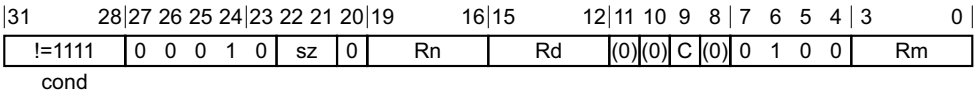



| Decode fields | | Instruction Page |
|---|---|---|
| opc | B | |
| x0 | 0 | MRS |
| x0 | 1 | MRS (Banked register) |
| x1 | 0 | MSR (register) |
| x1 | 1 | MSR (Banked register) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Cyclic Redundancy Check

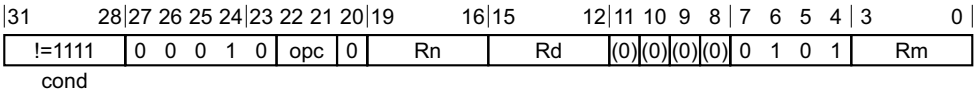
This section describes the encoding of the Cyclic Redundancy Check instruction class. This section is decoded from *Miscellaneous* on page F4-2514.




| Decode fields | | Instruction Page |
|---|---|---|
| sz | C | |
| 00 | 0 | CRC32 - CRC32B variant |
| 00 | 1 | CRC32C - CRC32CB variant |
| 01 | 0 | CRC32 - CRC32H variant |
| 01 | 1 | CRC32C - CRC32CH variant |
| 10 | 0 | CRC32 - CRC32W variant |
| 10 | 1 | CRC32C - CRC32CW variant |
| 11 | - | UNPREDICTABLE |

### Integer Saturating Arithmetic

This section describes the encoding of the Integer Saturating Arithmetic instruction class. This section is decoded from *Miscellaneous* on page F4-2514.

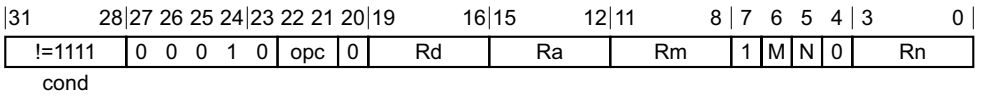



| Decode fields | Instruction Page |
|---|---|
| opc | |
| 00 | QADD |
| 01 | QSUB |
| 10 | QDADD |
| 11 | QDSUB |

## F4.2.5 Halfword Multiply and Accumulate

This section describes the encoding of the Halfword Multiply and Accumulate instruction class. This section is decoded from *Data-processing and miscellaneous instructions* on page F4-2509.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| opc | M | N | |
| 00 | - | - | SMLABB, SMLABT, SMLATB, SMLATT |
| 01 | 0 | 0 | SMLAWB, SMLAWT - SMLAWB variant |
| 01 | 0 | 1 | SMULWB, SMULWT - SMULWB variant |
| 01 | 1 | 0 | SMLAWB, SMLAWT - SMLAWT variant |
| 01 | 1 | 1 | SMULWB, SMULWT - SMULWT variant |
| 10 | - | - | SMLALBB, SMLALBT, SMLALTB, SMLALTT |
| 11 | - | - | SMULBB, SMULBT, SMULTB, SMULTT |

### F4.2.6 Data-processing register (immediate shift)

This section describes the encoding of the Data-processing register (immediate shift) group. This section is decoded from *Data-processing and miscellaneous instructions* on page F4-2509.

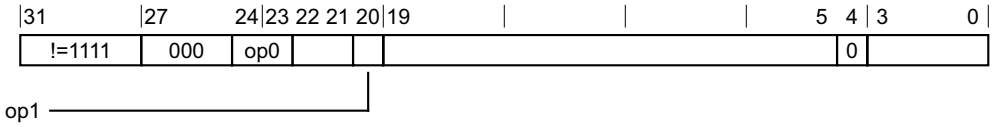


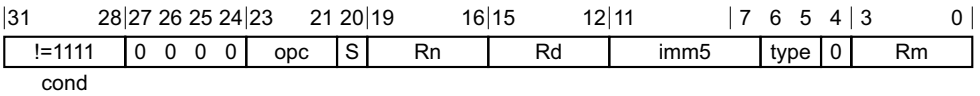

**Table F4-6 Encoding table for the Data-processing register (immediate shift) group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 0x | - | *Integer Data Processing (three register, immediate shift)* |
| 10 | 1 | *Integer Test & Compare (two register, immediate shift)* on page F4-2518 |
| 11 | - | *Logical Arithmetic (three register, immediate shift)* on page F4-2519 |

#### Integer Data Processing (three register, immediate shift)

This section describes the encoding of the Integer Data Processing (three register, immediate shift) instruction class. This section is decoded from *Data-processing register (immediate shift)*.

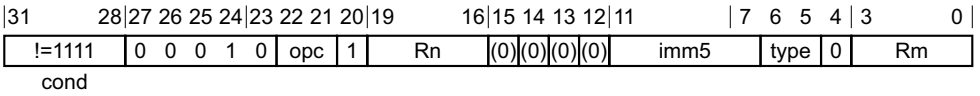

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| opc | S | Rn | |
| 000 | - | - | AND, ANDS (register) |
| 001 | - | - | EOR, EORS (register) |
| 010 | 0 | != 1101 | SUB, SUBS (register) - SUB, rotate right with extend variant |
| 010 | 0 | 1101 | SUB, SUBS (SP minus register) - SUB, rotate right with extend variant |
| 010 | 1 | != 1101 | SUB, SUBS (register) - SUBS, rotate right with extend variant |
| 010 | 1 | 1101 | SUB, SUBS (SP minus register) - SUBS, rotate right with extend variant |
| 011 | - | - | RSB, RSBS (register) |
| 100 | 0 | != 1101 | ADD, ADDS (register) - ADD, rotate right with extend variant |
| 100 | 0 | 1101 | ADD, ADDS (SP plus register) - ADD, rotate right with extend variant |
| 100 | 1 | != 1101 | ADD, ADDS (register) - ADDS, rotate right with extend variant |
| 100 | 1 | 1101 | ADD, ADDS (SP plus register) - ADDS, rotate right with extend variant |
| 101 | - | - | ADC, ADCS (register) |
| 110 | - | - | SBC, SBCS (register) |
| 111 | - | - | RSC, RSCS (register) |

### Integer Test & Compare (two register, immediate shift)

This section describes the encoding of the Integer Test & Compare (two register, immediate shift) instruction class. This section is decoded from *Data-processing register (immediate shift)* on page F4-2517.

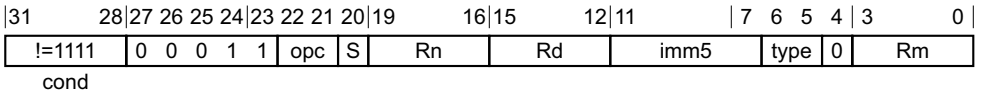



| Decode fields | Instruction Page |
|---|---|
| opc | |
| 00 | TST (register) |
| 01 | TEQ (register) |
| 10 | CMP (register) |
| 11 | CMN (register) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Logical Arithmetic (three register, immediate shift)

This section describes the encoding of the Logical Arithmetic (three register, immediate shift) instruction class. This section is decoded from *Data-processing register (immediate shift)* on page F4-2517.




| Decode fields | Instruction Page |
|---|---|
| opc | |
| 00 | ORR, ORRS (register) |
| 01 | MOV, MOVS (register) |
| 10 | BIC, BICS (register) |
| 11 | MVN, MVNS (register) |

## F4.2.7 Data-processing register (register shift)

This section describes the encoding of the Data-processing register (register shift) group. This section is decoded from *Data-processing and miscellaneous instructions* on page F4-2509.

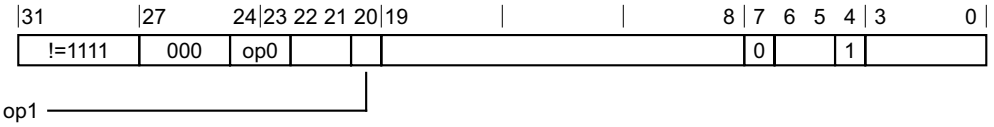


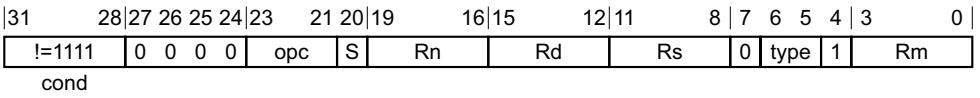

**Table F4-7 Encoding table for the Data-processing register (register shift) group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 0x | - | *Integer Data Processing (three register, register shift)* |
| 10 | 1 | *Integer Test & Compare (two register, register shift)* on page F4-2520 |
| 11 | - | *Logical Arithmetic (three register, register shift)* on page F4-2520 |

### Integer Data Processing (three register, register shift)

This section describes the encoding of the Integer Data Processing (three register, register shift) instruction class. This section is decoded from *Data-processing register (register shift)*.

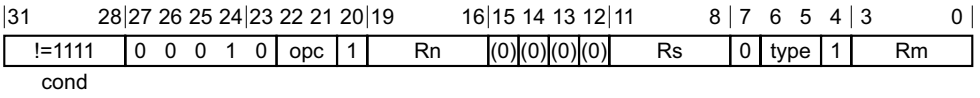

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields<br>opc | Instruction Page |
|---|---|
| 000 | AND, ANDS (register-shifted register) |
| 001 | EOR, EORS (register-shifted register) |
| 010 | SUB, SUBS (register-shifted register) |
| 011 | RSB, RSBS (register-shifted register) |
| 100 | ADD, ADDS (register-shifted register) |
| 101 | ADC, ADCS (register-shifted register) |
| 110 | SBC, SBCS (register-shifted register) |
| 111 | RSC, RSCS (register-shifted register) |

### Integer Test & Compare (two register, register shift)

This section describes the encoding of the Integer Test & Compare (two register, register shift) instruction class. This section is decoded from *Data-processing register (register shift)* on page F4-2519.

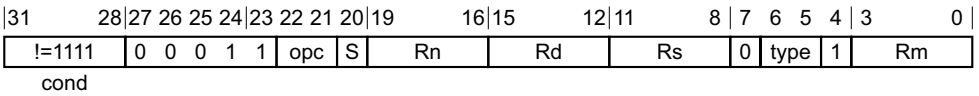



| Decode fields<br>opc | Instruction Page |
|---|---|
| 00 | TST (register-shifted register) |
| 01 | TEQ (register-shifted register) |
| 10 | CMP (register-shifted register) |
| 11 | CMN (register-shifted register) |

### Logical Arithmetic (three register, register shift)

This section describes the encoding of the Logical Arithmetic (three register, register shift) instruction class. This section is decoded from *Data-processing register (register shift)* on page F4-2519.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | Instruction Page |
|---|---|
| opc | |
| 00 | ORR, ORRS (register-shifted register) |
| 01 | MOV, MOVS (register-shifted register) |
| 10 | BIC, BICS (register-shifted register) |
| 11 | MVN, MVNS (register-shifted register) |

### F4.2.8 Data-processing immediate

This section describes the encoding of the Data-processing immediate group. This section is decoded from *Data-processing and miscellaneous instructions* on page F4-2509.

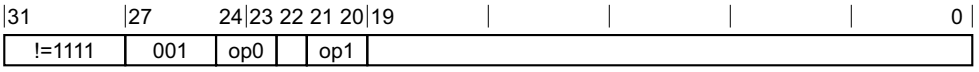


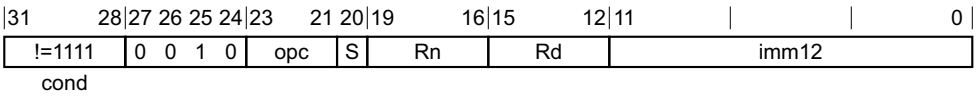

**Table F4-8 Encoding table for the Data-processing immediate group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 0x | - | *Integer Data Processing (two register and immediate)* |
| 10 | 00 | *Move Halfword (immediate)* on page F4-2522 |
| 10 | 10 | *Move Special Register & Hints (immediate)* on page F4-2523 |
| 10 | x1 | *Integer Test & Compare (one register and immediate)* on page F4-2523 |
| 11 | - | *Logical Arithmetic (two register and immediate)* on page F4-2523 |

#### Integer Data Processing (two register and immediate)

This section describes the encoding of the Integer Data Processing (two register and immediate) instruction class. This section is decoded from *Data-processing immediate*.

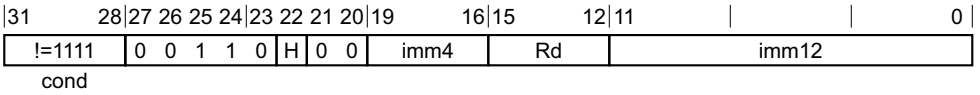

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| opc | S | Rn | |
| 000 | - | - | AND, ANDS (immediate) |
| 001 | - | - | EOR, EORS (immediate) |
| 010 | 0 | != 11x1 | SUB, SUBS (immediate) - SUB variant |
| 010 | 0 | 1101 | SUB, SUBS (SP minus immediate) - SUB variant |
| 010 | 0 | 1111 | ADR - A2 variant |
| 010 | 1 | != 1101 | SUB, SUBS (immediate) - SUBS variant |
| 010 | 1 | 1101 | SUB, SUBS (SP minus immediate) - SUBS variant |
| 011 | - | - | RSB, RSBS (immediate) |
| 100 | 0 | != 11x1 | ADD, ADDS (immediate) - ADD variant |
| 100 | 0 | 1101 | ADD, ADDS (SP plus immediate) - ADD variant |
| 100 | 0 | 1111 | ADR - A1 variant |
| 100 | 1 | != 1101 | ADD, ADDS (immediate) - ADDS variant |
| 100 | 1 | 1101 | ADD, ADDS (SP plus immediate) - ADDS variant |
| 101 | - | - | ADC, ADCS (immediate) |
| 110 | - | - | SBC, SBCS (immediate) |
| 111 | - | - | RSC, RSCS (immediate) |

### Move Halfword (immediate)

This section describes the encoding of the Move Halfword (immediate) instruction class. This section is decoded from *Data-processing immediate* on page F4-2521.

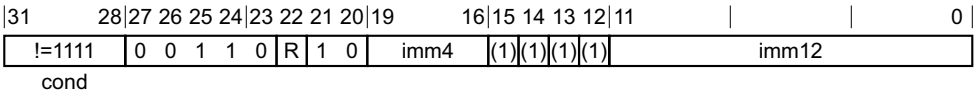



| Decode fields | Instruction Page |
|---|---|
| H | |
| 0 | MOV, MOVS (immediate) |
| 1 | MOVT |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Move Special Register & Hints (immediate)

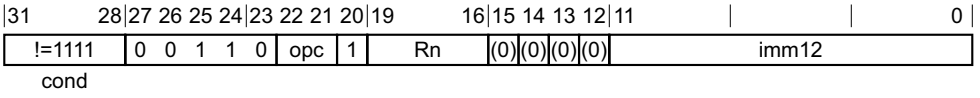
This section describes the encoding of the Move Special Register & Hints (immediate) instruction class. This section is decoded from *Data-processing immediate* on page F4-2521.




| Decode fields | | Instruction Page |
|---|---|---|
| R:imm4 | imm12 | |
| != 00000 | - | MSR (immediate) |
| 00000 | xxxx00000000 | NOP |
| 00000 | xxxx00000001 | YIELD |
| 00000 | xxxx00000010 | WFE |
| 00000 | xxxx00000011 | WFI |
| 00000 | xxxx00000100 | SEV |
| 00000 | xxxx00000101 | SEVL |
| 00000 | xxxx1111xxxx | DBG |

### Integer Test & Compare (one register and immediate)

This section describes the encoding of the Integer Test & Compare (one register and immediate) instruction class. This section is decoded from *Data-processing immediate* on page F4-2521.

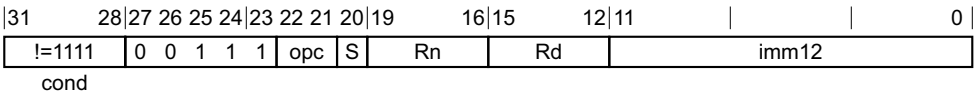



| Decode fields | Instruction Page |
|---|---|
| opc | |
| 00 | TST (immediate) |
| 01 | TEQ (immediate) |
| 10 | CMP (immediate) |
| 11 | CMN (immediate) |

### Logical Arithmetic (two register and immediate)

This section describes the encoding of the Logical Arithmetic (two register and immediate) instruction class. This section is decoded from *Data-processing immediate* on page F4-2521.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

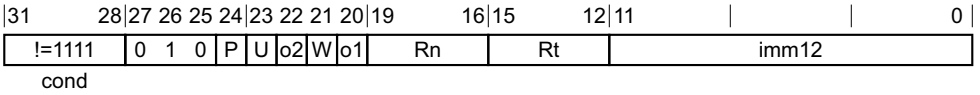



| Decode fields | Instruction Page |
|---|---|
| **opc** | |
| 00 | ORR, ORRS (immediate) |
| 01 | MOV, MOVS (immediate) |
| 10 | BIC, BICS (immediate) |
| 11 | MVN, MVNS (immediate) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F4.3 Load/Store Word, Unsigned Byte (immediate, literal)

This section describes the encoding of the Load/Store Word, Unsigned Byte (immediate, literal) instructions group. This section is decoded from *Top level A32 instruction set encoding* on page F4-2508.

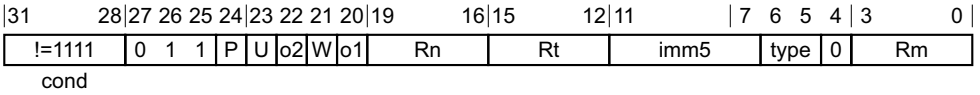



| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| o1 | o2 | P:W | Rn | |
| 0 | 0 | 00 | - | STR (immediate) - Post-indexed variant |
| 0 | 0 | 01 | - | STRT |
| 0 | 0 | 10 | - | STR (immediate) - Offset variant |
| 0 | 0 | 11 | - | STR (immediate) - Pre-indexed variant |
| 0 | 1 | 00 | - | STRB (immediate) - Post-indexed variant |
| 0 | 1 | 01 | - | STRBT |
| 0 | 1 | 10 | - | STRB (immediate) - Offset variant |
| 0 | 1 | 11 | - | STRB (immediate) - Pre-indexed variant |
| 1 | 0 | != 01 | 1111 | LDR (literal) |
| 1 | 0 | 00 | != 1111 | LDR (immediate) - Post-indexed variant |
| 1 | 0 | 01 | - | LDRT |
| 1 | 0 | 10 | != 1111 | LDR (immediate) - Offset variant |
| 1 | 0 | 11 | != 1111 | LDR (immediate) - Pre-indexed variant |
| 1 | 1 | != 01 | 1111 | LDRB (literal) |
| 1 | 1 | 00 | != 1111 | LDRB (immediate) - Post-indexed variant |
| 1 | 1 | 01 | - | LDRBT |
| 1 | 1 | 10 | != 1111 | LDRB (immediate) - Offset variant |
| 1 | 1 | 11 | != 1111 | LDRB (immediate) - Pre-indexed variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F4.4 Load/Store Word, Unsigned Byte (register)

This section describes the encoding of the Load/Store Word, Unsigned Byte (register) instructions group. This section is decoded from *Top level A32 instruction set encoding* on page F4-2508.




| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| o1 | o2 | P | W | |
| 0 | 0 | 0 | 0 | STR (register) - Post-indexed variant |
| 0 | 0 | 0 | 1 | STRT |
| 0 | 0 | 1 | - | STR (register) - Pre-indexed variant |
| 0 | 1 | 0 | 0 | STRB (register) - Post-indexed variant |
| 0 | 1 | 0 | 1 | STRBT |
| 0 | 1 | 1 | - | STRB (register) - Pre-indexed variant |
| 1 | 0 | 0 | 0 | LDR (register) - Post-indexed variant |
| 1 | 0 | 0 | 1 | LDRT |
| 1 | 0 | 1 | - | LDR (register) - Pre-indexed variant |
| 1 | 1 | 0 | 0 | LDRB (register) - Post-indexed variant |
| 1 | 1 | 0 | 1 | LDRBT |
| 1 | 1 | 1 | - | LDRB (register) - Pre-indexed variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F4.5 Media instructions

This section describes the encoding of the Media instructions group. This section is decoded from *Top level A32 instruction set encoding* on page F4-2508.

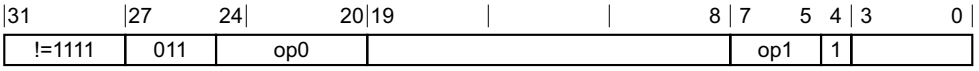



**Table F4-9 Encoding table for the Media instructions group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 00xxx | - | *Parallel Arithmetic* on page F4-2528 |
| 01000 | 101 | SEL |
| 01000 | 001 | Unallocated. |
| 01000 | xx0 | PKHBT, PKHTB |
| 01001 | x01 | Unallocated. |
| 01001 | xx0 | Unallocated. |
| 0110x | x01 | Unallocated. |
| 0110x | xx0 | Unallocated. |
| 01x10 | 001 | *Saturate 16-bit* on page F4-2530 |
| 01x10 | 101 | Unallocated. |
| 01x11 | x01 | *Reverse Bit/Byte* on page F4-2530 |
| 01x1x | xx0 | *Saturate 32-bit* on page F4-2530 |
| 01xxx | 111 | Unallocated. |
| 01xxx | 011 | *Extend and Add* on page F4-2531 |
| 10xxx | - | *Signed multiply, Divide* on page F4-2531 |
| 11000 | 000 | *Unsigned Sum of Absolute Differences* on page F4-2532 |
| 11000 | 100 | Unallocated. |
| 11001 | x00 | Unallocated. |
| 1101x | x00 | Unallocated. |
| 110xx | 111 | Unallocated. |
| 1110x | 111 | Unallocated. |
| 1110x | x00 | *Bitfield Insert* on page F4-2533 |
| 11110 | 111 | Unallocated. |
| 11111 | 111 | *Permanently UNDEFINED* on page F4-2533 |
| 1111x | x00 | Unallocated. |
| 11x0x | x10 | Unallocated. |

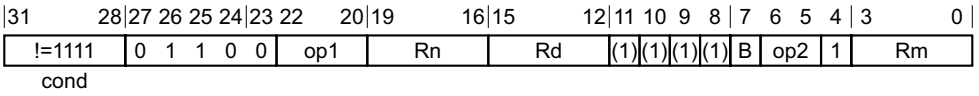

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F4-9 Encoding table for the Media instructions group (continued)**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| **op0** | **op1** | |
| 11x1x | x10 | *Bitfield Extract* on page F4-2534 |
| 11xxx | 011 | Unallocated. |
| 11xxx | x01 | Unallocated. |

## F4.5.1 Parallel Arithmetic

This section describes the encoding of the Parallel Arithmetic instruction class. This section is decoded from *Media instructions* on page F4-2527.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| **op1** | **op2** | **B** | |
| 000 | - | - | Unallocated. |
| 001 | 00 | 0 | SADD16 |
| 001 | 00 | 1 | SADD8 |
| 001 | 01 | 0 | SASX |
| 001 | 01 | 1 | Unallocated. |
| 001 | 10 | 0 | SSAX |
| 001 | 10 | 1 | Unallocated. |
| 001 | 11 | 0 | SSUB16 |
| 001 | 11 | 1 | SSUB8 |
| 010 | 00 | 0 | QADD16 |
| 010 | 00 | 1 | QADD8 |
| 010 | 01 | 0 | QASX |
| 010 | 01 | 1 | Unallocated. |
| 010 | 10 | 0 | QSAX |
| 010 | 10 | 1 | Unallocated. |
| 010 | 11 | 0 | QSUB16 |
| 010 | 11 | 1 | QSUB8 |
| 011 | 00 | 0 | SHADD16 |
| 011 | 00 | 1 | SHADD8 |

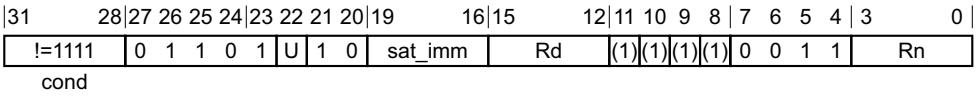

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| op1 | op2 | B | |
| 011 | 01 | 0 | SHASX |
| 011 | 01 | 1 | Unallocated. |
| 011 | 10 | 0 | SHSAX |
| 011 | 10 | 1 | Unallocated. |
| 011 | 11 | 0 | SHSUB16 |
| 011 | 11 | 1 | SHSUB8 |
| 100 | - | - | Unallocated. |
| 101 | 00 | 0 | UADD16 |
| 101 | 00 | 1 | UADD8 |
| 101 | 01 | 0 | UASX |
| 101 | 01 | 1 | Unallocated. |
| 101 | 10 | 0 | USAX |
| 101 | 10 | 1 | Unallocated. |
| 101 | 11 | 0 | USUB16 |
| 101 | 11 | 1 | USUB8 |
| 110 | 00 | 0 | UQADD16 |
| 110 | 00 | 1 | UQADD8 |
| 110 | 01 | 0 | UQASX |
| 110 | 01 | 1 | Unallocated. |
| 110 | 10 | 0 | UQSAX |
| 110 | 10 | 1 | Unallocated. |
| 110 | 11 | 0 | UQSUB16 |
| 110 | 11 | 1 | UQSUB8 |
| 111 | 00 | 0 | UHADD16 |
| 111 | 00 | 1 | UHADD8 |
| 111 | 01 | 0 | UHASX |
| 111 | 01 | 1 | Unallocated. |
| 111 | 10 | 0 | UHSAX |
| 111 | 10 | 1 | Unallocated. |
| 111 | 11 | 0 | UHSUB16 |
| 111 | 11 | 1 | UHSUB8 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F4.5.2 Saturate 16-bit

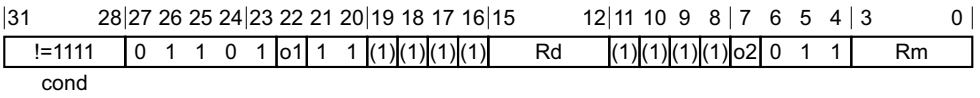
This section describes the encoding of the Saturate 16-bit instruction class. This section is decoded from *Media instructions* on page F4-2527.




| Decode fields | Instruction Page |
|---|---|
| U | |
| 0 | SSAT16 |
| 1 | USAT16 |

### F4.5.3 Reverse Bit/Byte

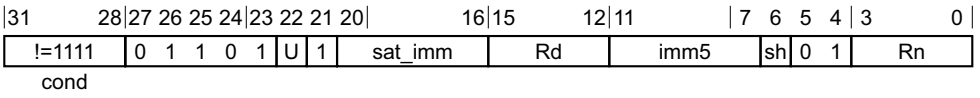
This section describes the encoding of the Reverse Bit/Byte instruction class. This section is decoded from *Media instructions* on page F4-2527.




| Decode fields | | Instruction Page |
|---|---|---|
| o1 | o2 | |
| 0 | 0 | REV |
| 0 | 1 | REV16 |
| 1 | 0 | RBIT |
| 1 | 1 | REVSH |

### F4.5.4 Saturate 32-bit

This section describes the encoding of the Saturate 32-bit instruction class. This section is decoded from *Media instructions* on page F4-2527.

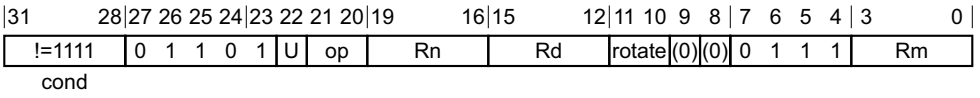



| Decode fields | Instruction Page |
|---|---|
| U | |
| 0 | SSAT |
| 1 | USAT |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F4.5.5 Extend and Add

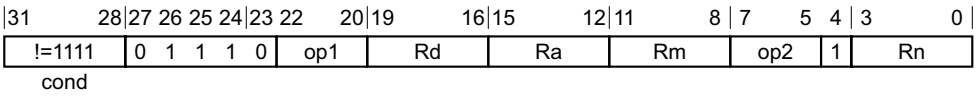
This section describes the encoding of the Extend and Add instruction class. This section is decoded from *Media instructions* on page F4-2527.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| op | U | Rn | |
| 00 | 0 | != 1111 | SXTAB16 |
| 00 | 0 | 1111 | SXTB16 |
| 00 | 1 | != 1111 | UXTAB16 |
| 00 | 1 | 1111 | UXTB16 |
| 10 | 0 | != 1111 | SXTAB |
| 10 | 0 | 1111 | SXTB |
| 10 | 1 | != 1111 | UXTAB |
| 10 | 1 | 1111 | UXTB |
| 11 | 0 | != 1111 | SXTAH |
| 11 | 0 | 1111 | SXTH |
| 11 | 1 | != 1111 | UXTAH |
| 11 | 1 | 1111 | UXTH |

### F4.5.6 Signed multiply, Divide

This section describes the encoding of the Signed multiply, Divide instruction class. This section is decoded from *Media instructions* on page F4-2527.




| Decode fields | | | Instruction Page |
|---|---|---|---|
| op1 | op2 | Ra | |
| 000 | 000 | != 1111 | SMLAD, SMLADX - SMLAD variant |
| 000 | 000 | 1111 | SMUAD, SMUADX - SMUAD variant |
| 000 | 001 | != 1111 | SMLAD, SMLADX - SMLADX variant |
| 000 | 001 | 1111 | SMUAD, SMUADX - SMUADX variant |
| 000 | 010 | != 1111 | SMLSD, SMLSDX - SMLSD variant |

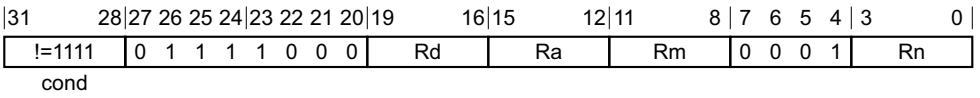

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Decode fields | | | Instruction Page |
|---|---|---|---|
| op1 | op2 | Ra | |
| 000 | 010 | 1111 | SMUSD, SMUSDX - SMUSD variant |
| 000 | 011 | != 1111 | SMLSD, SMLSDX - SMLSDX variant |
| 000 | 011 | 1111 | SMUSD, SMUSDX - SMUSDX variant |
| 000 | 1xx | - | Unallocated. |
| 001 | != 000 | - | Unallocated. |
| 001 | 000 | - | SDIV |
| 010 | - | - | Unallocated. |
| 011 | != 000 | - | Unallocated. |
| 011 | 000 | - | UDIV |
| 100 | 000 | - | SMLALD, SMLALDX - SMLALD variant |
| 100 | 001 | - | SMLALD, SMLALDX - SMLALDX variant |
| 100 | 010 | - | SMLSLD, SMLSLDX - SMLSLD variant |
| 100 | 011 | - | SMLSLD, SMLSLDX - SMLSLDX variant |
| 100 | 1xx | - | Unallocated. |
| 101 | 000 | != 1111 | SMMLA, SMMLAR - SMMLA variant |
| 101 | 000 | 1111 | SMMUL, SMMULR - SMMUL variant |
| 101 | 001 | != 1111 | SMMLA, SMMLAR - SMMLAR variant |
| 101 | 001 | 1111 | SMMUL, SMMULR - SMMULR variant |
| 101 | 01x | - | Unallocated. |
| 101 | 10x | - | Unallocated. |
| 101 | 110 | - | SMMLS, SMMLSR - SMMLS variant |
| 101 | 111 | - | SMMLS, SMMLSR - SMMLSR variant |
| 11x | - | - | Unallocated. |

### F4.5.7 Unsigned Sum of Absolute Differences

This section describes the encoding of the Unsigned Sum of Absolute Differences instruction class. This section is decoded from *Media instructions* on page F4-2527.

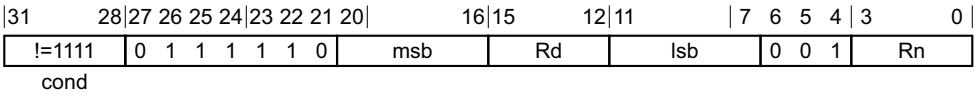

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | Instruction Page |
|---|---|
| **Ra** | |
| != 1111 | USADA8 |
| 1111 | USAD8 |

### F4.5.8 Bitfield Insert

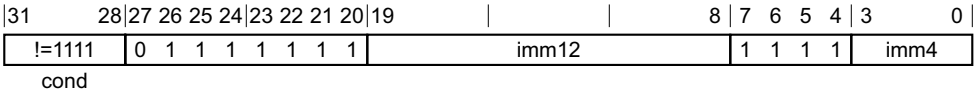
This section describes the encoding of the Bitfield Insert instruction class. This section is decoded from *Media instructions* on page F4-2527.




| Decode fields | Instruction Page |
|---|---|
| **Rn** | |
| != 1111 | BFI |
| 1111 | BFC |

### F4.5.9 Permanently UNDEFINED

This section describes the encoding of the Permanently UNDEFINED instruction class. This section is decoded from *Media instructions* on page F4-2527.

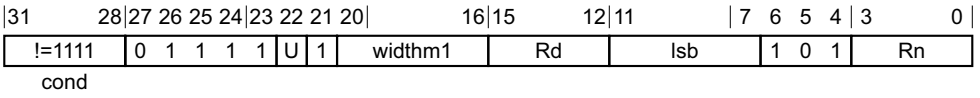



| Decode fields | Instruction Page |
|---|---|
| **cond** | |
| 0xxx | Unallocated. |
| 10xx | Unallocated. |
| 110x | Unallocated. |
| 1110 | UDF |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F4.5.10 Bitfield Extract

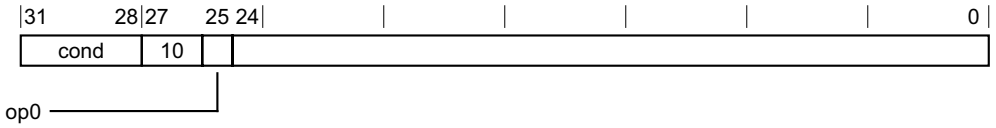
This section describes the encoding of the Bitfield Extract instruction class. This section is decoded from *Media instructions* on page F4-2527.




| Decode fields | Instruction Page |
|---|---|
| U | |
| 0 | SBFX |
| 1 | UBFX |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F4.6 Branch, branch with link, and block data transfer



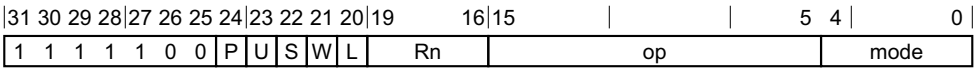

This section describes the encoding of the Branch, branch with link, and block data transfer group. This section is decoded from *Top level A32 instruction set encoding* on page F4-2508.

**Table F4-10 Encoding table for the Branch, branch with link, and block data transfer group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| cond | op0 | |
| 1111 | 0 | *Exception Save/Restore* |
| != 1111 | 0 | *Load/Store Multiple* |
| - | 1 | *Branch (immediate)* on page F4-2536 |

### F4.6.1 Exception Save/Restore

This section describes the encoding of the Exception Save/Restore instruction class. This section is decoded from *Branch, branch with link, and block data transfer*.

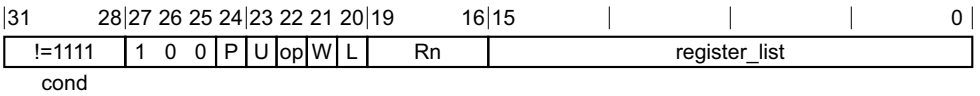



| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| P | U | S | L | |
| - | - | 0 | 0 | Unallocated. |
| 0 | 0 | 0 | 1 | RFE, RFEDA, RFEDB, RFEIA, RFEIB - Decrement After variant |
| 0 | 0 | 1 | 0 | SRS, SRSDA, SRSDB, SRSIA, SRSIB - Decrement After variant |
| 0 | 1 | 0 | 1 | RFE, RFEDA, RFEDB, RFEIA, RFEIB - Increment After variant |
| 0 | 1 | 1 | 0 | SRS, SRSDA, SRSDB, SRSIA, SRSIB - Increment After variant |
| 1 | 0 | 0 | 1 | RFE, RFEDA, RFEDB, RFEIA, RFEIB - Decrement Before variant |
| 1 | 0 | 1 | 0 | SRS, SRSDA, SRSDB, SRSIA, SRSIB - Decrement Before variant |
| - | - | 1 | 1 | Unallocated. |
| 1 | 1 | 0 | 1 | RFE, RFEDA, RFEDB, RFEIA, RFEIB - Increment Before variant |
| 1 | 1 | 1 | 0 | SRS, SRSDA, SRSDB, SRSIA, SRSIB - Increment Before variant |

### F4.6.2 Load/Store Multiple

This section describes the encoding of the Load/Store Multiple instruction class. This section is decoded from *Branch, branch with link, and block data transfer*.

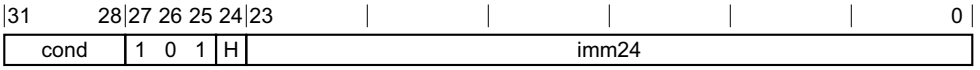

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | | | Instruction Page |
|---|---|---|---|---|---|
| op | P | U | L | register_list | |
| 0 | 0 | 0 | 0 | - | STMDA, STMED |
| 0 | 0 | 0 | 1 | - | LDMDA, LDMFA |
| 0 | 0 | 1 | 0 | - | STM, STMIA, STMEA |
| 0 | 0 | 1 | 1 | - | LDM, LDMIA, LDMFD |
| 0 | 1 | 0 | 0 | - | STMDB, STMFD |
| 0 | 1 | 0 | 1 | - | LDMDB, LDMEA |
| 0 | 1 | 1 | 0 | - | STMIB, STMFA |
| 0 | 1 | 1 | 1 | - | LDMIB, LDMED |
| 1 | - | - | 0 | - | STM (User registers) |
| 1 | - | - | 1 | 0xxxxxxxxxxxxxx | LDM (User registers) |
| 1 | - | - | 1 | 1xxxxxxxxxxxxxx | LDM (exception return) |

### F4.6.3 Branch (immediate)

This section describes the encoding of the Branch (immediate) instruction class. This section is decoded from *Branch, branch with link, and block data transfer* on page F4-2535.




| Decode fields | | Instruction Page |
|---|---|---|
| cond | H | |
| != 1111 | 0 | B |
| != 1111 | 1 | BL, BLX (immediate) - A1 variant |
| 1111 | - | BL, BLX (immediate) - A2 variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F4.7 Coprocessor instructions, and Supervisor Call

This section describes the encoding of the Coprocessor instructions, and Supervisor Call group. This section is decoded from *Top level A32 instruction set encoding* on page F4-2508.

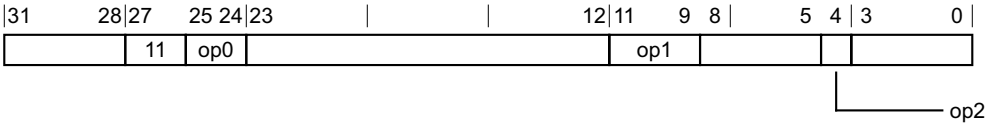



**Table F4-11 Encoding table for the Coprocessor instructions, and Supervisor Call group**

| Decode fields | | | Decode group or instruction page |
|---|---|---|---|
| op0 | op1 | op2 | |
| 0x | 101 | - | For more information about this encoding see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*. |
| 10 | 101 | 0 | For more information about this encoding see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*. |
| 10 | 101 | 1 | For more information about this encoding see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*. |
| 11 | - | - | SVC |
| != 11 | != 101 | - | *Coprocessor instructions* |

### F4.7.1 Supervisor call

This section describes the encoding of the Supervisor call group. This section is decoded from *Coprocessor instructions, and Supervisor Call*.

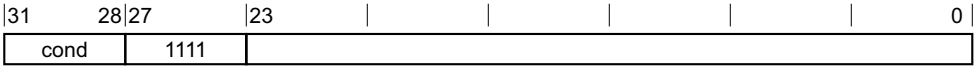



**Table F4-12 Encoding table for the Supervisor call group**

| Decode fields | Decode group or instruction page |
|---|---|
| cond | |
| 1111 | Unallocated. |
| != 1111 | SVC |

### F4.7.2 Coprocessor instructions

This section describes the encoding of the Coprocessor instructions group. This section is decoded from *Coprocessor instructions, and Supervisor Call*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

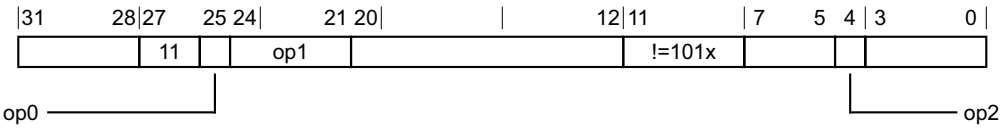
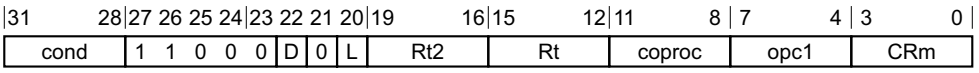



**Table F4-13 Encoding table for the Coprocessor instructions group**

| Decode fields | | | Decode group or instruction page |
|---|---|---|---|
| **op0** | **op1** | **op2** | |
| 0 | 00x0 | - | *Coprocessor 64-bit Move* |
| 0 | != 00x0 | - | *Coprocessor Load/Store* |
| 1 | 0xxx | 0 | CDP, CDP2 |
| 1 | 0xxx | 1 | *Coprocessor 32-bit Move* on page F4-2539 |

## Coprocessor 64-bit Move

This section describes the encoding of the Coprocessor 64-bit Move instruction class. This section is decoded from *Coprocessor instructions* on page F4-2537.

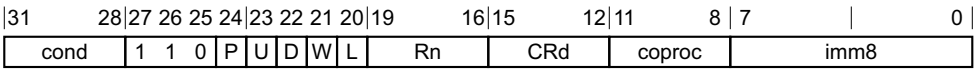



| Decode fields | | | Instruction Page |
|---|---|---|---|
| **cond** | **D** | **L** | |
| != 1111 | 1 | 0 | MCRR, MCRR2 - A1 variant |
| != 1111 | 1 | 1 | MRRC, MRRC2 - A1 variant |
| - | 0 | - | Unallocated. |
| 1111 | 1 | 0 | MCRR, MCRR2 - A2 variant |
| 1111 | 1 | 1 | MRRC, MRRC2 - A2 variant |

## Coprocessor Load/Store

This section describes the encoding of the Coprocessor Load/Store instruction class. This section is decoded from *Coprocessor instructions* on page F4-2537.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

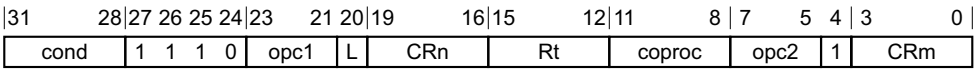



| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| cond | P:U:W | L | Rn | |
| != 1111 | != 000 | 1 | 1111 | LDC, LDC2 (literal) - A1 variant |
| != 1111 | 0x1 | 0 | - | STC, STC2 |
| != 1111 | 0x1 | 1 | != 1111 | LDC, LDC2 (immediate) |
| != 1111 | 010 | 0 | - | STC, STC2 |
| != 1111 | 010 | 1 | != 1111 | LDC, LDC2 (immediate) |
| != 1111 | 1x0 | 0 | - | STC, STC2 |
| != 1111 | 1x0 | 1 | != 1111 | LDC, LDC2 (immediate) |
| != 1111 | 1x1 | 0 | - | STC, STC2 |
| != 1111 | 1x1 | 1 | != 1111 | LDC, LDC2 (immediate) |
| 1111 | != 000 | 1 | 1111 | LDC, LDC2 (literal) - A2 variant |
| 1111 | 0x1 | 0 | - | STC, STC2 |
| 1111 | 0x1 | 1 | != 1111 | LDC, LDC2 (immediate) |
| 1111 | 010 | 0 | - | STC, STC2 |
| 1111 | 010 | 1 | != 1111 | LDC, LDC2 (immediate) |
| 1111 | 1x0 | 0 | - | STC, STC2 |
| 1111 | 1x0 | 1 | != 1111 | LDC, LDC2 (immediate) |
| 1111 | 1x1 | 0 | - | STC, STC2 |
| 1111 | 1x1 | 1 | != 1111 | LDC, LDC2 (immediate) |

### Coprocessor 32-bit Move

This section describes the encoding of the Coprocessor 32-bit Move instruction class. This section is decoded from *Coprocessor instructions* on page F4-2537.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | Instruction Page |
|---|---|---|
| **cond** | **L** | |
| != 1111 | 0 | MCR, MCR2 - A1 variant |
| != 1111 | 1 | MRC, MRC2 - A1 variant |
| 1111 | 0 | MCR, MCR2 - A2 variant |
| 1111 | 1 | MRC, MRC2 - A2 variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F4.8 Unconditional instructions

This section describes the encoding of the Unconditional instructions group. This section is decoded from *Top level A32 instruction set encoding* on page F4-2508.

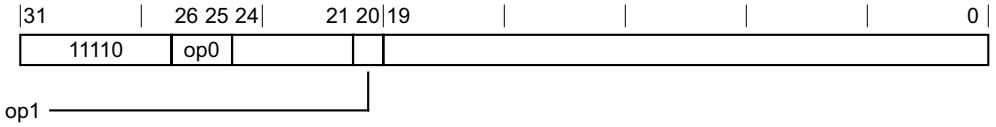



**Table F4-14 Encoding table for the Unconditional instructions group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 00 | - | *Miscellaneous* |
| 01 | - | For more information about this encoding see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*. |
| 1x | 1 | *Memory hints and barriers* on page F4-2542 |
| 10 | 0 | For more information about this encoding see Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*. |
| 11 | 0 | Unallocated. |

### F4.8.1 Miscellaneous

This section describes the encoding of the Miscellaneous group. This section is decoded from *Unconditional instructions*.

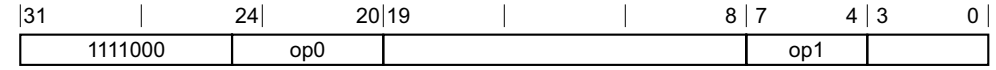



**Table F4-15 Encoding table for the Miscellaneous group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 0xxxx | - | Unallocated. |
| 10000 | xx0x | *Change Process State* on page F4-2542 |
| 10001 | xx0x | Unallocated. |
| 1001x | xx0x | Unallocated. |
| 100xx | 0011 | Unallocated. |
| 100xx | 0111 | UNPREDICTABLE |
| 100xx | 0x10 | Unallocated. |

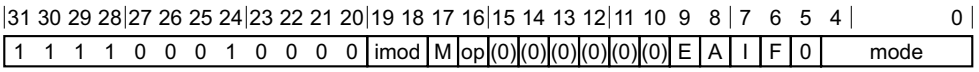

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F4-15 Encoding table for the Miscellaneous group (continued)**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 100xx | 1x1x | Unallocated. |
| 101xx | - | Unallocated. |
| 11xxx | - | Unallocated. |

### Change Process State

This section describes the encoding of the Change Process State instruction class. This section is decoded from *Miscellaneous* on page F4-2541.




| Decode fields | | | | Instruction Page |
|---|---|---|---|---|
| op | imod | M | mode | |
| 0 | - | - | - | CPS, CPSID, CPSIE |
| 1 | - | - | 0xxxx | SETEND |
| 1 | - | - | 1xxxx | Unallocated. |

## F4.8.2 Memory hints and barriers

This section describes the encoding of the Memory hints and barriers group. This section is decoded from *Unconditional instructions* on page F4-2541.

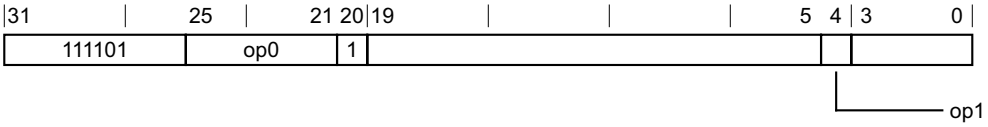



**Table F4-16 Encoding table for the Memory hints and barriers group**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 00x11 | - | Unallocated. |
| 01011 | - | *Barriers* on page F4-2543 |
| 01111 | - | Unallocated. |
| 0xx01 | - | Unallocated. |
| 0xxx0 | - | *Preload (immediate)* on page F4-2543 |

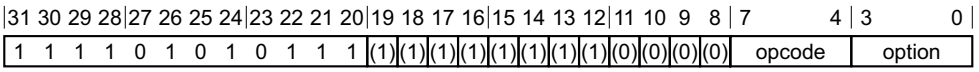

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F4-16 Encoding table for the Memory hints and barriers group (continued)**

| Decode fields | | Decode group or instruction page |
|---|---|---|
| op0 | op1 | |
| 1xxx0 | 0 | *Preload (register)* on page F4-2544 |
| 1xxx1 | 0 | Unallocated. |
| 1xxxx | 1 | Unallocated. |

## Barriers

This section describes the encoding of the Barriers instruction class. This section is decoded from *Memory hints and barriers* on page F4-2542.

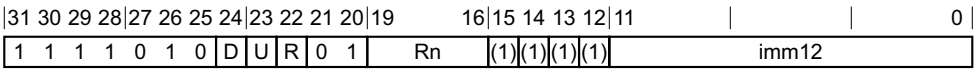



| Decode fields | Instruction Page |
|---|---|
| opcode | |
| 0000 | Unallocated. |
| 0001 | CLREX |
| 001x | Unallocated. |
| 0100 | DSB |
| 0101 | DMB |
| 0110 | ISB |
| 0111 | Unallocated. |
| 1xxx | Unallocated. |

## Preload (immediate)

This section describes the encoding of the Preload (immediate) instruction class. This section is decoded from *Memory hints and barriers* on page F4-2542.

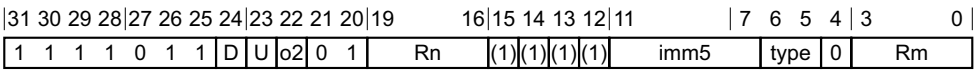

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




| Decode fields | | | Instruction Page |
|---|---|---|---|
| D | R | Rn | |
| 0 | 0 | - | Unallocated. |
| 0 | 1 | - | PLI (immediate, literal) |
| 1 | - | 1111 | PLD (literal) |
| 1 | 0 | != 1111 | PLD, PLDW (immediate) - Preload write variant |
| 1 | 1 | != 1111 | PLD, PLDW (immediate) - Preload read variant |

### Preload (register)

This section describes the encoding of the Preload (register) instruction class. This section is decoded from *Memory hints and barriers* on page F4-2542.




| Decode fields | | Instruction Page |
|---|---|---|
| o2 | D | |
| 0 | 0 | Unallocated. |
| 0 | 1 | PLD, PLDW (register) - Preload write, rotate right with extend variant |
| 1 | 0 | PLI (register) |
| 1 | 1 | PLD, PLDW (register) - Preload read, rotate right with extend variant |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter F5
# T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings

This chapter gives an overview of the Advanced SIMD and floating-point instruction sets. It contains the following sections:


- *Overview* on page F5-2546.
- *Advanced SIMD and floating-point instruction syntax* on page F5-2547.
- *Register encoding* on page F5-2551.
- *Advanced SIMD data-processing instructions* on page F5-2553.
- *Floating-point data-processing instructions* on page F5-2565.
- *Advanced SIMD and floating-point register load/store instructions* on page F5-2568.
- *Advanced SIMD element or structure load/store instructions* on page F5-2569.
- *8, 16, and 32-bit transfers accessing the SIMD and floating-point register file* on page F5-2572.
- *64-bit transfers accessing the SIMD and floating-point register file* on page F5-2573.


**Note**

- The Advanced SIMD architecture, its associated implementations, and supporting software, are commonly referred to as NEON™ technology.
- In the decode tables in this chapter, an entry of - for a field value means the value of the field does not affect the decoding.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# F5.1 Overview

All Advanced SIMD and floating-point instructions are available in both A32 and T32 instruction sets.

## F5.1.1 Advanced SIMD

The following sections describe the classes of Advanced SIMD instructions:

- *Advanced SIMD data-processing instructions* on page F5-2553.
- *Advanced SIMD element or structure load/store instructions* on page F5-2569.
- *Advanced SIMD and floating-point register load/store instructions* on page F5-2568.
- *8, 16, and 32-bit transfers accessing the SIMD and floating-point register file* on page F5-2572.
- *64-bit transfers accessing the SIMD and floating-point register file* on page F5-2573.

## F5.1.2 Floating-point

The following sections describe the classes of floating-point instructions:

- *Advanced SIMD and floating-point register load/store instructions* on page F5-2568.
- *8, 16, and 32-bit transfers accessing the SIMD and floating-point register file* on page F5-2572.
- *64-bit transfers accessing the SIMD and floating-point register file* on page F5-2573.
- *Floating-point data-processing instructions* on page F5-2565.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F5.2 Advanced SIMD and floating-point instruction syntax

Advanced SIMD and floating-point instructions use the general conventions of the T32 and A32 instruction sets.

Advanced SIMD and floating-point data-processing instructions use the following general format:

```
V{<modifier>}<operation>{<shape>}{<c>}{<q>}{.<dt>}  {<dest>,} <src1>, <src2>
```

All Advanced SIMD and floating-point instructions begin with a `V`. This distinguishes Advanced SIMD vector and floating-point instructions from scalar instructions.

The main operation is specified in the `<operation>` field. It is usually a three letter mnemonic the same as or similar to the corresponding scalar integer instruction.

The `<c>` and `<q>` fields are standard assembler syntax fields. For details see *Standard assembler syntax fields* on page F2-2446.

### F5.2.1 Advanced SIMD instruction modifiers

The `<modifier>` field provides additional variants of some instructions. Table F5-1 provides definitions of the modifiers. Modifiers are not available for every instruction.

**Table F5-1 Advanced SIMD instruction modifiers**

| <modifier> | Meaning |
|---|---|
| Q | The operation uses saturating arithmetic. |
| R | The operation performs rounding. |
| D | The operation doubles the result (before accumulation, if any). |
| H | The operation halves the result. |

### F5.2.2 Advanced SIMD operand shapes

The `<shape>` field provides additional variants of some instructions. Table F5-2 provides definitions of the shapes. Operand shapes are not available for every instruction.

**Table F5-2 Advanced SIMD operand shapes**

| <shape> | Meaning | Typical register shape | |
|---|---|---|---|
| (none) | The operands and result are all the same width. | Dd, Dn, Dm | Qd, Qn, Qm |
| L | Long operation - result is twice the width of both operands | Qd, Dn, Dm | |
| N | Narrow operation - result is half the width of both operands | Dd, Qn, Qm | |
| W | Wide operation - result and first operand are twice the width of the second operand | Qd, Qn, Dm | |

**Note**

- Some assemblers support a Q shape specifier, that requires all operands to be Q registers. An example of using this specifier is `VADDQ.S32  q0, q1, q2`. This is not standard UAL, and ARM recommends that programmers do not use a Q shape specifier.
- A disassembler must not generate any shape specifier not shown in Table F5-2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.2.3 Data type specifiers

The <dt> field normally contains one data type specifier. Unless the assembler syntax description for the instruction indicates otherwise, this indicates the data type contained in:

- The second operand, if any.
- The operand, if there is no second operand.
- The result, if there are no operand registers.

The data types of the other operand and result are implied by the <dt> field combined with the instruction shape. For information about data type formats see *Data types supported by the Advanced SIMD implementation* on page E1-2328.

In the instruction syntax descriptions in Chapter F2 *About the T32 and A32 Instruction Descriptions*, the <dt> field is usually specified as a single field. However, where more convenient, it is sometimes specified as a concatenation of two fields, <type><size>.

#### Syntax flexibility

There is some flexibility in the data type specifier syntax:

- Software can specify three data types, specifying the result and both operand data types. For example:

  `VSUBW.I16.I16.S8 Q3, Q5, D0` instead of `VSUBW.S8 Q3, Q5, D0`

- Software can specify two data types, specifying the data types of the two operands. The data type of the result is implied by the instruction shape. For example:

  `VSUBW.I16.S8 Q3, Q5, D0` instead of `VSUBW.S8 Q3, Q5, D0`

- Software can specify two data types, specifying the data types of the single operand and the result. For example:

  `VMOVN.I16.I32 D0, Q1` instead of `VMOVN.I32 D0, Q1`

- Where an instruction requires a less specific data type, software can instead specify a more specific type, as shown in Table F5-3.

- Where an instruction does not require a data type, software can provide one.

- The `F32` data type can be abbreviated to `F`.

- The `F64` data type can be abbreviated to `D`.

In all cases, if software provides additional information, the additional information must match the instruction shape. Disassembly does not regenerate this additional information.

**Table F5-3 Data type specification flexibility**

| Specified data type | Permitted more specific data types | | | | |
|---|---|---|---|---|---|
| None | Any | | | | |
| .I<size> | - | .S<size> | .U<size> | - | - |
| .8 | .I8 | .S8 | .U8 | .P8 | - |
| .16 | .I16 | .S16 | .U16 | .P16 | .F16 |
| .32 | .I32 | .S32 | .U32 | - | .F32 or .F |
| .64 | .I64 | .S64 | .U64 | - | .F64 or .D |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.2.4 Register specifiers

The <dest>, <src1>, and <src2> fields contain register specifiers, or in some cases scalar specifiers or register lists. Table F5-4 shows the register and scalar specifier formats that appear in the instruction descriptions.

If <dest> is omitted, it is the same as <src1>.

**Table F5-4 Advanced SIMD and floating-point register specifier formats**

| <specifier> | Usual meaning [a] | Used in |
|---|---|---|
| <Qd> | A quadword destination register for the result vector. | Advanced SIMD |
| <Qn> | A quadword source register for the first operand vector. | Advanced SIMD |
| <Qm> | A quadword source register for the second operand vector. | Advanced SIMD |
| <Dd> | A doubleword destination register for the result vector. | Both |
| <Dn> | A doubleword source register for the first operand vector. | Both |
| <Dm> | A doubleword source register for the second operand vector. | Both |
| <Sd> | A singleword destination register for the result vector. | Floating-point |
| <Sn> | A singleword source register for the first operand vector. | Floating-point |
| <Sm> | A singleword source register for the second operand vector. | Floating-point |
| <Dd[x]> | A destination scalar for the result. Element x of vector <Dd>. | Advanced SIMD |
| <Dn[x]> | A source scalar for the first operand. Element x of vector <Dn>. | Both [b] |
| <Dm[x]> | A source scalar for the second operand. Element x of vector <Dm>. | Advanced SIMD |
| <Rt> | A general-purpose register, used for a source or destination address. | Both |
| <Rt2> | A general-purpose register, used for a source or destination address. | Both |
| <Rn> | A general-purpose register, used as a load or store base address. | Both |
| <Rm> | A general-purpose register, used as a post-indexed address source. | Both |

a. In some instructions the roles of registers are different.

b. In the floating-point instructions, <Dn[x]> is used only in VMOV (scalar to general-purpose register), see *VMOV (scalar to general-purpose register)* on page F7-3487.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.2.5 Register lists

A register list is a list of register specifiers separated by commas and enclosed in brackets { and }. There are restrictions on what registers can appear in a register list. These restrictions are described in the individual instruction descriptions. Table F5-5 shows some register list formats, with examples of actual register lists corresponding to those formats.

**Note**

Register lists must not wrap around the end of the register bank.

#### Syntax flexibility

There is some flexibility in the register list syntax:

- Where a register list contains consecutive registers, they can be specified as a range, instead of listing every register, for example `{D0-D3}` instead of `{D0, D1, D2, D3}`.
- Where a register list contains an even number of consecutive doubleword registers starting with an even numbered register, it can be written as a list of quadword registers instead, for example `{Q1, Q2}` instead of `{D2-D5}`.
- Where a register list contains only one register, the enclosing braces can be omitted, for example `VLD1.8 D0, [R0]` instead of `VLD1.8 {D0}, [R0]`.

**Table F5-5 Example register lists**

| Format | Example | Alternative |
|---|---|---|
| `{<Dd>}` | `{D3}` | `D3` |
| `{<Dd>, <Dd+1>, <Dd+2>}` | `{D3, D4, D5}` | `{D3-D5}` |
| `{<Dd[x]>, <Dd+2[x]>}` | `{D0[3], D2[3]}` | - |
| `{<Dd[]>}` | `{D7[]}` | `D7[]` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F5.3 Register encoding

An Advanced SIMD register is either:

- *Quadword*, meaning it is 128 bits wide.
- *Doubleword*, meaning it is 64 bits wide.

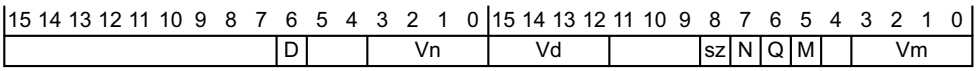
Some instructions have options for either doubleword or quadword registers. This is normally encoded in Q, bit[6], as Q = 0 for doubleword operations, or Q = 1 for quadword operations.

A floating-point register is either:

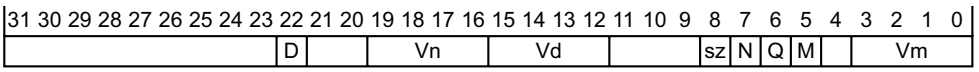
- Double-precision, meaning it is 64 bits wide.
- Single-precision, meaning it is 32 bits wide.

This is encoded in the sz field, bit[8], as sz = 1 for double-precision operations, or sz = 0 for single-precision operations.

The T32 instruction encoding of Advanced SIMD or floating-point registers is:




The A32 instruction encoding of Advanced SIMD or floating-point registers is:




Some instructions use only one or two registers, and use the unused register fields as additional opcode bits.

Table F5-6 shows the encodings for the registers.

**Table F5-6 Encoding of register numbers**

| Register mnemonic | Usual usage | Register number encoded in[a] | Notes[a] | Used in |
|---|---|---|---|---|
| <Qd> | Destination (quadword) | D, Vd (bits[22, 15:13]) | bit[12] == 0[b] | Advanced SIMD |
| <Qn> | First operand (quadword) | N, Vn (bits[7, 19:17]) | bit[16] == 0[b] | Advanced SIMD |
| <Qm> | Second operand (quadword) | M, Vm (bits[5, 3:1]) | bit[0] == 0[b] | Advanced SIMD |
| <Dd> | Destination (doubleword) | D, Vd (bits[22, 15:12]) | - | Both |
| <Dn> | First operand (doubleword) | N, Vn (bits[7, 19:16]) | - | Both |
| <Dm> | Second operand (doubleword) | M, Vm (bits[5, 3:0]) | - | Both |
| <Sd> | Destination (single-precision) | Vd, D (bits[15:12, 22]) | - | Floating-point |
| <Sn> | First operand (single-precision) | Vn, N (bits[19:16, 7]) | - | Floating-point |
| <Sm> | Second operand (single-precision) | Vm, M (bits[3:0, 5]) | - | Floating-point |

a. Bit numbers given for the A32 instruction encoding. See the figures in this section for the equivalent bits in the T32 encoding.

b. If this bit is 1, the instruction is UNDEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.3.1 Advanced SIMD scalars

Advanced SIMD scalars can be 8-bit, 16-bit, 32-bit, or 64-bit. Instructions other than multiply instructions can access any element in the register set. The instruction syntax refers to the scalars using an index into a doubleword vector. The descriptions of the individual instructions contain details of the encodings.

Table F5-7 shows the form of encoding for scalars used in multiply instructions. These instructions cannot access scalars in some registers. The descriptions of the individual instructions contain cross references to this section where appropriate.

32-bit Advanced SIMD scalars, when used as single-precision floating-point numbers, are equivalent to Floating-point single-precision registers. That is, `Dm[x]` in a 32-bit context (0 <= `m` <= 15, 0 <= `x` <=1) is equivalent to `S[2m + x]`.

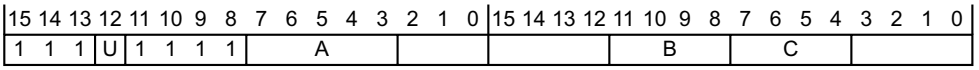
**Table F5-7 Encoding of scalars in multiply instructions**

| Scalar mnemonic | Usual usage | Scalar size | Register specifier | Index specifier | Accessible registers |
|---|---|---|---|---|---|
| `<Dm[x]>` | Second operand | 16-bit | Vm[2:0] | M, Vm[3] | D0-D7 |
| | | 32-bit | Vm[3:0] | M | D0-D15 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F5.4 Advanced SIMD data-processing instructions

The T32 encoding of Advanced SIMD data processing instructions is:




The A32 encoding of Advanced SIMD data processing instructions is:

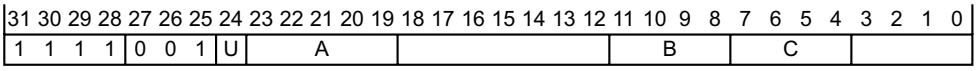



Table F5-8 shows the encoding for Advanced SIMD data-processing instructions. Other encodings in this space are UNDEFINED.

In these instructions, the U bit is in a different location in A32 and T32 instructions. This is bit[12] of the first halfword in the T32 encoding, and bit[24] in the A32 encoding. Other variable bits are in identical locations in the two encodings, after adjusting for the fact that the A32 encoding is held in memory as a single word and the T32 encoding is held as two consecutive halfwords.

The A32 instructions can only be executed unconditionally. The T32 instructions can be executed conditionally by using the `IT` instruction. For details see *IT* on page F6-2690.

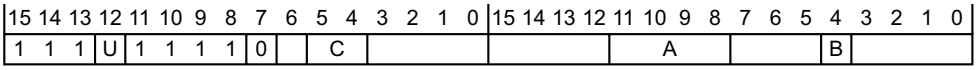
**Table F5-8 Data-processing instructions**

| U | A | B | C | See |
|---|---|---|---|---|
| - | 0xxxx | - | - | *Three registers of the same length* on page F5-2554 |
| | 1x000 | - | 0xx1 | *One register and a modified immediate value* on page F5-2562 |
| | 1x001 | - | 0xx1 | *Two registers and a shift amount* on page F5-2559 |
| | 1x01x | - | 0xx1 | |
| | 1x1xx | - | 0xx1 | |
| | 1xxxx | - | 1xx1 | |
| | 1x0xx | - | x0x0 | *Three registers of different lengths* on page F5-2557 |
| | 1x10x | - | x0x0 | |
| | 1x0xx | - | x1x0 | *Two registers and a scalar* on page F5-2558 |
| | 1x10x | - | x1x0 | |
| 0 | 1x11x | - | xxx0 | Vector Extract, *VEXT (byte elements)* on page F7-3376 |
| 1 | 1x11x | 0xxx | xxx0 | *Two registers, miscellaneous* on page F5-2560 |
| | | 10xx | xxx0 | Vector Table Lookup, *VTBL, VTBX* on page F7-3729 |
| | | 1100 | 0xx0 | Vector Duplicate, *VDUP (scalar)* on page F7-3372 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.4.1 Three registers of the same length

The T32 encoding of these instructions is:




The A32 encoding of these instructions is:

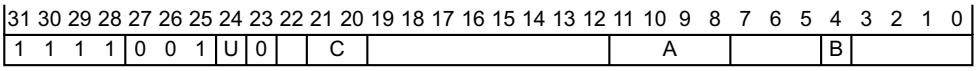



Table F5-9 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

**Table F5-9 Three registers of the same length**

| A | B | U | C | Instruction | See |
|---|---|---|---|---|---|
| 0000 | 0 | - | - | Vector Halving Add | *VHADD* on page F7-3390 |
| | 1 | - | - | Vector Saturating Add | *VQADD* on page F7-3555 |
| 0001 | 0 | - | - | Vector Rounding Halving Add | *VRHADD* on page F7-3615 |
| | 1 | 0 | 00 | Vector Bitwise AND | *VAND (register)* on page F7-3275 |
| | | | 01 | Vector Bitwise Bit Clear, AND complement | *VBIC (register)* on page F7-3279 |
| | | | 10 | Vector Bitwise OR[a] | *VORR (register)* on page F7-3531 |
| | | | 11 | Vector Bitwise OR NOT | *VORN (register)* on page F7-3527 |
| | | 1 | 00 | Vector Bitwise Exclusive OR | *VEOR* on page F7-3374 |
| | | | 01 | Vector Bitwise Select | *VBSL* on page F7-3285 |
| | | | 10 | Vector Bitwise Insert if True | *VBIT* on page F7-3283 |
| | | | 11 | Vector Bitwise Insert if False | *VBIF* on page F7-3281 |
| 0010 | 0 | - | - | Vector Halving Subtract | *VHSUB* on page F7-3392 |
| | 1 | - | - | Vector Saturating Subtract | *VQSUB* on page F7-3598 |
| 0011 | 0 | - | - | Vector Compare Greater Than | *VCGT (register)* on page F7-3300 |
| | 1 | - | - | Vector Compare Greater Than or Equal | *VCGE (register)* on page F7-3294 |
| 0100 | 0 | - | - | Vector Shift Left | *VSHL (register)* on page F7-3666 |
| | 1 | - | - | Vector Saturating Shift Left | *VQSHL (register)* on page F7-3589 |
| 0101 | 0 | - | - | Vector Rounding Shift Left | *VSHL (register)* on page F7-3666 |
| | 1 | - | - | Vector Saturating Rounding Shift Left | *VQRSHL* on page F7-3577 |
| 0110 | 0 | - | - | Vector Maximum | *VMAX (integer)* on page F7-3438 |
| | 1 | - | - | Vector Minimum | *VMIN (integer)* on page F7-3445 |
| 0111 | 0 | - | - | Vector Absolute Difference | *VABD (integer)* on page F7-3247 |
| | 1 | - | - | Vector Absolute Difference and Accumulate | *VABA* on page F7-3241 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F5-9 Three registers of the same length (continued)**

| A | B | U | C | Instruction | See |
|---|---|---|---|---|---|
| 1000 | 0 | 0 | - | Vector Add | *VADD (integer)* on page F7-3265 |
| | | 1 | - | Vector Subtract | *VSUB (integer)* on page F7-3719 |
| | 1 | 0 | - | Vector Test Bits | *VTST* on page F7-3734 |
| | | 1 | - | Vector Compare Equal | *VCEQ (register)* on page F7-3289 |
| 1001 | 0 | 0 | - | Vector Multiply Accumulate | *VMLA (integer)* on page F7-3453 |
| | | 1 | | Vector Multiply Subtract | *VMLS (integer)* on page F7-3465 |
| | 1 | - | - | Vector Multiply | *VMUL (integer and polynomial)* on page F7-3502 |
| 1010 | 0 | - | - | Vector Pairwise Maximum | *VPMAX (integer)* on page F7-3543 |
| | 1 | - | - | Vector Pairwise Minimum | *VPMIN (integer)* on page F7-3547 |
| 1011 | 0 | 0 | - | Vector Saturating Doubling Multiply Returning High Half | *VQDMULH* on page F7-3563 |
| | | 1 | - | Vector Saturating Rounding Doubling Multiply Returning High Half | *VQRDMULH* on page F7-3574 |
| | 1 | 0 | - | Vector Pairwise Add | *VPADD (integer)* on page F7-3537 |
| 1100 | 0 | 0 | 00 | SHA1 Hash Update (Choose) | *SHA1C* on page F7-3224 |
| | | | 01 | SHA1 Hash Update (Parity) | *SHA1P* on page F7-3229 |
| | | | 10 | SHA1 Hash Update (Majority) | *SHA1M* on page F7-3227 |
| | | | 11 | SHA1 Schedule Update 0 | *SHA1SU0* on page F7-3231 |
| | | 1 | 00 | SHA256 Hash Update (part 1) | *SHA256H* on page F7-3235 |
| | | | 01 | SHA256 Hash Update (part 2) | *SHA256H2* on page F7-3236 |
| | | | 10 | SHA256 Schedule Update 1 | *SHA256SU1* on page F7-3239 |
| | 1 | 0 | 00 | Vector Fused Multiply Accumulate | *VFMA* on page F7-3380 |
| | | | 10 | Vector Fused Multiply Subtract | *VFMS* on page F7-3383 |
| 1101 | 0 | 0 | 0x | Vector Add | *VADD (floating-point)* on page F7-3262 |
| | | | 1x | Vector Subtract | *VSUB (floating-point)* on page F7-3716 |
| | | 1 | 0x | Vector Pairwise Add | *VPADD (floating-point)* on page F7-3535 |
| | | | 1x | Vector Absolute Difference | *VABD (floating-point)* on page F7-3245 |
| | 1 | 0 | 00 | Vector Multiply Accumulate | *VMLA (floating-point)* on page F7-3450 |
| | | | 10 | Vector Multiply Subtract | *VMLS (floating-point)* on page F7-3462 |
| | | 1 | 0x | Vector Multiply | *VMUL (floating-point)* on page F7-3499 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F5-9 Three registers of the same length (continued)**

| A | B | U | C | Instruction | See |
|---|---|---|---|---|---|
| 1110 | 0 | 0 | 0x | Vector Compare Equal | *VCEQ (register)* on page F7-3289 |
| | | 1 | 0x | Vector Compare Greater Than or Equal[b] | *VCGE (register)* on page F7-3294[b] |
| | | | 1x | Vector Compare Greater Than[c] | *VCGT (register)* on page F7-3300[c] |
| | 1 | 1 | 00 | Vector Absolute Compare Greater Than or Equal[d] | *VACGE* on page F7-3254[d] |
| | | | 10 | Vector Absolute Compare Greater Than[e] | *VACGT* on page F7-3258[e] |
| 1111 | 0 | 0 | 00 | Vector Maximum | *VMAX (floating-point)* on page F7-3436 |
| | | | 10 | Vector Minimum | *VMIN (floating-point)* on page F7-3443 |
| | | 1 | 00 | Vector Pairwise Maximum | *VPMAX (floating-point)* on page F7-3541 |
| | | | 10 | Vector Pairwise Minimum | *VPMIN (floating-point)* on page F7-3545 |
| | 1 | 0 | 0x | Vector Reciprocal Step | *VRECPS* on page F7-3604 |
| | | | 1x | Vector Reciprocal Square Root Step | *VRSQRTS* on page F7-3655 |
| | | 1 | 00 | Floating-point Maximum Number | *VMAXNM* on page F7-3440 |
| | | | 10 | Floating-point Minimum Number | *VMINNM* on page F7-3447 |

a. Previous descriptions of the A32/T32 Advanced SIMD encodings have included `VMOV` (register, SIMD). This is an alias of an encoding of `VORR` (register), and its encoding is now described in *VORR (register)* on page F7-3531.

b. Vector Compare Less Than or Equal, `VCLE`, is an alias of `VCGE` and is described in the same section.

c. Vector Compare Less Than, `VCLT`, is an alias of `VCGT` and is described in the same section.

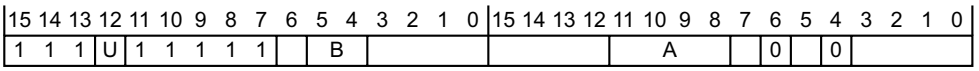
d. Vector Absolute Compare Less Than or Equal, `VACLE`, is an alias of `VACGE` and is described in the same section.

e. Vector Absolute Compare Less Than, `VACLT`, is an alias of `VACGT` and is described in the same section.

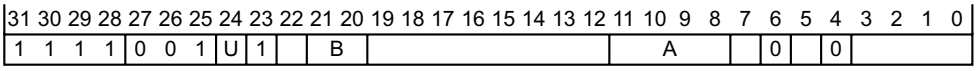

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.4.2 Three registers of different lengths

The T32 encoding of these instructions is:




The A32 encoding of these instructions is:




If B == `0b11`, see *Advanced SIMD data-processing instructions* on page F5-2553.

Otherwise, Table F5-10 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

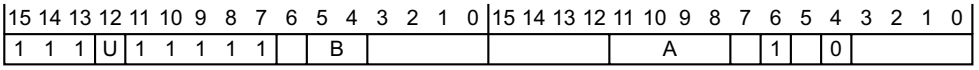
**Table F5-10 Data-processing instructions with three registers of different lengths**

| A | U | Instruction | See |
|---|---|---|---|
| 0000 | - | Vector Add Long | *VADDL* on page F7-3269 |
| 0001 | - | Vector Add Wide | *VADDW* on page F7-3271 |
| 0010 | - | Vector Subtract Long | *VSUBL* on page F7-3723 |
| 0011 | - | Vector Subtract Wide | *VSUBW* on page F7-3725 |
| 0100 | 0 | Vector Add and Narrow, returning High Half | *VADDHN* on page F7-3267 |
| | 1 | Vector Rounding Add and Narrow, returning High Half | *VRADDHN* on page F7-3600 |
| 0101 | - | Vector Absolute Difference and Accumulate Long | *VABAL* on page F7-3243 |
| 0110 | 0 | Vector Subtract and Narrow, returning High Half | *VSUBHN* on page F7-3721 |
| | 1 | Vector Rounding Subtract and Narrow, returning High Half | *VRSUBHN* on page F7-3659 |
| 0111 | - | Vector Absolute Difference Long | *VABDL (integer)* on page F7-3249 |
| 1000 | - | Vector Multiply Accumulate Long | *VMLAL (integer)* on page F7-3458 |
| 1010 | - | Vector Multiply Subtract Long | *VMLSL (integer)* on page F7-3470 |
| 1001 | 0 | Vector Saturating Doubling Multiply Accumulate Long | *VQDMLAL* on page F7-3557 |
| 1011 | 0 | Vector Saturating Doubling Multiply Subtract Long | *VQDMLSL* on page F7-3560 |
| 11x0 | - | Vector Multiply Long | *VMULL (integer and polynomial)* on page F7-3508 |
| 1101 | 0 | Vector Saturating Doubling Multiply Long | *VQDMULL* on page F7-3566 |

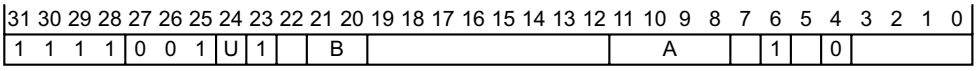

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.4.3 Two registers and a scalar

The T32 encoding of these instructions is:




The A32 encoding of these instructions is:




If B == `0b11`, see *Advanced SIMD data-processing instructions* on page F5-2553.

Otherwise, Table F5-11 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

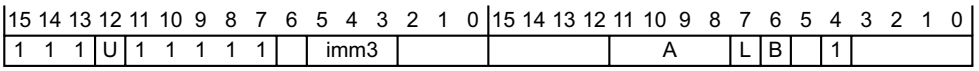
**Table F5-11 Data-processing instructions with two registers and a scalar**

| A | U | Instruction | See |
|---|---|---|---|
| 000x | - | Vector Multiply Accumulate | *VMLA (by scalar)* on page F7-3455 |
| 010x | - | Vector Multiply Subtract | *VMLS (by scalar)* on page F7-3467 |
| 0010 | - | Vector Multiply Accumulate | *VMLAL (by scalar)* on page F7-3460 |
| 0110 | - | Vector Multiply Subtract Long | *VMLSL (by scalar)* on page F7-3472 |
| 0011 | 0 | Vector Saturating Doubling Multiply Accumulate Long | *VQDMLAL* on page F7-3557 |
| 0111 | 0 | Vector Saturating Doubling Multiply Subtract Long | *VQDMLSL* on page F7-3560 |
| 100x | - | Vector Multiply | *VMUL (by scalar)* on page F7-3505 |
| 1010 | - | Vector Multiply Long | *VMULL (by scalar)* on page F7-3510 |
| 1011 | 0 | Vector Saturating Doubling Multiply Long | *VQDMULL* on page F7-3566 |
| 1100 | - | Vector Saturating Doubling Multiply returning High Half | *VQDMULH* on page F7-3563 |
| 1101 | - | Vector Saturating Rounding Doubling Multiply returning High Half | *VQRDMULH* on page F7-3574 |

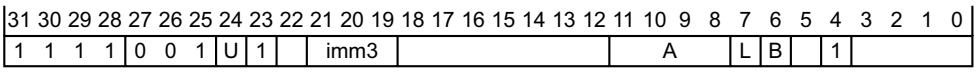

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.4.4 Two registers and a shift amount

The T32 encoding of these instructions is:




The A32 encoding of these instructions is:




If [L, imm3] == `0b0000`, see *One register and a modified immediate value* on page F5-2562.

Otherwise, Table F5-12 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

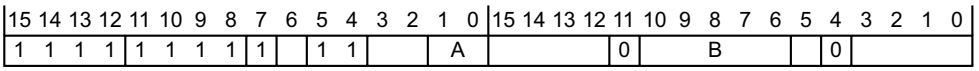
**Table F5-12 Data-processing instructions with two registers and a shift amount**

| A | U | B | L | Instruction | See |
|---|---|---|---|---|---|
| 0000 | - | - | - | Vector Shift Right | *VSHR* on page F7-3671 |
| 0001 | - | - | - | Vector Shift Right and Accumulate | *VSRA* on page F7-3683 |
| 0010 | - | - | - | Vector Rounding Shift Right | *VRSHR* on page F7-3645 |
| 0011 | - | - | - | Vector Rounding Shift Right and Accumulate | *VRSRA* on page F7-3657 |
| 0100 | 1 | - | - | Vector Shift Right and Insert | *VSRI* on page F7-3685 |
| 0101 | 0 | - | - | Vector Shift Left | *VSHL (immediate)* on page F7-3664 |
| | 1 | - | - | Vector Shift Left and Insert | *VSLI* on page F7-3679 |
| 011x | - | - | - | Vector Saturating Shift Left | *VQSHL, VQSHLU (immediate)* on page F7-3586 |
| 1000 | 0 | 0 | 0 | Vector Shift Right Narrow | *VSHRN* on page F7-3675 |
| | | 1 | 0 | Vector Rounding Shift Right Narrow | *VRSHRN* on page F7-3649 |
| | 1 | 0 | 0 | Vector Saturating Shift Right, Unsigned Narrow | *VQSHRN, VQSHRUN* on page F7-3593 |
| | | 1 | 0 | Vector Saturating Shift Right, Rounded Unsigned Narrow | *VQRSHRN, VQRSHRUN* on page F7-3581 |
| 1001 | - | 0 | 0 | Vector Saturating Shift Right, Narrow | *VQSHRN, VQSHRUN* on page F7-3593 |
| | | 1 | 0 | Vector Saturating Shift Right, Rounded Narrow | *VQRSHRN, VQRSHRUN* on page F7-3581 |
| 1010 | - | 0 | 0 | Vector Shift Left Long | *VSHLL* on page F7-3668 |
| | | | | Vector Move Long | *VMOVL* on page F7-3491 |
| 111x | - | - | 0 | Vector Convert | *VCVT (between floating-point and fixed-point, Advanced SIMD)* on page F7-3337 |

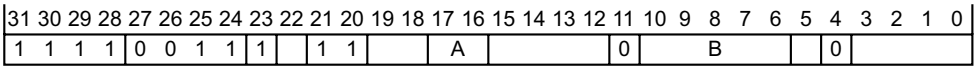

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.4.5 Two registers, miscellaneous

The T32 encoding of these instructions is:




The A32 encoding of these instructions is:




The allocation of encodings in this space is shown in Table F5-13. Other encodings in this space are UNDEFINED.

**Table F5-13 Instructions with two registers, miscellaneous**

| A | B | Instruction | See |
|---|---|---|---|
| 00 | 0000x | Vector Reverse in doublewords | *VREV64* on page F7-3612 |
| | 0001x | Vector Reverse in words | *VREV32* on page F7-3609 |
| | 0010x | Vector Reverse in halfwords | *VREV16* on page F7-3606 |
| | 010xx | Vector Pairwise Add Long | *VPADDL* on page F7-3539 |
| | 01100 | AES Single Round Encryption | *AESE* on page F7-3211 |
| | 01101 | AES Single Round Decryption | *AESD* on page F7-3209 |
| | 01110 | AES Mix Columns | *AESMC* on page F7-3214 |
| | 01111 | AES Inverse Mix Columns | *AESIMC* on page F7-3213 |
| | 1000x | Vector Count Leading Sign Bits | *VCLS* on page F7-3309 |
| | 1001x | Vector Count Leading Zeros | *VCLZ* on page F7-3316 |
| | 1010x | Vector Count | *VCNT* on page F7-3324 |
| | 1011x | Vector Bitwise NOT | *VMVN (register)* on page F7-3514 |
| | 110xx | Vector Pairwise Add and Accumulate Long | *VPADAL* on page F7-3533 |
| 00 | 1110x | Vector Saturating Absolute | *VQABS* on page F7-3553 |
| | 1111x | Vector Saturating Negate | *VQNEG* on page F7-3572 |
| 01 | x000x | Vector Compare Greater Than Zero | *VCGT (immediate #0)* on page F7-3298 |
| | x001x | Vector Compare Greater Than or Equal to Zero | *VCGE (immediate #0)* on page F7-3292 |
| | x010x | Vector Compare Equal to zero | *VCEQ (immediate #0)* on page F7-3287 |
| | x011x | Vector Compare Less Than or Equal to Zero | *VCLE (immediate #0)* on page F7-3304 |
| | x100x | Vector Compare Less Than Zero | *VCLT (immediate #0)* on page F7-3311 |
| | x110x | Vector Absolute | *VABS* on page F7-3251 |
| | x111x | Vector Negate | *VNEG* on page F7-3516 |
| | 01011 | SHA1 Fixed Rotate | *SHA1H* on page F7-3226 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

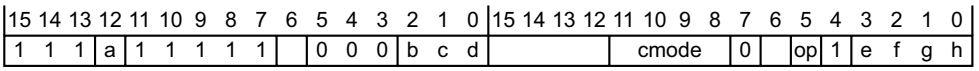
**Table F5-13 Instructions with two registers, miscellaneous (continued)**

| A | B | Instruction | See |
|---|---|---|---|
| 10 | 0000x | Vector Swap | *VSWP* on page F7-3727 |
| | 0001x | Vector Transpose | *VTRN* on page F7-3731 |
| | 0010x | Vector Unzip | *VUZP* on page F7-3736 |
| | 0011x | Vector Zip | *VZIP* on page F7-3739 |
| | 01000 | Vector Move and Narrow | *VMOVN* on page F7-3493 |
| | 01001 | Vector Saturating Move and Unsigned Narrow | *VQMOVN, VQMOVUN* on page F7-3569 |
| | 0101x | Vector Saturating Move and Narrow | *VQMOVN, VQMOVUN* on page F7-3569 |
| | 01100 | Vector Shift Left Long (maximum shift) | *VSHLL* on page F7-3668 |
| | 01110 | SHA1 Schedule Update 1 | *SHA1SU1* on page F7-3233 |
| | 01111 | SHA256 Schedule Update 0 | *SHA1SU0* on page F7-3231 |
| | 1000x | Vector Round to Integer to Nearest with Ties to Even | *VRINTN (Advanced SIMD)* on page F7-3625 |
| | 1001x | Vector Round to Integer Inexact | *VRINTX (Advanced SIMD)* on page F7-3635 |
| | 1010x | Vector Round to Integer to Nearest with Ties to Away | *VRINTA (Advanced SIMD)* on page F7-3617 |
| | 1011x | Vector Round to Integer towards Zero | *VRINTZ (Advanced SIMD)* on page F7-3639 |
| | 1101x | Vector Round to Integer towards -Infinity | *VRINTM (Advanced SIMD)* on page F7-3621 |
| | 11x00 | Vector Convert between Half-precision and Single-precision | *VCVT (between half-precision and single-precision, Advanced SIMD)* on page F7-3328 |
| | 1111x | Vector Round to Integer towards +Infinity | *VRINTP (Advanced SIMD)* on page F7-3629 |
| 11 | 000xx | Vector Convert Floating-point to Integer with Round to Nearest with Ties to Away | *VCVTA (Advanced SIMD)* on page F7-3343 |
| | 001xx | Vector Convert Floating-point to Integer with Round to Nearest | *VCVTN (Advanced SIMD)* on page F7-3354 |
| | 010xx | Vector Convert Floating-point to Integer with Round towards +Infinity | *VCVTP (Advanced SIMD)* on page F7-3358 |
| | 011xx | Vector Convert Floating-point to Integer with Round towards -Infinity | *VCVTM (Advanced SIMD)* on page F7-3350 |
| | 10x0x | Vector Reciprocal Estimate | *VRECPE* on page F7-3602 |
| | 10x1x | Vector Reciprocal Square Root Estimate | *VRSQRTE* on page F7-3653 |
| | 11xxx | Vector Convert between Floating-point and Integer | *VCVT (between floating-point and integer, Advanced SIMD)* on page F7-3330 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.4.6 One register and a modified immediate value

The T32 encoding of these instructions is:




The A32 encoding of these instructions is:

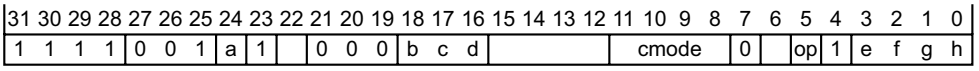



Table F5-14 shows the allocation of encodings in this space.

**Table F5-14 Data-processing instructions with one register and a modified immediate value**

| op | cmode | Instruction | See |
|---|---|---|---|
| 0 | 0xx0 | Vector Move | *VMOV (immediate)* on page F7-3476 |
| | 0xx1 | Vector Bitwise OR | *VORR (immediate)* on page F7-3529 |
| | 10x0 | Vector Move | *VMOV (immediate)* on page F7-3476 |
| | 10x1 | Vector Bitwise OR | *VORR (immediate)* on page F7-3529 |
| | 11xx | Vector Move | *VMOV (immediate)* on page F7-3476 |
| 1 | 0xx0 | Vector Bitwise NOT | *VMVN (immediate)* on page F7-3512 |
| | 0xx1 | Vector Bit Clear | *VBIC (immediate)* on page F7-3277 |
| | 10x0 | Vector Bitwise NOT | *VMVN (immediate)* on page F7-3512 |
| | 10x1 | Vector Bit Clear | *VBIC (immediate)* on page F7-3277 |
| | 110x | Vector Bitwise NOT | *VMVN (immediate)* on page F7-3512 |
| | 1110 | Vector Move | *VMOV (immediate)* on page F7-3476 |
| | 1111 | UNDEFINED | - |

Table F5-15 shows the modified immediate constants available with these instructions, and how they are encoded.

**Table F5-15 Modified immediate values for Advanced SIMD instructions**

| op | cmode | Constant[a] | <dt>[b] | Notes |
|---|---|---|---|---|
| - | 000x | `00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh` | I32 | c |
| | 001x | `00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000` | I32 | c, d |
| | 010x | `00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000` | I32 | c, d |
| | 011x | `abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000` | I32 | c, d |
| | 100x | `00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh` | I16 | c |
| | 101x | `abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000` | I16 | c, d |
| | 1100 | `00000000 00000000 abcdefgh 11111111 00000000 00000000 abcdefgh 11111111` | I32 | d, e |
| | 1101 | `00000000 abcdefgh 11111111 11111111 00000000 abcdefgh 11111111 11111111` | I32 | d, e |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F5-15 Modified immediate values for Advanced SIMD instructions (continued)**

| op | cmode | Constant[a] | <dt>[b] | Notes |
|---|---|---|---|---|
| 0 | 1110 | abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh | I8 | f |
| | 1111 | aBbbbbbc defgh000 00000000 00000000 aBbbbbbc defgh000 00000000 00000000 | F32 | f, g |
| 1 | 1110 | aaaaaaaa bbbbbbbb cccccccc dddddddd eeeeeeee ffffffff gggggggg hhhhhhhh | I64 | f |
| | 1111 | UNDEFINED | - | - |

a. In this table, the immediate value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembler syntax, the constant is specified by a data type and a value of that type. That value is specified in the normal way (a decimal number by default) and is replicated enough times to fill the 64-bit immediate. For example, a data type of `I32` and a value of 10 specify the 64-bit constant `0x0000000A0000000A`.

b. This specifies the data type used when the instruction is disassembled. On assembly, the data type must be matched in the table if possible. Other data types are permitted as pseudo-instructions when a program is assembled, provided the 64-bit constant specified by the data type and value is available for the instruction. If a constant is available in more than one way, the first entry in this table that can produce it is used. For example, `VMOV.I64 D0, #0x8000000080000000` does not specify a 64-bit constant that is available from the I64 line of the table, but does specify one that is available from the fourth I32 line or the F32 line. It is assembled to the first of these, and therefore is disassembled as `VMOV.I32 D0, #0x80000000`.

c. This constant is available for the `VBIC`, `VMOV`, `VMVN`, and `VORR` instructions.

d. UNPREDICTABLE if `abcdefgh` == 00000000.

e. This constant is available for the `VMOV` and `VMVN` instructions only.

f. This constant is available for the `VMOV` instruction only.

g. In this entry, `B` = NOT(`b`). The bit pattern represents the floating-point number $(-1)^S \times 2^{exp} \times$ mantissa, where S = `UInt(a)`, exp = `UInt(NOT(b):c:d)-3` and mantissa = `(16+UInt(e:f:g:h))/16`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Advanced SIMD expand immediate pseudocode

```
// AdvSIMDExpandImm()
// ==================

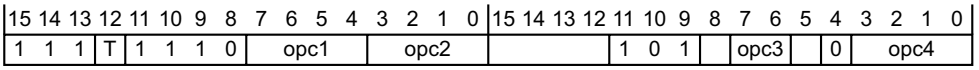
bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);

    return imm64;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F5.5 Floating-point data-processing instructions

The T32 encoding of floating-point data processing instructions is:




The A32 encoding of floating-point data processing instructions is:

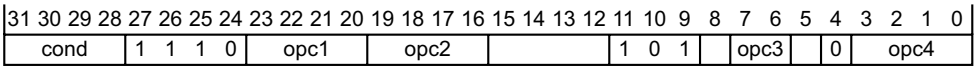



Table F5-16 shows the decode of these instructions. Other encodings in this space are UNDEFINED.

Table F5-16 shows the full decode of encodings for three-register floating-point data-processing instructions. *Other floating-point data processing instructions* on page F5-2566 shows the additional decoding of the other floating-point data-processing instructions.

These instructions are `CDP` instructions for coprocessors 10 and 11.

**Table F5-16 Three-register floating-point data-processing instructions**

| T[a] | opc1 | opc3 | Instruction | See |
|---|---|---|---|---|
| 0 | 0x00 | - | Vector Multiply Accumulate | *VMLA (floating-point)* on page F7-3450 |
| | 0x10 | - | Vector Multiply Subtract | *VMLS (floating-point)* on page F7-3462 |
| | 0x01 | x1 | Vector Negate Multiply Accumulate | *VNMLA* on page F7-3519 |
| | | x0 | Vector Negate Multiply Subtract | *VNMLS* on page F7-3521 |
| | 0x10 | x1 | Vector Negate Multiply | *VNMUL* on page F7-3523 |
| | | x0 | Vector Multiply | *VMUL (floating-point)* on page F7-3499 |
| | 0x11 | x0 | Vector Add | *VADD (floating-point)* on page F7-3262 |
| | | x1 | Vector Subtract | *VSUB (floating-point)* on page F7-3716 |
| | 1x00 | x0 | Vector Divide | *VDIV* on page F7-3368 |
| | 1x01 | x1 | Vector Fused Negate Multiply Accumulate | *VFNMA* on page F7-3386 |
| | | x0 | Vector Fused Negate Multiply Subtract | *VFNMS* on page F7-3388 |
| | 1x10 | x0 | Vector Fused Multiply Accumulate | *VFMA* on page F7-3380 |
| | | x1 | Vector Fused Multiply Subtract | *VFMS* on page F7-3383 |
| 1 | 0xxx | x0 | Floating-point Selection | *VSELEQ, VSELGE, VSELGT, VSELVS* on page F7-3661 |
| | 1x00 | x0 | Floating-point Maximum Number | *VMAXNM* on page F7-3440 |
| | | x1 | Floating-point Minimum Number | *VMINNM* on page F7-3447 |
| x | 1x11 | - | Other floating-point data-processing instructions | *Other floating-point data processing instructions* on page F5-2566 |

a. See Table F5-17 on page F5-2566 for the interpretation of this bit.

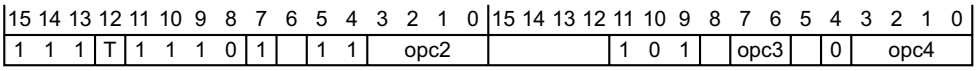

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F5-17 Interpretation of the T bit in Table F5-16 on page F5-2565**

| T | Meaning for T32 encoding | Meaning for A32 encoding |
|---|---|---|
| 0 | The value of the T bit is 0 | The value of the cond field is not 0b1111 |
| 1 | The value of the T bit is 1 | The value of the cond field is 0b1111 |

### F5.5.1 Other floating-point data processing instructions

The T32 encoding of these instructions is:




The A32 encoding of these instructions is:

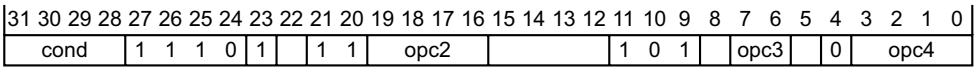



Table F5-18 shows the decode of these instructions. Other encodings in this space are UNDEFINED.

**Table F5-18 Other floating-point data-processing instructions**

| T[a] | opc2 | opc3 | Instruction | See |
|---|---|---|---|---|
| 0 | - | x0 | Vector Move | *VMOV (immediate)* on page F7-3476 |
| | 0000 | 01 | Vector Move | *VMOV (register)* on page F7-3479 |
| | | 11 | Vector Absolute | *VABS* on page F7-3251 |
| | 0001 | 01 | Vector Negate | *VNEG* on page F7-3516 |
| | | 11 | Vector Square Root | *VSQRT* on page F7-3681 |
| | 001x | 01 | Vector Convert | *VCVTB* on page F7-3347 |
| | | 11 | Vector Convert | *VCVTT* on page F7-3365 |
| | 010x | 01 | Vector Compare | *VCMP* on page F7-3318 |
| | | 11 | Vector Compare, raising Invalid Operation on NaN | *VCMPE* on page F7-3321 |
| | 0110 | 01 | Floating-point Round to Integer | *VRINTR* on page F7-3633 |
| | | 11 | Floating-point Round to Integer towards Zero | *VRINTZ (floating-point)* on page F7-3641 |
| | 0111 | 01 | Floating-point Round to Integer | *VRINTX (floating-point)* on page F7-3637 |
| | | 11 | Vector Convert | *VCVT (between double-precision and single-precision)* on page F7-3326 |
| | 1000 | x1 | Vector Convert | *VCVT (integer to floating-point, floating-point)* on page F7-3335 |
| | 1x1x | x1 | Vector Convert | *VCVT (between floating-point and fixed-point, floating-point)* on page F7-3340 |
| | 110x | 01 | Vector Convert | *VCVTR* on page F7-3362 |
| | | 11 | Vector Convert | *VCVT (floating-point to integer, floating-point)* on page F7-3332 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F5-18 Other floating-point data-processing instructions (continued)**

| T[a] | opc2 | opc3 | Instruction | See |
|---|---|---|---|---|
| 1 | 1000 | 01 | Floating-point Round to Integer to Nearest with Ties to Away | *VRINTA (floating-point)* on page F7-3619 |
| | 1001 | 01 | Floating-point Round to Integer to Nearest with Ties to Even | *VRINTN (floating-point)* on page F7-3627 |
| | 1010 | 01 | Floating-point Round to Integer towards +Infinity | *VRINTP (floating-point)* on page F7-3631 |
| | 1011 | 01 | Floating-point Round to Integer towards -Infinity | *VRINTM (floating-point)* on page F7-3623 |
| | 1100 | x1 | Floating-point Convert to Integer with Round to Nearest with Ties to Away | *VCVTA (floating-point)* on page F7-3345 |
| | 1101 | x1 | Floating-point Convert to Integer with Round to Nearest | *VCVTN (floating-point)* on page F7-3356 |
| | 1110 | x1 | Floating-point Convert to Integer with Round towards +Infinity | *VCVTP (floating-point)* on page F7-3360 |
| | 1111 | x1 | Floating-point Convert to Integer with Rounds towards -Infinity | *VCVTM (floating-point)* on page F7-3352 |

a. See Table F5-17 on page F5-2566 for the interpretation of this bit.

Table F5-19 shows the immediate constants available in the `VMOV` (immediate) instruction.

**Table F5-19 Floating-point modified immediate constants**

| Data type | opc2 | opc4 | Constant [a] |
|---|---|---|---|
| F32 | abcd | efgh | aBbbbbbc defgh000 00000000 00000000 |
| F64 | abcd | efgh | aBbbbbbb bbcdefgh 00000000 00000000 00000000 00000000 00000000 00000000 |

a. In this column, `B` = NOT(`b`). The bit pattern represents the floating-point number $(-1)^S \times 2^{exp} \times$ mantissa, where S = `UInt(a)`, exp = `UInt(NOT(b):c:d)-3` and mantissa = `(16+UInt(e:f:g:h))/16`.

## F5.5.2 Operation of modified immediate constants, floating-point

The `VFPExpandImm()` pseudocode function describes the operation of an immediate constant in a floating-point instruction.

```
// VFPExpandImm()
// ==============

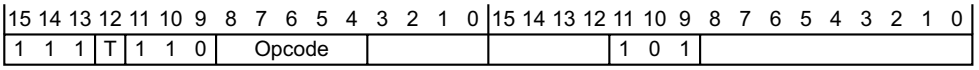
bits(N) VFPExpandImm(bits(8) imm8)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    return sign : exp : frac;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

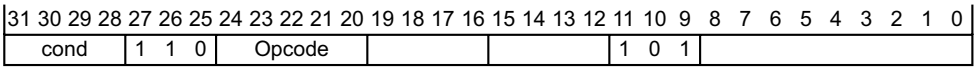
## F5.6 Advanced SIMD and floating-point register load/store instructions

These are instructions that transfer data to or from the registers in the SIMD and floating-point register file.

The T32 encoding of Advanced SIMD and floating-point register load or store instructions is:




The A32 encoding of Advanced SIMD and floating-point register load or store instructions is:




If `T` == 1 in the T32 encoding or `cond` == `0b1111` in the A32 encoding, the instruction is UNDEFINED.

Otherwise, the allocation of encodings in this space is shown in Table F5-20. Other encodings in this space are UNDEFINED.

These instructions are `LDC` and `STC` instructions for coprocessors 10 and 11.

**Table F5-20 Advanced SIMD and floating-point register load/store instructions**

| Opcode | Instruction | See |
|---|---|---|
| 0010x | - | *64-bit transfers accessing the SIMD and floating-point register file* on page F5-2573 |
| 01xx0<br>10x10 | Vector Store Multiple[a] | *VSTM, VSTMDB, VSTMIA* on page F7-3711[a] |
| 1xx00 | Vector Store Register | *VSTR* on page F7-3714 |
| 01xx1<br>10x11 | Vector Load Multiple[b] | *VLDM, VLDMDB, VLDMIA* on page F7-3430[b] |
| 1xx01 | Vector Load Register | *VLDR* on page F7-3433 |

a. Previous descriptions of the T32/A32 Advanced SIMD and floating-point encodings have included `VPUSH`. This is an alias of an encoding of `VSTM`, and its encoding is now described in *VSTM, VSTMDB, VSTMIA* on page F7-3711.

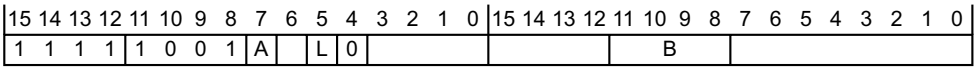
b. Previous descriptions of the T32/A32 Advanced SIMD and floating-point encodings have included `VPOP`. This is an alias of an encoding of `VLDM`, and its encoding is now described in *VLDM, VLDMDB, VLDMIA* on page F7-3430.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

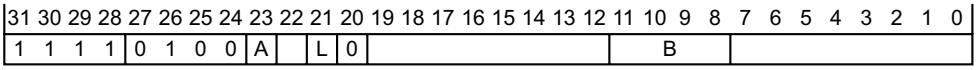
## F5.7 Advanced SIMD element or structure load/store instructions

These are instructions that transfer data to or from Advanced SIMD elements or structures in the SIMD and floating-point register file.

The T32 encoding of Advanced SIMD element load or store instructions is:




The A32 encoding of Advanced SIMD element load or store instructions is:




The allocation of encodings in this space is shown in:

- Table F5-21 if L == 0. These are the encodings for store instructions.
- Table F5-22 on page F5-2570 if L == 1. These are the encodings for load instructions.

Other encodings in this space are UNDEFINED.

The variable bits are in identical locations in the two encodings, after adjusting for the fact that the A32 encoding is held in memory as a single word and the T32 encoding is held as two consecutive halfwords.

The A32 instructions can only be executed unconditionally. The T32 instructions can be executed conditionally by using the `IT` instruction. For details see *IT* on page F6-2690.

**Table F5-21 Element and structure store instructions (L == 0)**

| A | B | Instruction | See |
|---|---|---|---|
| 0 | 0010<br>011x<br>1010 | Vector Store | *VST1 (multiple single elements)* on page F7-3690 |
| | 0011<br>100x | Vector Store | *VST2 (multiple 2-element structures)* on page F7-3696 |
| | 010x | Vector Store | *VST3 (multiple 3-element structures)* on page F7-3702 |
| | 000x | Vector Store | *VST4 (multiple 4-element structures)* on page F7-3708 |
| 1 | 0x00<br>1000 | Vector Store | *VST1 (single element from one lane)* on page F7-3687 |
| | 0x01<br>1001 | Vector Store | *VST2 (single 2-element structure from one lane)* on page F7-3693 |
| | 0x10<br>1010 | Vector Store | *VST3 (single 3-element structure from one lane)* on page F7-3699 |
| | 0x11<br>1011 | Vector Store | *VST4 (single 4-element structure from one lane)* on page F7-3705 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F5-22 Element and structure load instructions (L == 1)**

| A | B | Instruction | See |
|---|---|---|---|
| 0 | 0010<br>011x<br>1010 | Vector Load | *VLD1 (multiple single elements)* on page F7-3400 |
| | 0011<br>100x | Vector Load | *VLD2 (multiple 2-element structures)* on page F7-3409 |
| | 010x | Vector Load | *VLD3 (multiple 3-element structures)* on page F7-3418 |
| | 000x | Vector Load | *VLD4 (multiple 4-element structures)* on page F7-3427 |
| 1 | 0x00<br>1000 | Vector Load | *VLD1 (single element to one lane)* on page F7-3394 |
| | 1100 | Vector Load | *VLD1 (single element to all lanes)* on page F7-3397 |
| | 0x01<br>1001 | Vector Load | *VLD2 (single 2-element structure to one lane)* on page F7-3403 |
| | 1101 | Vector Load | *VLD2 (single 2-element structure to all lanes)* on page F7-3406 |
| | 0x10<br>1010 | Vector Load | *VLD3 (single 3-element structure to one lane)* on page F7-3412 |
| | 1110 | Vector Load | *VLD3 (single 3-element structure to all lanes)* on page F7-3415 |
| | 0x11<br>1011 | Vector Load | *VLD4 (single 4-element structure to one lane)* on page F7-3421 |
| | 1111 | Vector Load | *VLD4 (single 4-element structure to all lanes)* on page F7-3424 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F5.7.1 Advanced SIMD addressing mode

All the element and structure load/store instructions use this addressing mode. There is a choice of three formats:

`[<Rn>{:<align>}]`
: The address is contained in general-purpose register Rn.

  Rn is not updated by this instruction.

  Encoded as Rm = `0b1111`.

  If Rn is encoded as `0b1111`, the instruction is UNPREDICTABLE.

`[<Rn>{:<align>}]!`
: The address is contained in general-purpose register Rn.

  Rn is updated by this instruction: `Rn = Rn + transfer_size`

  Encoded as Rm = `0b1101`.

  `transfer_size` is the number of bytes transferred by the instruction. This means that, after the instruction is executed, Rn points to the address in memory immediately following the last address loaded from or stored to.

  If Rn is encoded as `0b1111`, the instruction is UNPREDICTABLE.

  This addressing mode can also be written as:

  `[<Rn>{:align}], #<transfer_size>`

  However, disassembly produces the `[<Rn>{:align}]!` form.

`[<Rn>{:<align>}], <Rm>`
: The address is contained in general-purpose register `<Rn>`.

  Rn is updated by this instruction: `Rn = Rn + Rm`

  Encoded as Rm = Rm. Rm must not be encoded as `0b1111` or `0b1101`, the PC or the SP.

  If Rn is encoded as `0b1111`, the instruction is UNPREDICTABLE.

In all cases, `<align>` specifies an alignment. Details are given in the individual instruction descriptions.

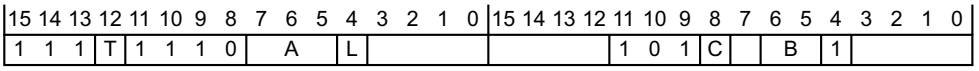
Previous versions of the manual used the `@` character for alignment. So, for example, the first format in this section was shown as `[<Rn>{@<align>}]`. Both `@` and `:` are supported. However, to ensure portability of code to assemblers that treat `@` as a comment character, `:` is preferred.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

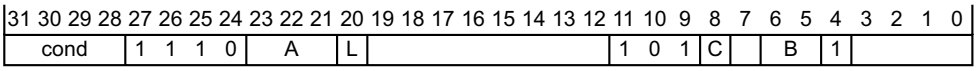
## F5.8 8, 16, and 32-bit transfers accessing the SIMD and floating-point register file

These are instructions that transfer data to or from registers in the SIMD and floating-point register file.

The T32 encoding of an Advanced SIMD and floating-point 8-bit, 16-bit, or 32-bit register data transfer instructions is:




The A32 encoding of an Advanced SIMD and floating-point 8-bit, 16-bit, or 32-bit register data transfer instructions is:




If `T` == 1 in the T32 encoding or `cond` == `0b1111` in the A32 encoding, the instruction is UNDEFINED.

Otherwise, the allocation of encodings in this space is shown in Table F5-23. Other encodings in this space are UNDEFINED.

These instructions are `MRC` and `MCR` instructions for coprocessors 10 and 11.

**Table F5-23 8-bit, 16-bit and 32-bit Advanced SIMD and floating-point data transfer instructions**

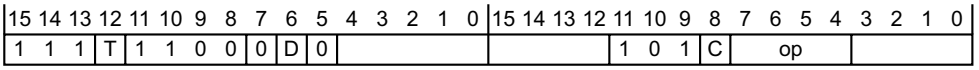
| L | C | A | B | Instruction | See |
|---|---|---|---|---|---|
| 0 | 0 | 000 | - | Vector Move | *VMOV (between general-purpose register and single-precision register)* on page F7-3485 |
| | | 111 | - | Move to Advanced SIMD and Floating-point System register from general-purpose register | *VMSR* on page F7-3497 |
| 0 | 1 | 0xx | - | Vector Move | *VMOV (general-purpose register to scalar)* on page F7-3483 |
| | | 1xx | 0x | Vector Duplicate | *VDUP (general-purpose register)* on page F7-3370 |
| 1 | 0 | 000 | - | Vector Move | *VMOV (between general-purpose register and single-precision register)* on page F7-3485 |
| | | 111 | - | Move to general-purpose register from Advanced SIMD and Floating-point System register | *VMRS* on page F7-3495 |
| | 1 | xxx | - | Vector Move | *VMOV (scalar to general-purpose register)* on page F7-3487 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

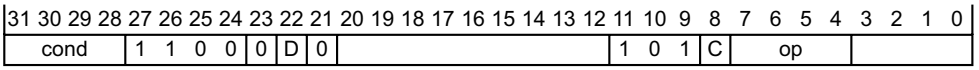
## F5.9 64-bit transfers accessing the SIMD and floating-point register file

These are instructions that transfer data to or from the registers in the SIMD and floating-point register file.

The T32 encoding of Advanced SIMD and floating-point 64-bit register data transfer instructions is:




The A32 encoding of Advanced SIMD and floating-point 64-bit register data transfer instructions is:




If `T` == 1 in the T32 encoding or `cond` == `0b1111` in the A32 encoding, the instruction is UNDEFINED.

Otherwise, the allocation of encodings in this space is shown in Table F5-24. Other encodings in this space are UNDEFINED.

These instructions are `MRRC` and `MCRR` instructions for coprocessors 10 and 11.

**Table F5-24 Advanced SIMD and floating-point 64-bit data transfer instructions**

| C | op | D | Instruction |
|---|---|---|---|
| 0 | 00x1 | 1 | *VMOV (between two general-purpose registers and two single-precision registers)* on page F7-3489 |
| 1 | 00x1 | 1 | *VMOV (between two general-purpose registers and a doubleword floating-point register)* on page F7-3474 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter F6
# T32 and A32 Base Instruction Set Instruction Descriptions

This chapter describes each instruction. It contains the following sections:


- *Alphabetical list of T32 and A32 base instruction set instructions* on page F6-2576.
- *Encoding and use of Banked register transfer instructions* on page F6-3202.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F6.1 Alphabetical list of T32 and A32 base instruction set instructions

This section lists every instruction in the T32 and A32 base instruction sets. For details of the format used see *Format of instruction descriptions* on page F2-2442.

This section is formatted so that each instruction description starts on a new page.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.1 ADC, ADCS (immediate)

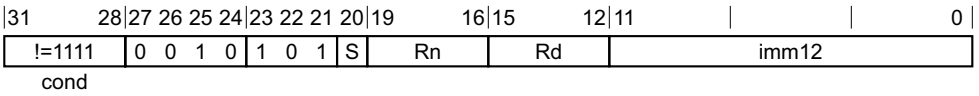
Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register.

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *ADC variant*

Applies when S = 0.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *ADCS variant*

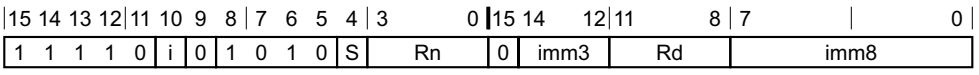
Applies when S = 1.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

#### T1




##### *ADC variant*

Applies when S = 0.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *ADCS variant*

Applies when S = 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 || n == 15 then UNPREDICTABLE;  // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], imm32, PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.2 ADC, ADCS (register)

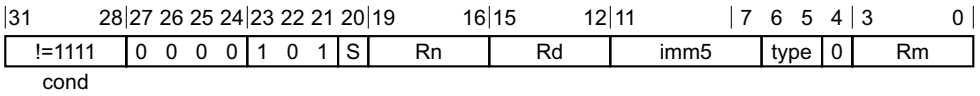
Add with Carry (register) adds a register value, the Carry flag value, and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *ADC, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *ADC, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *ADCS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *ADCS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

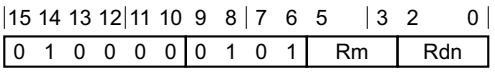
```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



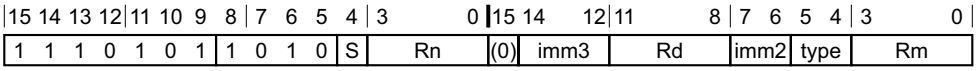

#### *T1 variant*

```
ADC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
ADCS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *ADC, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *ADC, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
ADC<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *ADCS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && imm2 = 00 && type = 11`.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *ADCS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
ADCS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rdn>` Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

`LSL` when `type = 00`

`LSR` when `type = 01`

`ASR` when `type = 10`

`ROR` when `type = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if `ADCS <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ADCS <Rd>, <Rn>` had been written.
- Inside an IT block, if `ADC<c> <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ADC<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the .W qualifier.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

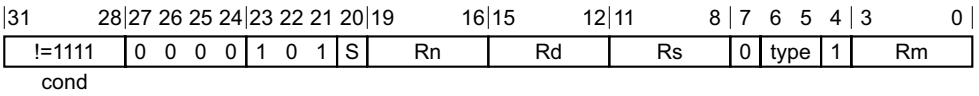
```
        ALUWritePC(result);
else
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.3 ADC, ADCS (register-shifted register)

Add with Carry (register-shifted register) adds a register value, the Carry flag value, and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

#### A1




##### *Flag setting variant*

Applies when `S = 1`.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Not flag setting variant*

Applies when `S = 0`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.4 ADD, ADDS (immediate)

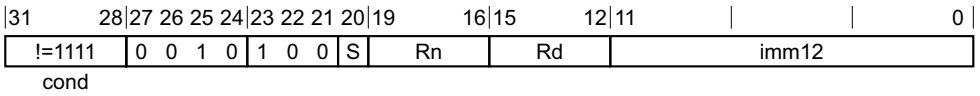
Add (immediate) adds an immediate value to a register value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The ADDS variant of the instruction performs an exception return without the use of the stack. ARM deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *ADD variant*

Applies when S = 0 && Rn != 11x1.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *ADDS variant*

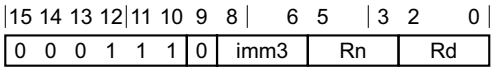
Applies when S = 1 && Rn != 1101.

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *Decode for all variants of this encoding*

```
if Rn == '1111' && S == '0' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

#### T1




##### *T1 variant*

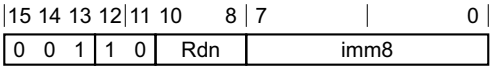
```
ADD<c>{<q>} <Rd>, <Rn>, #<imm3> // Inside IT block
ADDS{<q>} <Rd>, <Rn>, #<imm3> // Outside IT block
```

##### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2



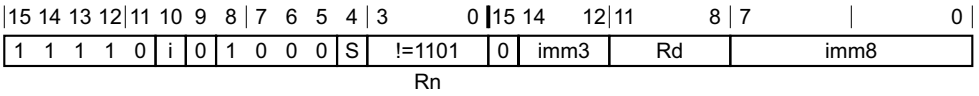

#### *T2 variant*

```
ADD<c>{<q>} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> can be represented in T1
ADD<c>{<q>} {<Rdn>,} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
ADDS{<q>} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> can be represented in T1
ADDS{<q>} {<Rdn>,} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1
```

#### *Decode for this encoding*

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

### T3




#### *ADD variant*

Applies when `S = 0`.

```
ADD<c>.W {<Rd>,} <Rn>, #<const> // Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or
T2
ADD{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *ADDS variant*

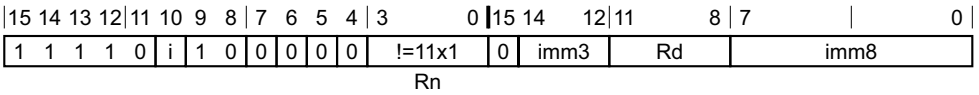
Applies when `S = 1 && Rd != 1111`.

```
ADDS.W {<Rd>,} <Rn>, #<const> // Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
ADDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *Decode for all variants of this encoding*

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T4




#### *T4 variant*

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
ADDW{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // <imm12> can be represented in T1, T2, or T3
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Decode for this encoding*

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;   // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rdn>` Is the general-purpose source and destination register, encoded in the "Rdn" field.

`<imm8>` Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. ARM deprecates use of this instruction.

For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD, ADDS (SP plus immediate). If the PC is used, see ADR.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD, ADDS (SP plus immediate).

`<imm3>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.

`<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T3: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `ADDW` syntax). Encoding T1 is preferred to encoding T2 if `<Rd>` is specified and encoding T2 is preferred to encoding T1 if `<Rd>` is omitted.

### Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        (result, nzcv) = AddWithCarry(R[n], imm32, '0');
        if d == 15 then          // Can only occur for A32 encoding
            if setflags then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                    ALUExceptionReturn(result);
                else
                    ALUWritePC(result);
            else
                R[d] = result;
                if setflags then
                    PSTATE.<N,Z,C,V> = nzcv;
    else
        if ConditionPassed() then
            EncodingSpecificOperations();
            (result, nzcv) = AddWithCarry(R[n], imm32, '0');
            R[d] = result;
            if setflags then
                PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.5 ADD, ADDS (register)

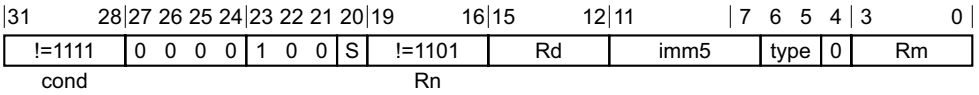
Add (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The ADDS variant of the instruction performs an exception return without the use of the stack. ARM deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *ADD, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *ADD, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *ADDS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *ADDS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

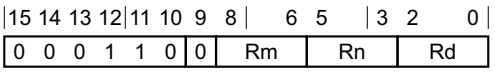
```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



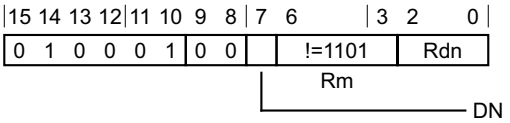

#### *T1 variant*

```
ADD<c>{<q>} <Rd>, <Rn>, <Rm> // Inside IT block
ADDS{<q>} {<Rd>,} <Rn>, <Rm> // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *T2 variant*

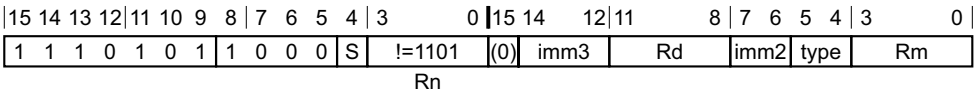
Applies when `!(DN == 1 && Rdn == 101)`.

```
ADD<c>{<q>} <Rdn>, <Rm> // Preferred syntax, Inside IT block
ADD{<c>}{<q>} {<Rdn>,} <Rdn>, <Rm>
```

#### *Decode for this encoding*

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = d;  m = UInt(Rm);  setflags = FALSE;  (shift_t, shift_n) = (SRType_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T3




#### *ADD, rotate right with extend variant*

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *ADD, shift or rotate by value variant*

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
ADD<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *ADDS, rotate right with extend variant*

Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && type == 11`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

### *ADDS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11) && Rd != 1111`.

```
ADDS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

### *Decode for all variants of this encoding*

```
if Rd == '1111' && S == '1' then SEE CMN (register);
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rdn>` Is the general-purpose source and destination register, encoded in the "DN:Rdn" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318. The assembler language allows `<Rdn>` to be specified once or twice in the assembler syntax. When used inside an IT block, and `<Rdn>` and `<Rm>` are in the range R0 to R7, `<Rdn>` must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when `<Rdn>` is specified once or twice.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. ARM deprecates use of this instruction.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. When used inside an IT block, `<Rd>` must be specified. When used outside an IT block, `<Rd>` is optional, and:

- If omitted, this register is the same as `<Rn>`.
- If present, encoding T1 is preferred to encoding T2.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used. If the SP is used, see ADD, ADDS (SP plus register).

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD, ADDS (SP plus register).


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rm>` For encoding A1 and T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T3: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

`LSL` when `type = 00`

`LSR` when `type = 01`

`ASR` when `type = 10`

`ROR` when `type = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

Inside an IT block, if `ADD<c> <Rd>, <Rn>, <Rd>` cannot be assembled using encoding T1, it is assembled using encoding T2 as though `ADD<c> <Rd>, <Rn>` had been written. To prevent this happening, use the `.W` qualifier.

## Operation for all encodings

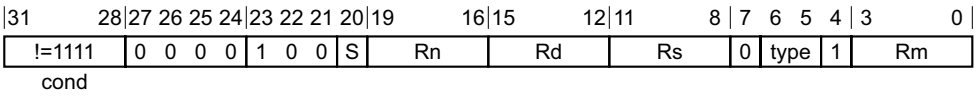
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.6 ADD, ADDS (register-shifted register)

Add (register-shifted register) adds a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

#### A1




##### *Flag setting variant*

Applies when `S = 1`.

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Not flag setting variant*

Applies when `S = 0`.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

#### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

#### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.7 ADD, ADDS (SP plus immediate)

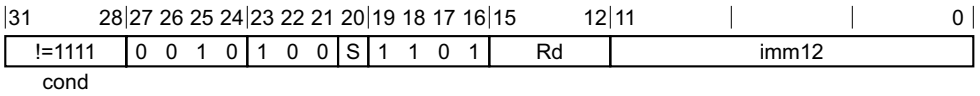
Add to SP (immediate) adds an immediate value to the SP value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The ADDS variant of the instruction performs an exception return without the use of the stack. ARM deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *ADD variant*

Applies when `S = 0`.

```
ADD{<c>}{<q>} {<Rd>,} SP, #<const>
```

##### *ADDS variant*

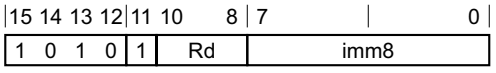
Applies when `S = 1`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, #<const>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

#### T1




##### *T1 variant*

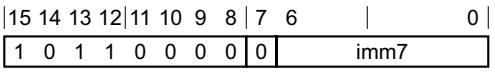
```
ADD{<c>}{<q>} <Rd>, SP, #<imm8>
```

##### *Decode for this encoding*

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm8:'00', 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2



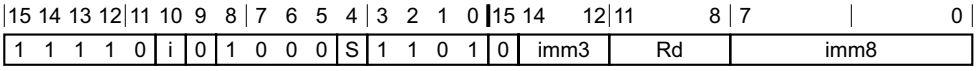

#### *T2 variant*

```
ADD{<c>}{<q>} {SP,} SP, #<imm7>
```

#### *Decode for this encoding*

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

### T3




#### *ADD variant*

Applies when `S = 0`.

```
ADD{<c>}.W {<Rd>,} SP, #<const> // <Rd>, <const> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} SP, #<const>
```

#### *ADDS variant*

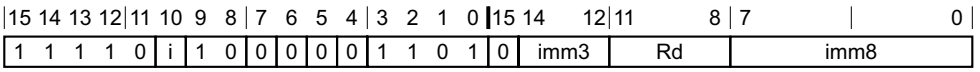
Applies when `S = 1 && Rd != 1111`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, #<const>
```

#### *Decode for all variants of this encoding*

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && !setflags then UNPREDICTABLE;
```

### T4




#### *T4 variant*

```
ADD{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
ADDW{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> can be represented in T1, T2, or T3
```

#### *Decode for this encoding*

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<imm7> Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.

For encoding T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

<imm8> Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T3: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(SP, imm32, '0');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.8 ADD, ADDS (SP plus register)

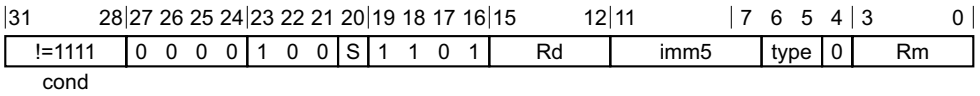
Add to SP (register) adds an optionally-shifted register value to the SP value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The ADDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *ADD, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
ADD{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

##### *ADD, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
ADD{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

##### *ADDS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

##### *ADDS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

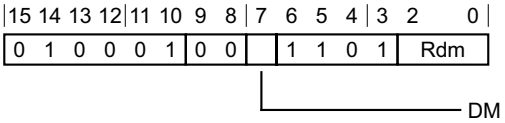
```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



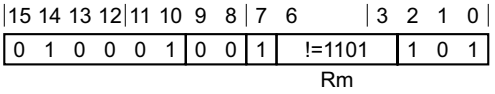

#### *T1 variant*

```
ADD{<c>}{<q>} {<Rdm>,} SP, <Rdm>
```

#### *Decode for this encoding*

```
d = UInt(DM:Rdm);  m = UInt(DM:Rdm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T2



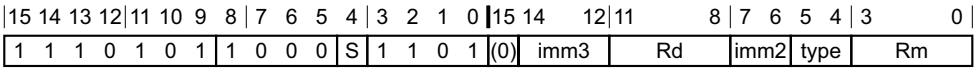

#### *T2 variant*

```
ADD{<c>}{<q>} {SP,} SP, <Rm>
```

#### *Decode for this encoding*

```
if Rm == '1101' then SEE encoding T1;
d = 13;  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T3




#### *ADD, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
ADD{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

#### *ADD, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
ADD{<c>}.W {<Rd>,} SP, <Rm> // <Rd>, <Rm> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

#### *ADDS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && Rd != 1111 && imm2 = 00 && type = 11`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *ADDS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11) && Rd != 1111`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

### *Decode for all variants of this encoding*

```
if Rd == '1111' && S == '1' then SEE CMN (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if (d == 15 && !setflags) || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rdm>` Is the general-purpose destination and second source register, encoded in the "Rdm" field. If omitted, this register is the SP. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

`<Rm>` For encoding A1 and T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(SP, shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

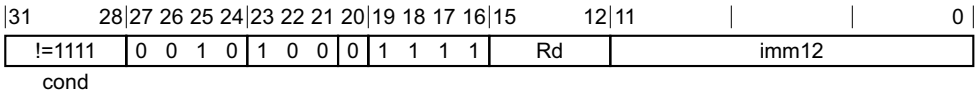
### F6.1.9 ADD (immediate, to PC)

Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. ARM recommends that, where possible, software avoids using this alias

This instruction is an alias of the ADR instruction. This means that:

- The encodings in this description are named to match the encodings of ADR.
- The description of ADR gives the operational pseudocode for this instruction.

#### A1




##### *A1 variant*

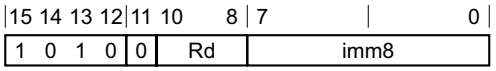
```
ADD{<c>}{<q>} <Rd>, PC, #<const>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is never the preferred disassembly.

#### T1




##### *T1 variant*

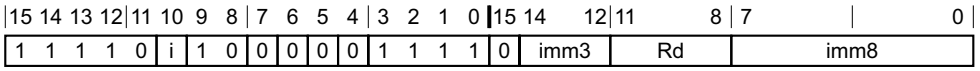
```
ADD{<c>}{<q>} <Rd>, PC, #<imm8>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is never the preferred disassembly.

#### T3




##### *T3 variant*

```
ADDW{<c>}{<q>} <Rd>, PC, #<imm12> // <Rd>, <imm12> can be represented in T1
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is never the preferred disassembly.

```
ADD{<c>}{<q>} <Rd>, PC, #<imm12>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is never the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T1 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<label>` For encoding A1 and A2: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. If the offset is zero or positive, encoding A1 is used, with `imm32` equal to the offset. If the offset is negative, encoding A2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions* on page F2-2456.

For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset. If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are 0-4095.

`<imm8>` Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.

`<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

`<const>` An immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

### Operation for all encodings

The description of ADR gives the operational pseudocode for this instruction.

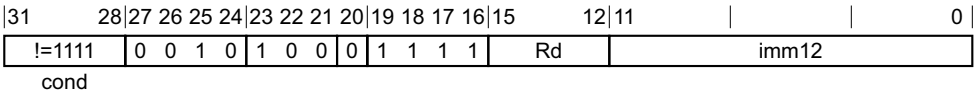

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.10 ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

This instruction is used by the aliases ADD (immediate, to PC) and SUB (immediate, from PC). See the *Alias conditions* on page F6-2605 table for details of when each alias is preferred.

### A1



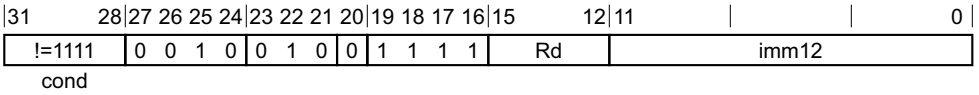

#### *A1 variant*

```
ADR{<c>}{<q>} <Rd>, <label>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  imm32 = A32ExpandImm(imm12);  add = TRUE;
```

### A2



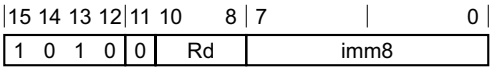

#### *A2 variant*

```
ADR{<c>}{<q>} <Rd>, <label>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  imm32 = A32ExpandImm(imm12);  add = FALSE;
```

### T1



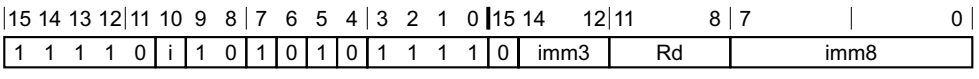

#### *T1 variant*

```
ADR{<c>}{<q>} <Rd>, <label>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

### T2





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

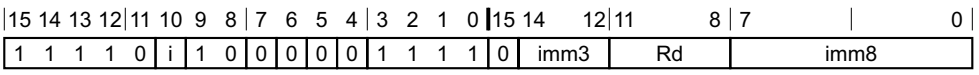
#### T2 variant

```
ADR{<c>}{<q>} <Rd>, <label>
```

#### Decode for this encoding

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = FALSE;
if d == 15 then UNPREDICTABLE;     // ARMv8-A removes UNPREDICTABLE for R13
```

### T3




#### T3 variant

```
ADR{<c>}.W <Rd>, <label> // <Rd>, <label> can be presented in T1
ADR{<c>}{<q>} <Rd>, <label>
```

#### Decode for this encoding

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = TRUE;
if d == 15 then UNPREDICTABLE;   // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

## Alias conditions

| Alias | of variant | is preferred when |
|---|---|---|
| ADD (immediate, to PC) | - | `Never` |
| SUB (immediate, from PC) | T2 | `i:imm3:imm8 = '000000000000'` |
| SUB (immediate, from PC) | A2 | `imm12==000000000000` |

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> For encoding A1 and A2: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<label> For encoding A1 and A2: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the ADR instruction to this label. If the offset is zero or positive, encoding A1 is used, with `imm32` equal to the offset. If the offset is negative, encoding A2 is used, with `imm32` equal to the size of the offset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

That is, the use of encoding A2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions* on page F2-2456.

For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset. If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are 0-4095.

The instruction aliases permit the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if d == 15 then          // Can only occur for A32 encodings
        ALUWritePC(result);
    else
        R[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.11 AND, ANDS (immediate)

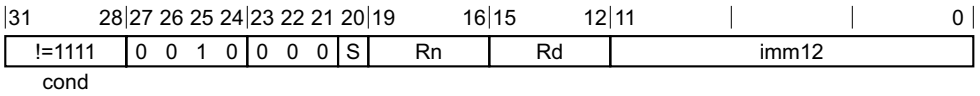
Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *AND variant*

Applies when S = 0.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *ANDS variant*

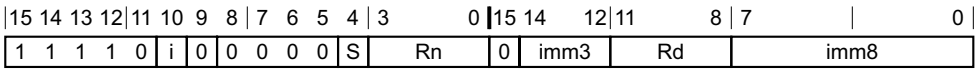
Applies when S = 1.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

#### T1




##### *AND variant*

Applies when S = 0.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *ANDS variant*

Applies when S = 1 && Rd != 1111.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE TST (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.12 AND, ANDS (register)

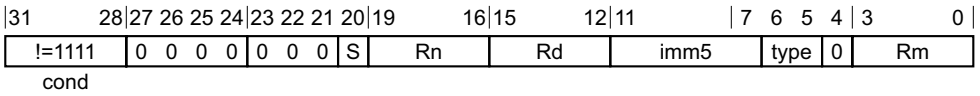
Bitwise AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *AND, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *AND, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *ANDS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *ANDS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

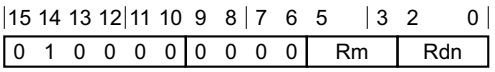
```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



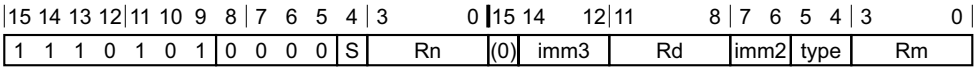

#### *T1 variant*

```
AND<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
ANDS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *AND, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *AND, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
AND<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *ANDS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && Rd != 1111 && imm2 = 00 && type = 11`.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *ANDS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11) && Rd != 1111`.

```
ANDS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
if Rd == '1111' && S == '1' then SEE TST (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rdn>` Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

`LSL` when `type = 00`

`LSR` when `type = 01`

`ASR` when `type = 10`

`ROR` when `type = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if `ANDS <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `ANDS <Rd>, <Rn>` had been written.
- Inside an IT block, if `AND<c> <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `AND<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the `.W` qualifier.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

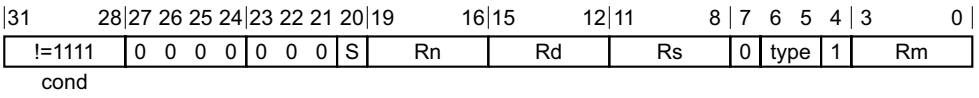
```
        ALUWritePC(result);
else
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.13 AND, ANDS (register-shifted register)

Bitwise AND (register-shifted register) performs a bitwise AND of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

#### A1




##### *Flag setting variant*

Applies when `S = 1`.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Not flag setting variant*

Applies when `S = 0`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

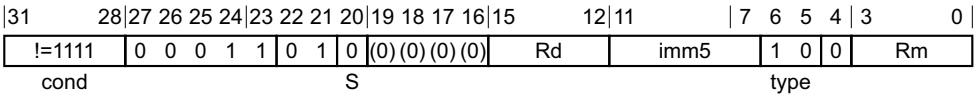
### F6.1.14 ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

#### A1




##### *MOV, shift or rotate by value variant*

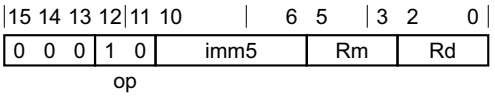
```
ASR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

#### T2




##### *T2 variant*

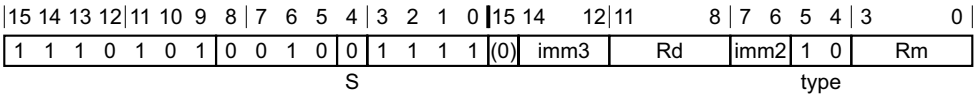
```
ASR<c>{<q>} {<Rd>,} <Rm>, #<imm> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is the preferred disassembly when `InITBlock()`.

#### T3




##### *MOV, shift or rotate by value variant*

```
ASR<c>.W {<Rd>,} <Rm>, #<imm> // Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

```
ASR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

### Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

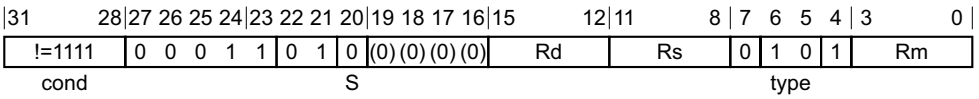
### F6.1.15 ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

#### A1




##### *Not flag setting variant*

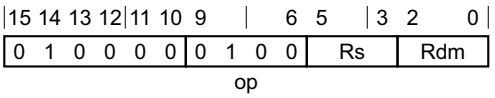
```
ASR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

#### T1




##### *Arithmetic shift right variant*

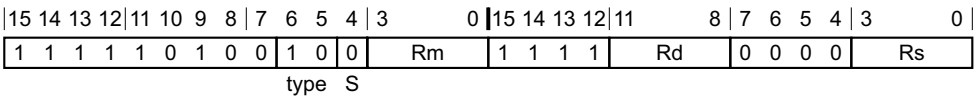
```
ASR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

#### T2




##### *Not flag setting variant*

```
ASR<c>.W {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

```
ASR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Rdm>` | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| `<Rd>` | Is the general-purpose destination register, encoded in the "Rd" field. |
| `<Rm>` | Is the first general-purpose source register, encoded in the "Rm" field. |
| `<Rs>` | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.16 ASRS (immediate)

Arithmetic Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

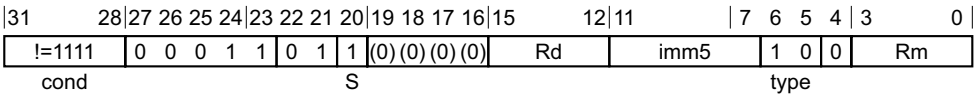
The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

#### A1




##### *MOVS, shift or rotate by value variant*

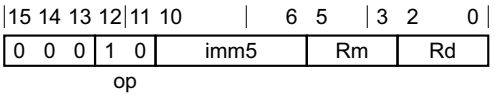
```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

#### T2




##### *T2 variant*

```
ASRS{<q>} {<Rd>,} <Rm>, #<imm> // Outside IT block
```

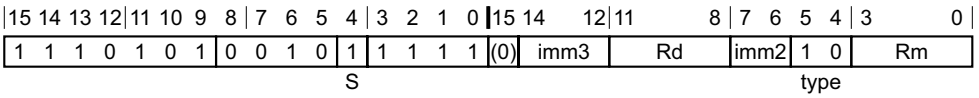
is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is the preferred disassembly when `!InITBlock()`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T3




#### *MOVS, shift or rotate by value variant*

```
ASRS.W {<Rd>,} <Rm>, #<imm> // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

## Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

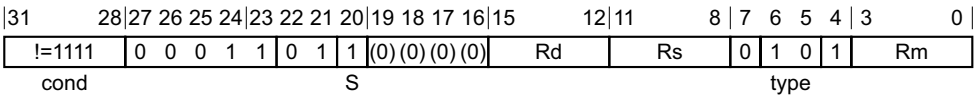
## F6.1.17 ASRS (register)

Arithmetic Shift Right, setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### A1




#### *Flag setting variant*

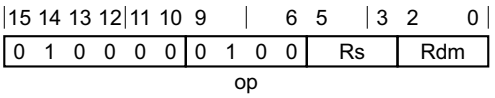
```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

### T1




#### *Arithmetic shift right variant*

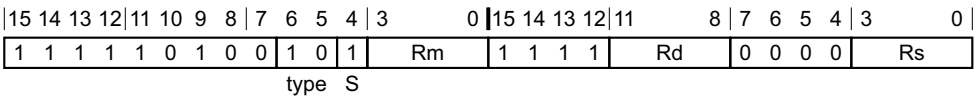
```
ASRS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

### T2




#### *Flag setting variant*

```
ASRS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Rdm>` | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| `<Rd>` | Is the general-purpose destination register, encoded in the "Rd" field. |
| `<Rm>` | Is the first general-purpose source register, encoded in the "Rm" field. |
| `<Rs>` | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

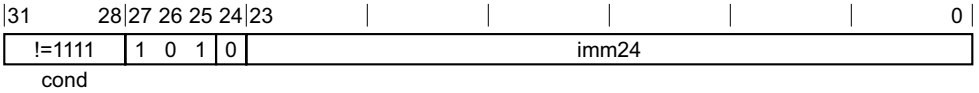
The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.18 B

Branch causes a branch to a target address.

### A1



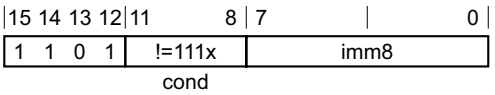

#### *A1 variant*

```
B{<c>}{<q>} <label>
```

#### *Decode for this encoding*

```
imm32 = SignExtend(imm24:'00', 32);
```

### T1



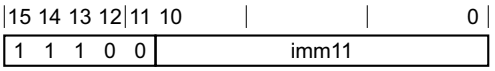

#### *T1 variant*

```
B<c>{<q>} <label> // Not permitted in IT block
```

#### *Decode for this encoding*

```
if cond == '1110' then SEE UDF;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

### T2



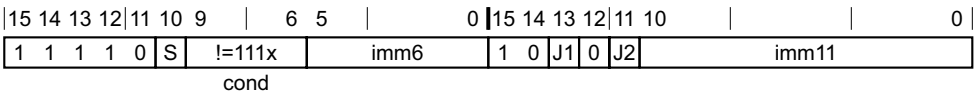

#### *T2 variant*

```
B{<c>}{<q>} <label> // Outside or last in IT block
```

#### *Decode for this encoding*

```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T3





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

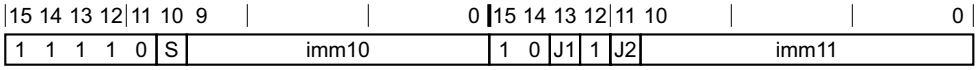
### *T3 variant*

```
B<c>.W <label> // Not permitted in IT block, and <label> can be represented in T1
B<c>{<q>} <label> // Not permitted in IT block
```

### *Decode for this encoding*

```
if cond<3:1> == '111' then SEE "Related encodings";
imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

## T4




### *T4 variant*

```
B{<c>}.W <label> // <label> can be represented in T2
B{<c>}{<q>} <label>
```

### *Decode for this encoding*

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: *Branches and miscellaneous control* on page F3-2485.

## Assembler symbols

<c> For encoding A1, T2 and T4: see *Standard assembler syntax fields* on page F2-2446.

For encoding T1: see *Standard assembler syntax fields* on page F2-2446. Must not be AL or omitted.

For encoding T3: see *Standard assembler syntax fields* on page F2-2446. <c> must not be AL or omitted.

<q> See *Standard assembler syntax fields* on page F2-2446.

<label> For encoding A1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are multiples of 4 in the range –33554432 to 33554428.

For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –256 to 254.

For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –2048 to 2046.

For encoding T3: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –1048576 to 1048574.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T4: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214.

### Operation for all encodings

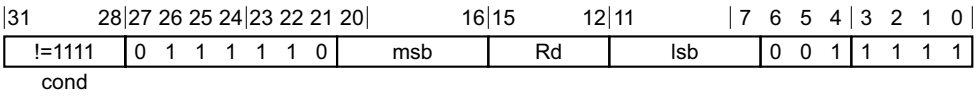
```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.19 BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

#### A1



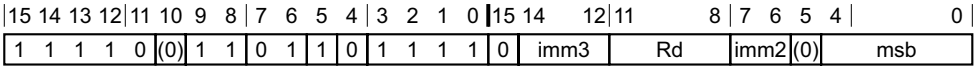

##### *A1 variant*

```
BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>
```

##### *Decode for this encoding*

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
```

#### T1




##### *T1 variant*

```
BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>
```

##### *Decode for this encoding*

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE;  // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *BFC* on page K1-5433.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <lsb> | For encoding A1: is the least significant bit to be cleared, in the range 0 to 31, encoded in the "lsb" field.<br>For encoding T1: is the least significant bit that is to be cleared, in the range 0 to 31, encoded in the "imm3:imm2" field. |
| <width> | Is the number of bits to be cleared, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

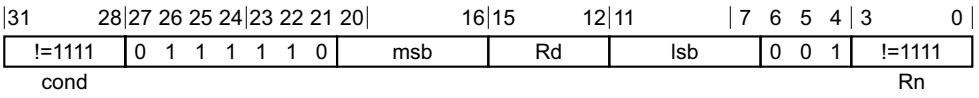
```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = Replicate('0', msbit-lsbit+1);
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.20 BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

### A1



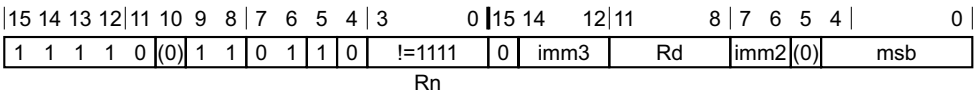

#### *A1 variant*

```
BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE BFC;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE BFC;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *BFI* on page K1-5433.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <lsb> | For encoding A1: is the least significant destination bit, in the range 0 to 31, encoded in the "lsb" field.<br>For encoding T1: is the least significant destination bit, in the range 0 to 31, encoded in the "imm3:imm2" field. |
| <width> | Is the number of bits to be copied, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.21 BIC, BICS (immediate)

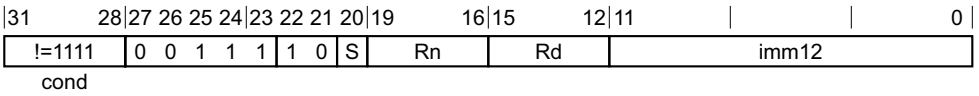
Bitwise Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *BIC variant*

Applies when S = 0.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *BICS variant*

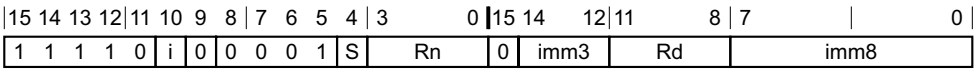
Applies when S = 1.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

#### T1




##### *BIC variant*

Applies when S = 0.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *BICS variant*

Applies when S = 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 || n == 15 then UNPREDICTABLE;  // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND NOT(imm32);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.22 BIC, BICS (register)

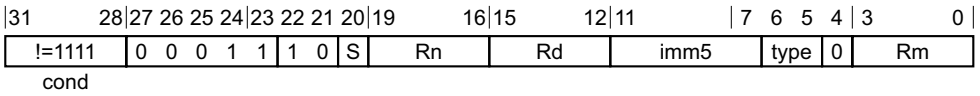
Bitwise Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1




#### *BIC, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *BIC, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *BICS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *BICS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

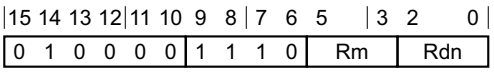
```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



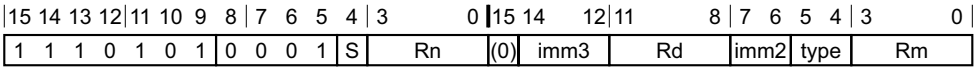

#### *T1 variant*

```
BIC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
BICS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *BIC, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *BIC, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
BIC<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *BICS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && imm2 = 00 && type = 11`.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *BICS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
BICS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| LSL | when type = 00 |
| LSR | when type = 01 |
| ASR | when type = 10 |
| ROR | when type = 11 |

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

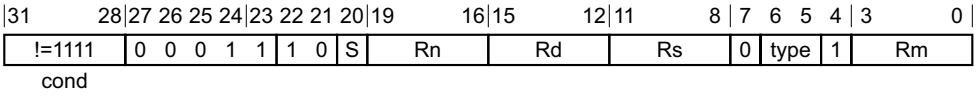
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.23 BIC, BICS (register-shifted register)

Bitwise Bit Clear (register-shifted register) performs a bitwise AND of a register value and the complement of a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

#### A1




##### *Flag setting variant*

Applies when `S = 1`.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Not flag setting variant*

Applies when `S = 0`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

#### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

#### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

- `LSL` when `type = 00`
- `LSR` when `type = 01`
- `ASR` when `type = 10`
- `ROR` when `type = 11`

`<Rs>` Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

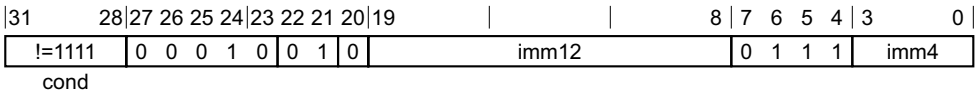

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.24 BKPT

Breakpoint causes a Software Breakpoint Instruction exception.

Breakpoint is always unconditional, even when inside an IT block.

### A1



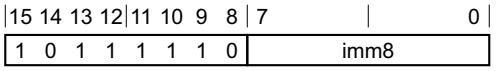

#### *A1 variant*

```
BKPT{<q>} {#}<imm>
```

#### *Decode for this encoding*

```
imm16 = imm12:imm4;
if cond != '1110' then UNPREDICTABLE;  // BKPT must be encoded with AL condition
```

### T1




#### *T1 variant*

```
BKPT{<q>} {#}<imm>
```

#### *Decode for this encoding*

```
imm16 = ZeroExtend(imm8, 16);
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *BKPT* on page K1-5433.

## Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446. An BKPT instruction must be unconditional.

<imm> For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. The PE:

- Records this value in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
- Ignores this value otherwise.

For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE:

- Records this value in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
- Ignores this value otherwise.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
EncodingSpecificOperations();
AArch32.SoftwareBreakpoint(imm16);
```

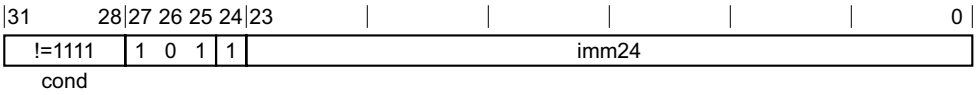

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.25 BL, BLX (immediate)

Branch with Link calls a subroutine at a PC-relative address, and setting LR to the return address.

Branch with Link and Exchange Instruction Sets (immediate) calls a subroutine at a PC-relative address, setting LR to the return address, and changes the instruction set from A32 to T32, or from T32 to A32.

### A1



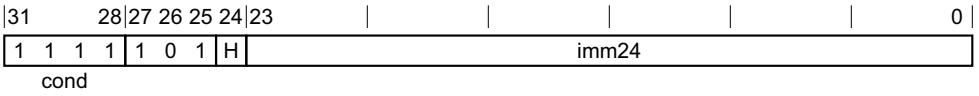

#### *A1 variant*

```
BL{<c>}{<q>} <label>
```

#### *Decode for this encoding*

```
imm32 = SignExtend(imm24:'00', 32);  targetInstrSet = InstrSet_A32;
```

### A2



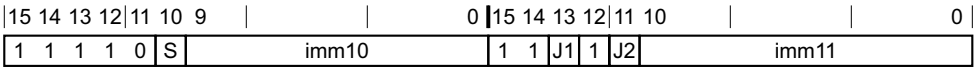

#### *A2 variant*

```
BLX{<c>}{<q>} <label>
```

#### *Decode for this encoding*

```
imm32 = SignExtend(imm24:H:'0', 32);  targetInstrSet = InstrSet_T32;
```

### T1



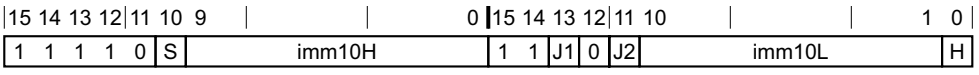

#### *T1 variant*

```
BL{<c>}{<q>} <label>
```

#### *Decode for this encoding*

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
targetInstrSet = InstrSet_T32;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T2





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *T2 variant*

```
BLX{<c>}{<q>} <label>
```

### *Decode for this encoding*

```
if H == '1' then UNDEFINED;
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10H:imm10L:'00', 32);
targetInstrSet = InstrSet_A32;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> For encoding A1, T1 and T2: see *Standard assembler syntax fields* on page F2-2446.

For encoding A2: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted.

<q> See *Standard assembler syntax fields* on page F2-2446.

<label> For encoding A1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are multiples of 4 in the range –33554432 to 33554428.

For encoding A2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range –33554432 to 33554430.

For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214.

For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the Align(PC, 4) value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are multiples of 4 in the range –16777216 to 16777212.

## Operation for all encodings

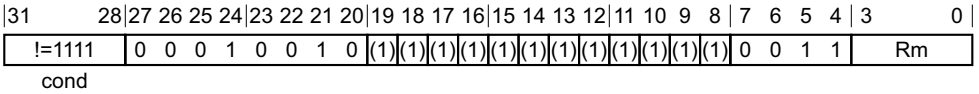
```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_A32 then
        LR = PC - 4;
    else
        LR = PC<31:1> : '1';
    if targetInstrSet == InstrSet_A32 then
        targetAddress = Align(PC,4) + imm32;
    else
        targetAddress = PC + imm32;
    SelectInstrSet(targetInstrSet);
    BranchWritePC(targetAddress);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.26 BLX (register)

Branch with Link and Exchange (register) calls a subroutine at an address specified in the register, and if necessary changes to the instruction set indicated by bit[0] of the register value. If the value in bit[0] is 0, the instruction set after the branch will be A32. If the value in bit[0] is 1, the instruction set after the branch will be T32.

### A1



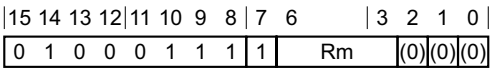

#### *A1 variant*

```
BLX{<c>}{<q>} <Rm>
```

#### *Decode for this encoding*

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
BLX{<c>}{<q>} <Rm>
```

#### *Decode for this encoding*

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rm> Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The SP can be used, but this is deprecated.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    target = R[m];
    if CurrentInstrSet() == InstrSet_A32 then
        next_instr_addr = PC - 4;
        LR = next_instr_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

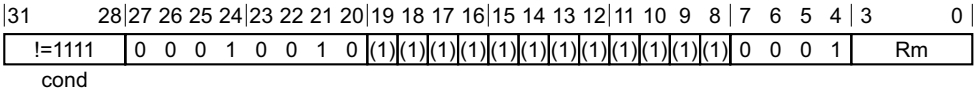
```
else
    next_instr_addr = PC - 2;
    LR = next_instr_addr<31:1> : '1';
BXWritePC(target);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.27 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register.

### A1



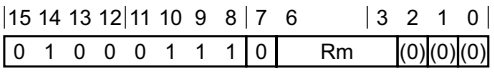

#### *A1 variant*

```
BX{<c>}{<q>} <Rm>
```

#### *Decode for this encoding*

```
m = UInt(Rm);
```

### T1




#### *T1 variant*

```
BX{<c>}{<q>} <Rm>
```

#### *Decode for this encoding*

```
m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rm> For encoding A1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The SP can be used, but this is deprecated. The PC can be used.

For encoding T1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The SP can be used, but this is deprecated. The PC can be used.

**Note**

If <Rm> is the PC at a non word-aligned address, it results in UNPREDICTABLE behavior because the address passed to the `BXWritePC()` pseudocode function has bits<1:0> = '10'.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m]);
```

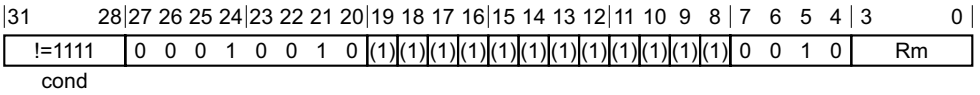

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.28 BXJ

Branch and Exchange, previously Branch and Exchange Jazelle.

In ARMv8, BXJ behaves as a BX instruction, see BX. This means it causes a branch to an address and instruction set specified by a register.

### A1



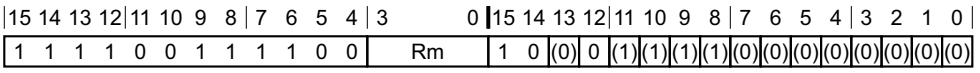

#### *A1 variant*

```
BXJ{<c>}{<q>} <Rm>
```

#### *Decode for this encoding*

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
BXJ{<c>}{<q>} <Rm>
```

#### *Decode for this encoding*

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;  // ARMv8-A removes UNPREDICTABLE for R13
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rm> Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

## Operation for all encodings

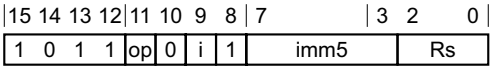
```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.29 CBNZ, CBZ

Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

### T1




#### *CBNZ variant*

Applies when `op = 1`.

```
CBNZ{<q>} <Rn>, <label>
```

#### *CBZ variant*

Applies when `op = 0`.

```
CBZ{<q>} <Rn>, <label>
```

#### *Decode for all variants of this encoding*

```
n = UInt(Rn);  imm32 = ZeroExtend(i:imm5:'0', 32);  nonzero = (op == '1');
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Rn>` | Is the general-purpose register to be tested, encoded in the "Rn" field. |
| `<label>` | Is the program label to be conditionally branched to. Its offset from the PC, a multiple of 2 and in the range 0 to 126, is encoded as "i:imm5" times 4. |

### Operation

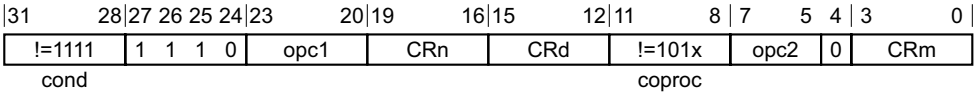
```
EncodingSpecificOperations();
if nonzero != IsZero(R[n]) then
    BranchWritePC(PC + imm32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.30 CDP, CDP2

Coprocessor Data Processing is a generic coprocessor instruction. The coproc field identifies the target conceptual coprocessor. This must be one of CP10, CP11, CP14, or CP15, and for these CP values, this manual defines the opc1, opc2, CRd, CRn, and CRm field values that are valid CDP and CDP2 instructions. Other encodings are UNDEFINED. For more information see *Conceptual coprocessor support* on page E1-2353 and *General behavior of System registers* on page G4-4118.

### A1



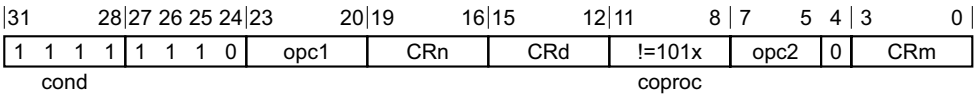

#### *A1 variant*

```
CDP{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}
```

#### *Decode for this encoding*

```
if coproc == '101x' then SEE "Floating-point instructions";
cp = UInt(coproc);
```

### A2



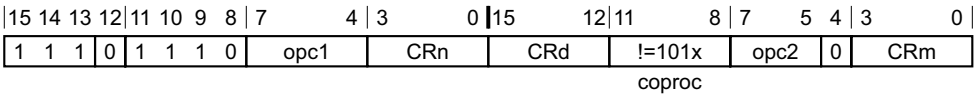

#### *A2 variant*

```
CDP2{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}
```

#### *Decode for this encoding*

```
if coproc == '101x' then SEE "Floating-point instructions";
cp = UInt(coproc);
```

### T1




#### *T1 variant*

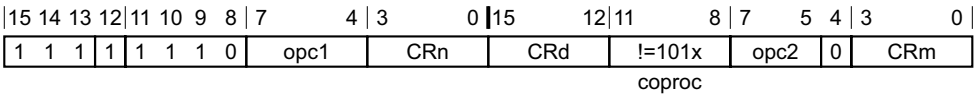
```
CDP{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}
```

#### *Decode for this encoding*

```
if coproc == '101x' then SEE "Floating-point instructions";
cp = UInt(coproc);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *T2 variant*

```
CDP2{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}
```

#### *Decode for this encoding*

```
if coproc == '101x' then SEE "Floating-point instructions";
cp = UInt(coproc);
```

## Notes for all encodings

Floating-point instructions: Floating-point data-processing instructions.

## Assembler symbols

| | |
|---|---|
| <c> | For encoding A1, T1 and T2: see *Standard assembler syntax fields* on page F2-2446.<br>For encoding A2: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| <opc1> | Is a coprocessor-specific opcode, in the range 0 to 15, encoded in the "opc1" field. |
| <CRd> | Is the destination coprocessor register, encoded in the "CRd" field. |
| <CRn> | Is the coprocessor register that contains the first operand, encoded in the "CRn" field. |
| <CRm> | Is the coprocessor register that contains the second operand, encoded in the "CRm" field. |
| <opc2> | Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field. |

## Operation for all encodings

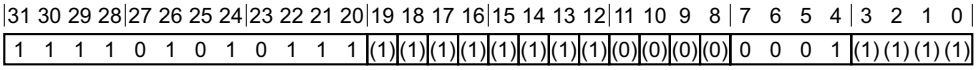
```
if ConditionPassed() then
    EncodingSpecificOperations();
    Coproc_CheckInstr(cp, ThisInstr());
    Coproc_InternalOperation(cp, ThisInstr());
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.31 CLREX

Clear-Exclusive clears the local monitor of the executing PE.

### A1



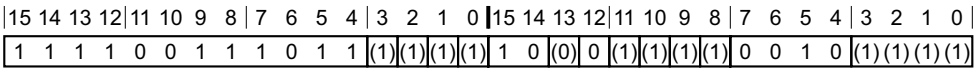

#### *A1 variant*

```
CLREX{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T1




#### *T1 variant*

```
CLREX{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-2446. Must be AL or omitted.
For encoding T1: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

### Operation for all encodings

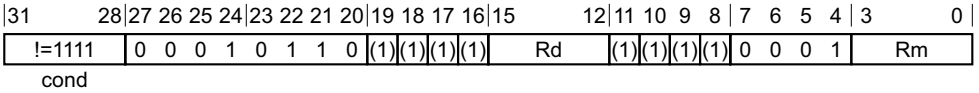
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.32 CLZ

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

### A1



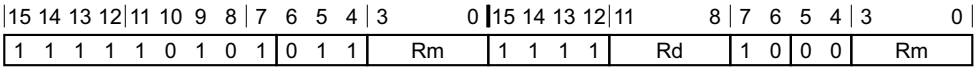

#### *A1 variant*

```
CLZ{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
CLZ{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *CLZ* on page K1-5434.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | For encoding A1: is the general-purpose source register, encoded in the "Rm" field.<br>For encoding T1: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice. |

### Operation for all encodings

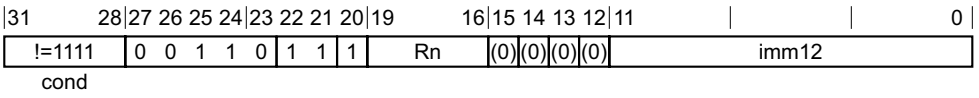
```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = CountLeadingZeroBits(R[m]);
    R[d] = result<31:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.33 CMN (immediate)

Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

#### A1



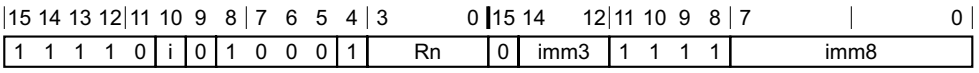

##### A1 variant

```
CMN{<c>}{<q>} <Rn>, #<const>
```

##### Decode for this encoding

```
n = UInt(Rn);  imm32 = A32ExpandImm(imm12);
```

#### T1




##### T1 variant

```
CMN{<c>}{<q>} <Rn>, #<const>
```

##### Decode for this encoding

```
n = UInt(Rn);  imm32 = T32ExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

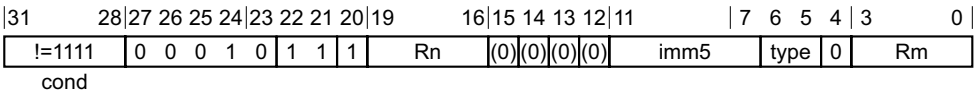
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], imm32, '0');
    PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.34 CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

### A1




#### *Rotate right with extend variant*

Applies when `imm5 = 00000 && type = 11`.

```
CMN{<c>}{<q>} <Rn>, <Rm>, RRX
```

#### *Shift or rotate by value variant*

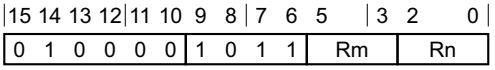
Applies when `!(imm5 == 00000 && type == 11)`.

```
CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

### T1



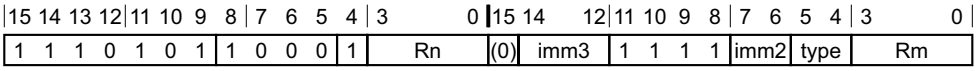

#### *T1 variant*

```
CMN{<c>}{<q>} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *Rotate right with extend variant*

Applies when `imm3 = 000 && imm2 = 00 && type = 11`.

```
CMN{<c>}{<q>} <Rn>, <Rm>, RRX
```

#### *Shift or rotate by value variant*

Applies when `!(imm3 == 000 && imm2 == 00 && type == 11)`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
CMN{<c>}.W <Rn>, <Rm> // <Rn>, <Rm> can be represented in T1
CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

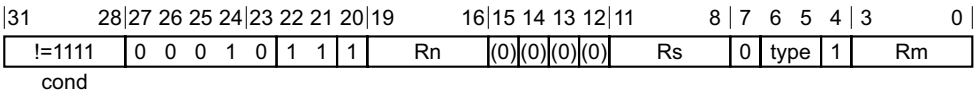
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.35 CMN (register-shifted register)

Compare Negative (register-shifted register) adds a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

### A1




#### *A1 variant*

```
CMN{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

#### *Decode for this encoding*

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<type> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| LSL | when type = 00 |
| LSR | when type = 01 |
| ASR | when type = 10 |
| ROR | when type = 11 |

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

### Operation

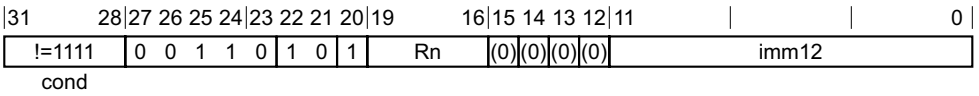
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.36 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

### A1



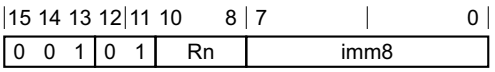

#### A1 variant

```
CMP{<c>}{<q>} <Rn>, #<const>
```

#### Decode for this encoding

```
n = UInt(Rn);  imm32 = A32ExpandImm(imm12);
```

### T1



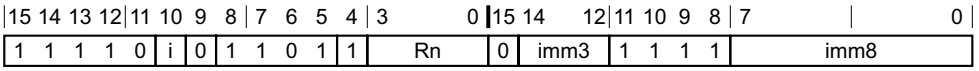

#### T1 variant

```
CMP{<c>}{<q>} <Rn>, #<imm8>
```

#### Decode for this encoding

```
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
```

### T2




#### T2 variant

```
CMP{<c>}.W <Rn>, #<const> // <Rd>, <const> can be represented in T1
CMP{<c>}{<q>} <Rn>, #<const>
```

#### Decode for this encoding

```
n = UInt(Rn);  imm32 = T32ExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is a general-purpose source register, encoded in the "Rn" field.

For encoding T2: is the general-purpose source register, encoded in the "Rn" field.

<imm8> Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

## Operation for all encodings

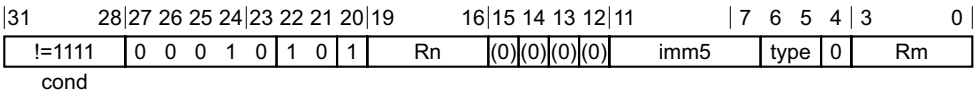
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), '1');
    PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.37 CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

### A1




#### *Rotate right with extend variant*

Applies when `imm5 = 00000 && type = 11`.

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

#### *Shift or rotate by value variant*

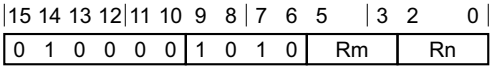
Applies when `!(imm5 == 00000 && type == 11)`.

```
CMP{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

### T1



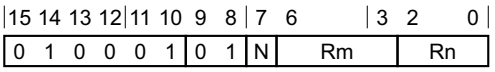

#### *T1 variant*

```
CMP{<c>}{<q>} <Rn>, <Rm> // <Rn> and <Rm> both from R0-R7
```

#### *Decode for this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *T2 variant*

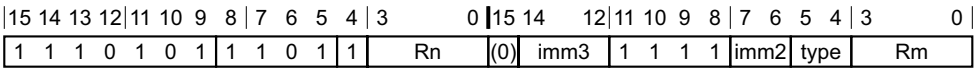
```
CMP{<c>}{<q>} <Rn>, <Rm> // <Rn> and <Rm> not both from R0-R7
```

#### *Decode for this encoding*

```
n = UInt(N:Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T3




#### *Rotate right with extend variant*

Applies when `imm3 = 000 && imm2 = 00 && type = 11`.

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

#### *Shift or rotate by value variant*

Applies when `!(imm3 == 000 && imm2 == 00 && type == 11)`.

```
CMP{<c>}.W <Rn>, <Rm> // <Rn>, <Rm> can be represented in T1 or T2
CMP{<c>}{<q>} <Rn>, <Rm>, <shift> #<amount>
```

#### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *CMP (register)* on page K1-5434.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1 and T3: is the first general-purpose source register, encoded in the "Rn" field.

For encoding T2: is the first general-purpose source register, encoded in the "N:Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1, T2 and T3: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

- `LSL` when `type = 00`
- `LSR` when `type = 01`
- `ASR` when `type = 10`
- `ROR` when `type = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

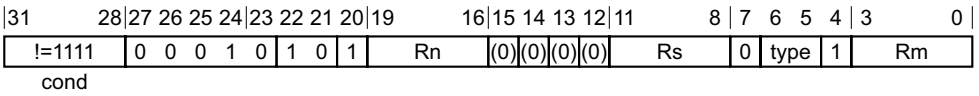
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.38 CMP (register-shifted register)

Compare (register-shifted register) subtracts a register-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

### A1




#### *A1 variant*

```
CMP{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

#### *Decode for this encoding*

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<type> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| LSL | when type = 00 |
| LSR | when type = 01 |
| ASR | when type = 10 |
| ROR | when type = 11 |

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

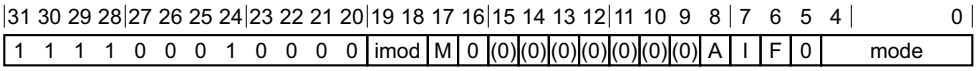
## F6.1.39 CPS, CPSID, CPSIE

Change PE State changes one or more of the PSTATE.{A, I, F} interrupt mask bits and, optionally, the PSTATE.M mode field, without changing any other PSTATE bits.

CPS is treated as NOP if executed in User mode unless it is defined as being CONSTRAINED UNPREDICTABLE elsewhere in this section.

The PE checks whether the value being written to PSTATE.M is legal. See *Illegal changes to PSTATE.M* on page G1-3771.

### A1




#### *CPS variant*

Applies when imod = 00 && M = 1.

```
CPS{<q>} #<mode> // Cannot be conditional
```

#### *CPSID variant*

Applies when imod = 11 && M = 0.

```
CPSID{<q>} <iflags> // Cannot be conditional
```

#### *CPSID variant*

Applies when imod = 11 && M = 1.

```
CPSID{<q>} <iflags> , #<mode> // Cannot be conditional
```

#### *CPSIE variant*

Applies when imod = 10 && M = 0.

```
CPSIE{<q>} <iflags> // Cannot be conditional
```

#### *CPSIE variant*

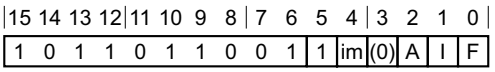
Applies when imod = 10 && M = 1.

```
CPSIE{<q>} <iflags> , #<mode> // Cannot be conditional
```

#### *Decode for all variants of this encoding*

```
if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if (imod == '00' && M == '0') || imod == '01' then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### CPSID variant

Applies when im = 1.

```
CPSID{<q>} <iflags> // Not permitted in IT block
```

#### CPSIE variant

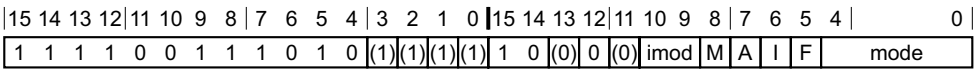
Applies when im = 0.

```
CPSIE{<q>} <iflags> // Not permitted in IT block
```

#### Decode for all variants of this encoding

```
if A:I:F == '000' then UNPREDICTABLE;
enable = (im == '0');  disable = (im == '1');  changemode = FALSE;
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if InITBlock() then UNPREDICTABLE;
```

### T2




#### CPS variant

Applies when imod = 00 && M = 1.

```
CPS{<q>} #<mode> // Not permitted in IT block
```

#### CPSID variant

Applies when imod = 11 && M = 0.

```
CPSID.W <iflags> // Not permitted in IT block
```

#### CPSID variant

Applies when imod = 11 && M = 1.

```
CPSID{<q>} <iflags>, #<mode> // Not permitted in IT block
```

#### CPSIE variant

Applies when imod = 10 && M = 0.

```
CPSIE.W <iflags> // Not permitted in IT block
```

#### CPSIE variant

Applies when imod = 10 && M = 1.

```
CPSIE{<q>} <iflags>, #<mode> // Not permitted in IT block
```

#### Decode for all variants of this encoding

```
if imod == '00' && M == '0' then SEE "Hint instructions";
if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if imod == '01' || InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

Hint instructions: In encoding T2, if the `imod` field is `00` and the `M` bit is `0`, a hint instruction is encoded. To determine which hint instruction, see *Branches and miscellaneous control* on page F3-2485.

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *CPS (T32)* on page K1-5479 and .

### Assembler symbols

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<iflags>` Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:

- `a` Sets the A bit in the instruction, causing the specified effect on PSTATE.A, the asynchronous abort mask bit.
- `i` Sets the I bit in the instruction, causing the specified effect on PSTATE.I, the IRQ interrupt mask bit.
- `f` Sets the F bit in the instruction, causing the specified effect on PSTATE.F, the FIQ interrupt mask bit.

`<mode>` Is the number of the mode to change to, in the range 0 to 31, encoded in the "mode" field.

### Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0';
            if affectI then PSTATE.I = '0';
            if affectF then PSTATE.F = '0';
        if disable then
            if affectA then PSTATE.A = '1';
            if affectI then PSTATE.I = '1';
            if affectF then PSTATE.F = '1';
        if changemode then
            // WriteMode will set PSTATE.IL if 'mode' is not legal.
            AArch32.WriteMode(mode);
else
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0';
            if affectI then PSTATE.I = '0';
            if affectF then PSTATE.F = '0';
        if disable then
            if affectA then PSTATE.A = '1';
            if affectI then PSTATE.I = '1';
            if affectF then PSTATE.F = '1';
        if changemode then
            // WriteMode will set PSTATE.IL if 'mode' is not legal.
            AArch32.WriteMode(mode);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.40 CRC32

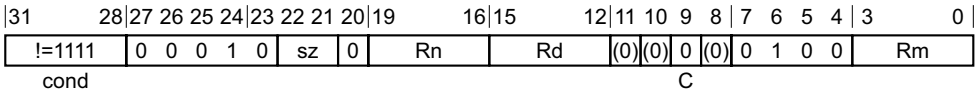
CRC32 performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In ARMv8-A, this is an OPTIONAL instruction.

**Note**

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

### A1




#### *CRC32B variant*

Applies when sz = 00.

```
CRC32B{<q>} <Rd>, <Rn>, <Rm>
```

#### *CRC32H variant*

Applies when sz = 01.

```
CRC32H{<q>} <Rd>, <Rn>, <Rm>
```

#### *CRC32W variant*

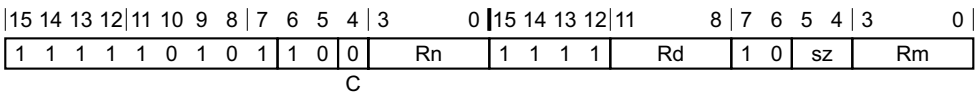
Applies when sz = 10.

```
CRC32W{<q>} <Rd>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;
```

### T1




#### *CRC32B variant*

Applies when sz = 00.

```
CRC32B{<q>} <Rd>, <Rn>, <Rm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *CRC32H variant*

Applies when sz = 01.

```
CRC32H{<q>} <Rd>, <Rn>, <Rm>
```

#### *CRC32W variant*

Applies when sz = 10.

```
CRC32W{<q>} <Rd>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *CRC32, CRC32C* on page K1-5434.

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. An CRC32 instruction must be unconditional. |
| <Rd> | Is the general-purpose accumulator output register, encoded in the "Rd" field. |
| <Rn> | Is the general-purpose accumulator input register, encoded in the "Rn" field. |
| <Rm> | Is the general-purpose data source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    acc = R[n];             // accumulator
    val = R[m]<size-1:0>;   // input value
    poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
    tempacc = BitReverse(acc):Zeros(size);
    tempval = BitReverse(val):Zeros(32);
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.41 CRC32C

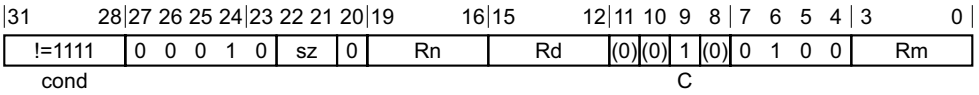
CRC32C performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In ARMv8-A, this is an OPTIONAL instruction.

**Note**

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

### A1




#### *CRC32CB variant*

Applies when sz = 00.

```
CRC32CB{<q>} <Rd>, <Rn>, <Rm>
```

#### *CRC32CH variant*

Applies when sz = 01.

```
CRC32CH{<q>} <Rd>, <Rn>, <Rm>
```

#### *CRC32CW variant*

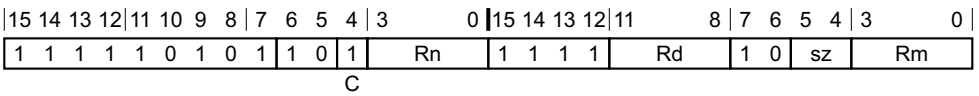
Applies when sz = 10.

```
CRC32CW{<q>} <Rd>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;
```

### T1




#### *CRC32CB variant*

Applies when sz = 00.

```
CRC32CB{<q>} <Rd>, <Rn>, <Rm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *CRC32CH variant*

Applies when sz = 01.

```
CRC32CH{<q>} <Rd>, <Rn>, <Rm>
```

#### *CRC32CW variant*

Applies when sz = 10.

```
CRC32CW{<q>} <Rd>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *CRC32, CRC32C* on page K1-5434.

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. An CRC32C instruction must be unconditional. |
| <Rd> | Is the general-purpose accumulator output register, encoded in the "Rd" field. |
| <Rn> | Is the general-purpose accumulator input register, encoded in the "Rn" field. |
| <Rm> | Is the general-purpose data source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

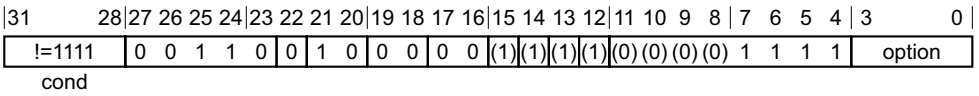
    acc = R[n];             // accumulator
    val = R[m]<size-1:0>;   // input value
    poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
    tempacc = BitReverse(acc):Zeros(size);
    tempval = BitReverse(val):Zeros(32);
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.42 DBG

Debug Hint provides a hint to debug and related systems. See the system documentation for what use (if any) is made of this instruction.

### A1



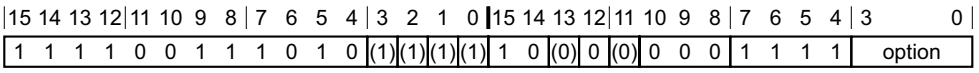

#### *A1 variant*

```
DBG{<c>}{<q>} #<option>
```

#### *Decode for this encoding*

```
// Any decoding of 'option' is specified by the debug system
```

### T1




#### *T1 variant*

```
DBG{<c>}{<q>} #<option>
```

#### *Decode for this encoding*

```
// Any decoding of 'option' is specified by the debug system
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<option> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "option" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Debug(option);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.43 DCPS1, DCPS2, DCPS3

Debug Change PE State allows the debugger to move the PE into a higher Exception Level or to a specific mode at the current Exception Level.

These instructions are always UNDEFINED in Non-debug state.

`DCPS1` targets EL1 and:

- If EL1 is using AArch32, the PE enters SVC mode. If EL3 is using AArch32, Secure SVC is an EL3 mode. This means `DCPS1` causes the PE to enter EL3.
- If EL1 is using AArch64, the PE enters EL1h, and executes future instructions as A64 instructions.

`DCPS2` targets EL2 and:

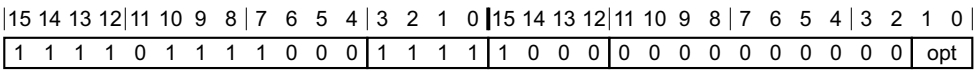
- If EL2 is using AArch32, the PE enters Hyp mode.
- If EL2 is using AArch64, the PE enters EL2h, and executes future instructions as A64 instructions.

`DCPS3` targets EL3 and:

- If EL3 is using AArch32, the PE enters Monitor mode.
- If EL3 is using AArch64, the PE enters EL3h, and executes future instructions as A64 instructions.

For more information on the operation of these instructions, see DCPS.

### T1




#### *DCPS1 variant*

Applies when `opt = 01`.

`DCPS1`

#### *DCPS2 variant*

Applies when `opt = 10`.

`DCPS2`

#### *DCPS3 variant*

Applies when `opt = 11`.

`DCPS3`

#### *Decode for all variants of this encoding*

```
if !Halted() || opt == '00' then UNDEFINED;
```

### Operation

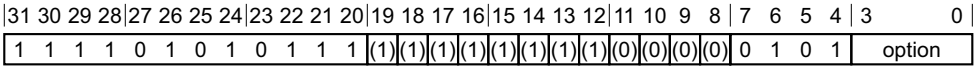
```
DCPSInstruction(opt);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.44 DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see .

### A1



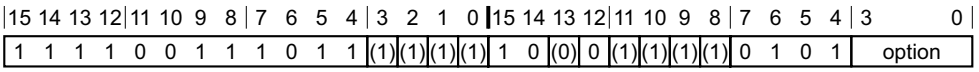

#### *A1 variant*

```
DMB{<c>}{<q>} {<option>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T1




#### *T1 variant*

```
DMB{<c>}{<q>} {<option>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-2446. Must be AL or omitted.
For encoding T1: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<option> Specifies an optional limitation on the barrier operation. Values are:

- SY — Full system is the required shareability domain, reads and writes are the required access types in both and . Can be omitted. This option is referred to as the full system DMB. Encoded as option = 0b1111.
- ST — Full system is the required shareability domain, writes are the required access type in both and . SYST is a synonym for ST. Encoded as option = 0b1110.
- LD — Full system is the required shareability domain, reads are the required access type in , and reads and writes are the required access types in . Encoded as option = 0b1101.
- ISH — Inner Shareable is the required shareability domain, reads and writes are the required access types in both and . Encoded as option = 0b1011.
- ISHST — Inner Shareable is the required shareability domain, writes are the required access type in both and . Encoded as option = 0b1010.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISHLD — Inner Shareable is the required shareability domain, reads are the required access type in , and reads and writes are the required access types in . Encoded as option = 0b1001.

NSH — Non-shareable is the required shareability domain, reads and writes are the required access types in both  and . Encoded as option = 0b0111.

NSHST — Non-shareable is the required shareability domain, writes are the required access type in both  and . Encoded as option = 0b0110.

NSHLD — Non-shareable is the required shareability domain, reads are the required access type in , and reads and writes are the required access types in . Encoded as option = 0b0101.

OSH — Outer Shareable is the required shareability domain, reads and writes are the required access types in both  and . Encoded as option = 0b0011.

OSHST — Outer Shareable is the required shareability domain, writes are the required access type in both  and . Encoded as option = 0b0010.

OSHLD — Outer Shareable is the required shareability domain, reads are the required access type in , and reads and writes are the required access types in . Encoded as option = 0b0001.

All other encodings of option are reserved. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior.

**Note**

The instruction supports the following alternative <option> values, but ARM recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST is an alias for NSHST.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0001'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Reads;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0011'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0101'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Reads;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1101'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Reads;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;

    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        if HCR.BSU == '11' then
            domain = MBReqDomain_FullSystem;
        if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
            domain = MBReqDomain_OuterShareable;
        if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
            domain = MBReqDomain_InnerShareable;

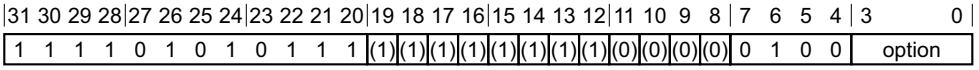
    DataMemoryBarrier(domain, types);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.45 DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see .

### A1



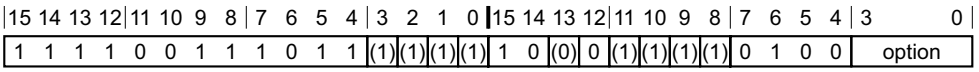

#### *A1 variant*

```
DSB{<c>}{<q>} {<option>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T1




#### *T1 variant*

```
DSB{<c>}{<q>} {<option>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-2446. Must be AL or omitted.
For encoding T1: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<option> Specifies an optional limitation on the barrier operation. Values are:

- SY — Full system is the required shareability domain, reads and writes are the required access types in both and . Can be omitted. This option is referred to as the full system DMB. Encoded as option = 0b1111.
- ST — Full system is the required shareability domain, writes are the required access type in both and . SYST is a synonym for ST. Encoded as option = 0b1110.
- LD — Full system is the required shareability domain, reads are the required access type in , and reads and writes are the required access types in . Encoded as option = 0b1101.
- ISH — Inner Shareable is the required shareability domain, reads and writes are the required access types in both and . Encoded as option = 0b1011.
- ISHST — Inner Shareable is the required shareability domain, writes are the required access type in both and . Encoded as option = 0b1010.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

ISHLD — Inner Shareable is the required shareability domain, reads are the required access type in , and reads and writes are the required access types in . Encoded as option = 0b1001.

NSH — Non-shareable is the required shareability domain, reads and writes are the required access types in both  and . Encoded as option = 0b0111.

NSHST — Non-shareable is the required shareability domain, writes are the required access type in both  and . Encoded as option = 0b0110.

NSHLD — Non-shareable is the required shareability domain, reads are the required access type in , and reads and writes are the required access types in . Encoded as option = 0b0101.

OSH — Outer Shareable is the required shareability domain, reads and writes are the required access types in both  and . Encoded as option = 0b0011.

OSHST — Outer Shareable is the required shareability domain, writes are the required access type in both  and . Encoded as option = 0b0010.

OSHLD — Outer Shareable is the required shareability domain, reads are the required access type in , and reads and writes are the required access types in . Encoded as option = 0b0001.

All other encodings of option are reserved. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior.

**Note**

The instruction supports the following alternative <option> values, but ARM recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST is an alias for NSHST.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0001'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Reads;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0011'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0101'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Reads;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1101'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Reads;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;

    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1}  then
        if HCR.BSU == '11' then
            domain = MBReqDomain_FullSystem;
        if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
            domain = MBReqDomain_OuterShareable;
        if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
            domain = MBReqDomain_InnerShareable;

    DataSynchronizationBarrier(domain, types);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.46 EOR, EORS (immediate)

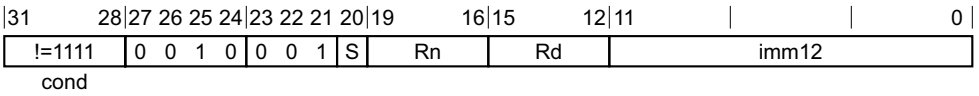
Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1




#### *EOR variant*

Applies when S = 0.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *EORS variant*

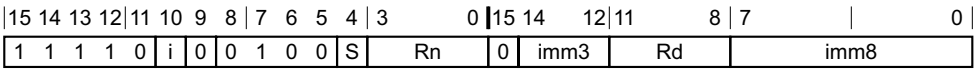
Applies when S = 1.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

### T1




#### *EOR variant*

Applies when S = 0.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *EORS variant*

Applies when S = 1 && Rd != 1111.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE TEQ (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.47 EOR, EORS (register)

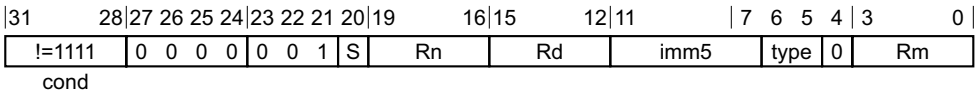
Bitwise Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1




#### *EOR, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *EOR, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *EORS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *EORS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

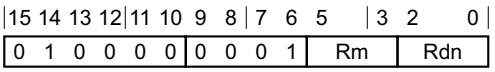
```
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



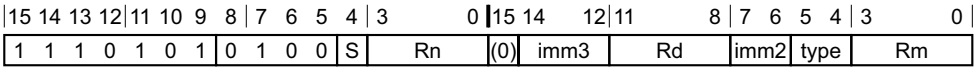

#### *T1 variant*

```
EOR<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
EORS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *EOR, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *EOR, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
EOR<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *EORS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && Rd != 1111 && imm2 = 00 && type = 11`.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *EORS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11) && Rd != 1111`.

```
EORS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
if Rd == '1111' && S == '1' then SEE TEQ (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rdn>` Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

`LSL` when `type = 00`

`LSR` when `type = 01`

`ASR` when `type = 10`

`ROR` when `type = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if `EORS <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `EORS <Rd>, <Rn>` had been written
- Inside an IT block, if `EOR<c> <Rd>, <Rn>, <Rd>` has `<Rd>` and `<Rn>` both in the range R0-R7, it is assembled using encoding T1 as though `EOR<c> <Rd>, <Rn>` had been written.

To prevent either of these happening, use the `.W` qualifier.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

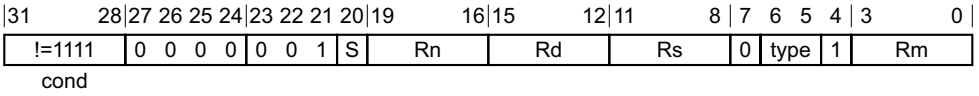
```
        ALUWritePC(result);
else
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.48 EOR, EORS (register-shifted register)

Bitwise Exclusive OR (register-shifted register) performs a bitwise Exclusive OR of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

#### *Not flag setting variant*

Applies when `S = 0`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

- `LSL` when `type = 00`
- `LSR` when `type = 01`
- `ASR` when `type = 10`
- `ROR` when `type = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.49 ERET

Exception Return.

The PE branches to the address held in the register holding the preferred return address, and restores PSTATE from SPSR_<current_mode>.

The register holding the preferred return address is:

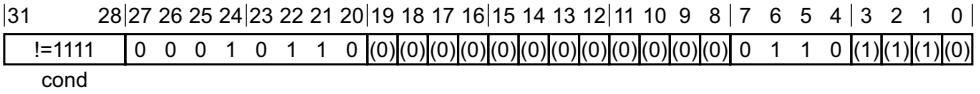
- ELR_hyp, when executing in Hyp mode.
- LR, when executing in a mode other than Hyp mode, User mode, or System mode.

The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.

Exception Return is CONSTRAINED UNPREDICTABLE in User mode and System mode.

In Debug state, the T1 encoding of ERET is decoded as DRPS.

### A1



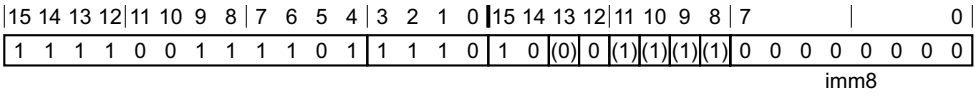

#### *A1 variant*

```
ERET{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T1




#### *T1 variant*

```
ERET{<c>}{<q>}
```

#### *Decode for this encoding*

```
if imm8 != '00000000' then SEE SUBS PC, LR and related instructions;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;                          // UNDEFINED or NOP
    else
        new_pc_value = if PSTATE.EL == EL2 then ELR_hyp else R[14];
        AArch32.ExceptionReturn(new_pc_value, SPSR[]);
```

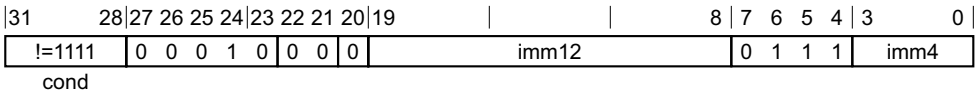

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.50 HLT

Halting breakpoint causes a software breakpoint to occur.

Halting breakpoint is always unconditional, even inside an IT block.

### A1



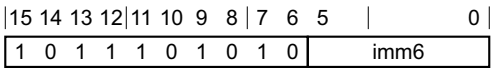

#### *A1 variant*

```
HLT{<q>} {#}<imm>
```

#### *Decode for this encoding*

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE; // HLT must be encoded with AL condition
// imm12:imm4 are for assembly/disassembly only and ignored by hardware
```

### T1




#### *T1 variant*

```
HLT{<q>} {#}<imm>
```

#### *Decode for this encoding*

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
// imm6 is for assembly/disassembly only and ignored by hardware
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *HLT* on page K1-5435.

## Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446. An HLT instruction must be unconditional.

<imm> For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

For encoding T1: is a 6-bit unsigned immediate, in the range 0 to 63, encoded in the "imm6" field. This value is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

## Operation for all encodings

```
EncodingSpecificOperations();
Halt(DebugHalt_HaltInstruction);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.51 HVC

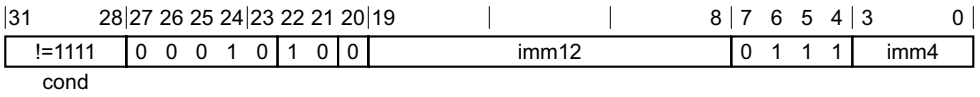
Hypervisor Call causes a Hypervisor Call exception. For more information see . Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is:

- UNDEFINED in Secure state, and in User mode in Non-secure state.
- When SCR.HCE is set to 0, UNDEFINED in Non-secure EL1 modes and CONSTRAINED UNPREDICTABLE in Hyp mode.

On executing an HVC instruction, the HSR reports the exception as a Hypervisor Call exception, using the EC value 0x12, and captures the value of the immediate argument, see *Use of the HSR* on page G4-4105.

#### A1



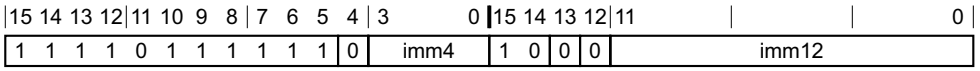

##### *A1 variant*

```
HVC{<q>} {#}<imm16>
```

##### *Decode for this encoding*

```
if cond != '1110' then UNPREDICTABLE;
imm16 = imm12:imm4;
// imm16 is for assembly/disassembly. It is reported in the HSR but otherwise is ignored by
// hardware. An HVC handler might interpret imm16, for example to determine the required service.
```

#### T1




##### *T1 variant*

```
HVC{<q>} {#}<imm16>
```

##### *Decode for this encoding*

```
// imm16 is for assembly/disassembly. It is reported in the HSR but otherwise is ignored by
// hardware. An HVC handler might interpret imm16, for example to determine the required service.
imm16 = imm4:imm12;
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and in particular, .

### Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446. An HVC instruction must be unconditional.

<imm16> For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.

### Operation for all encodings

```
EncodingSpecificOperations();
if !HaveEL(EL2) || PSTATE.EL == EL0 || IsSecure() then
    UNDEFINED;

hvc_enable = if HaveEL(EL3) then SCR_GEN[].HCE else NOT(HCR.HCD);
if hvc_enable == '0' then
    UNDEFINED;
else
    AArch32.CallHypervisor(imm16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

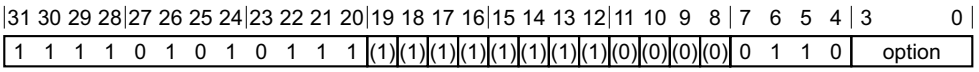
## F6.1.52 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE, so that all instructions following the `ISB` are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context changing operations executed before the `ISB` instruction are visible to the instructions fetched after the `ISB`. Context changing operations include changing the Address Space Identifier (ASID), TLB maintenance instructions, branch predictor maintenance operations, and all changes to the System registers.

In addition, any branches that appear in program order after the `ISB` instruction are written into the branch prediction logic with the context that is visible after the `ISB` instruction. This is needed to ensure correct execution of the instruction stream.

For more information, see *Instruction Synchronization Barrier (ISB)* on page E2-2375.

### A1



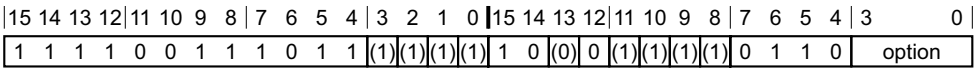

#### *A1 variant*

```
ISB{<c>}{<q>} {<option>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T1




#### *T1 variant*

```
ISB{<c>}{<q>} {<option>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-2446. Must be `AL` or omitted.
For encoding T1: see *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<option>` Specifies an optional limitation on the barrier operation. Values are:

`SY` Full system barrier operation, encoded as option = `0b1111`. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.53 IT

If-Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block are the same as, or the inverse of, the condition the IT instruction specifies for the first instruction in the block.

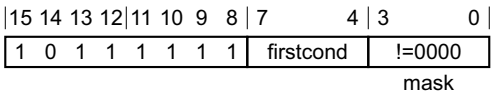
The IT instruction itself does not affect the condition flags, but the execution of the instructions in the IT block can change the condition flags.

16-bit instructions in the IT block, other than CMP, CMN and TST, do not set the condition flags. An IT instruction with the AL condition can change the behavior without conditional execution.

The architecture permits exception return to an instruction in the IT block only if the restoration of the CPSR restores ITSTATE to a state consistent with the conditions specified by the IT instruction. Any other exception return to an instruction in an IT block is UNPREDICTABLE. Any branch to a target instruction in an IT block is not permitted, and if such a branch is made it is UNPREDICTABLE what condition is used when executing that target instruction and any subsequent instruction in the IT block.

See also *Conditional instructions* on page F1-2409 and *Conditional execution* on page F2-2447.

### T1




#### *T1 variant*

```
IT{<x>{<y>{<z>}}}{<q>} <cond>
```

#### *Decode for this encoding*

```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *IT* on page K1-5435.

Related encodings: *Miscellaneous 16-bit instructions* on page F3-2471.

### Assembler symbols

<x> The condition for the second instruction in the IT block. If omitted, the "mask" field is set to 0b1000. If present it is encoded in the "mask[3]" field:

- T — firstcond[0]
- E — NOT firstcond[0]

<y> The condition for the third instruction in the IT block. If omitted and <x> is present, the "mask[2:0]" field is set to 0b100. If <y> is present it is encoded in the "mask[2]" field:

- T — firstcond[0]
- E — NOT firstcond[0]


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<z> The condition for the fourth instruction in the IT block. If omitted and <y> is present, the "mask[1:0]" field is set to 0b10. If <z> is present, the "mask[0]" field is set to 1, and it is encoded in the "mask[1]" field:

T firstcond[0]

E NOT firstcond[0]

<q> See *Standard assembler syntax fields* on page F2-2446.

<cond> The condition for the first instruction in the IT block, encoded in the "firstcond" field. See  for the range of conditions available, and the encodings.

The conditions specified in an IT instruction must match those specified in the syntax of the instructions in its IT block. When assembling to A32 code, assemblers check IT instruction syntax for validity but do not generate assembled instructions for them. See *Conditional instructions* on page F1-2409.

## Operation

```
EncodingSpecificOperations();
AArch32.CheckITEnabled(mask);
PSTATE.IT<7:0> = firstcond:mask;
```

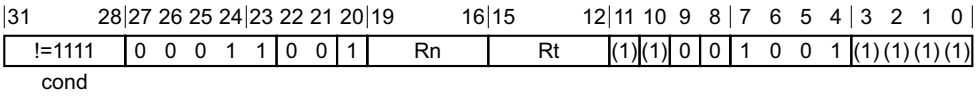

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.54 LDA

Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



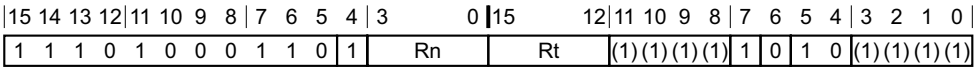

#### *A1 variant*

```
LDA{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDA{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = MemO[address, 4];
```

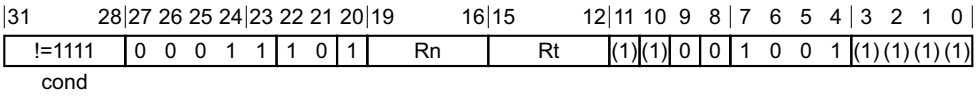

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.55 LDAB

Load-Acquire Byte loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



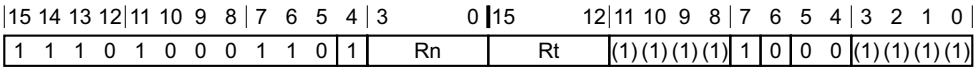

#### *A1 variant*

```
LDAB{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDAB{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = ZeroExtend(MemO[address, 1], 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.56 LDAEX

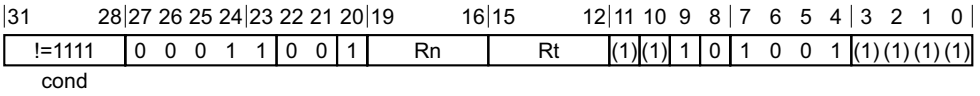
Load-Acquire Exclusive Word loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



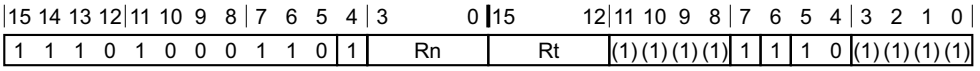

#### *A1 variant*

```
LDAEX{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDAEX{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDAEX* on page K1-5435.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 4);
    R[t] = MemO[address, 4];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.57 LDAEXB

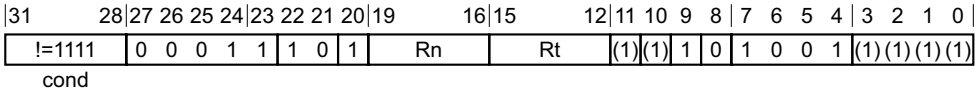
Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



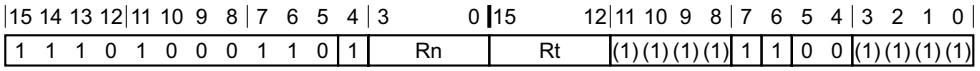

#### *A1 variant*

```
LDAEXB{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDAEXB{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDAEXB* on page K1-5435.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 1);
    R[t] = ZeroExtend(MemO[address, 1], 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.58 LDAEXD

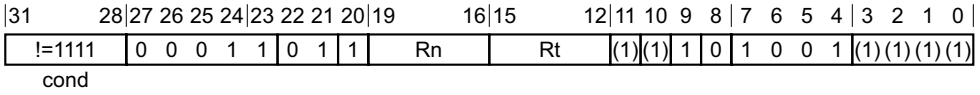
Load-Acquire Exclusive Doubleword loads a doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also acts as a barrier instruction with the ordering requirements described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



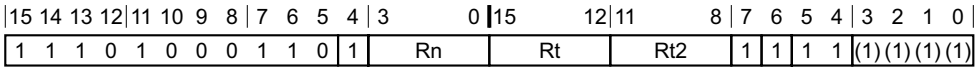

#### *A1 variant*

```
LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  t2 = t + 1;  n = UInt(Rn);
if Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDAEXD* on page K1-5435.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rt2> For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 8);
    value = MemO[address, 8];
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R[t]  = if BigEndian() then value<63:32> else value<31:0>;
    R[t2] = if BigEndian() then value<31:0>  else value<63:32>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.59 LDAEXH

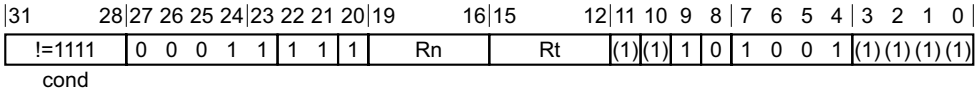
Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



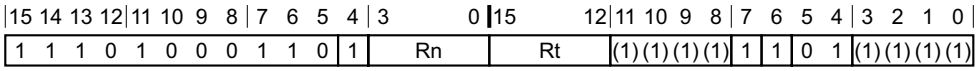

#### *A1 variant*

```
LDAEXH{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDAEXH{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDAEXH* on page K1-5436.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 2);
    R[t] = ZeroExtend(MemO[address, 2], 32);
```

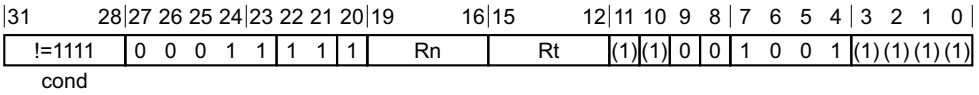

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.60 LDAH

Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



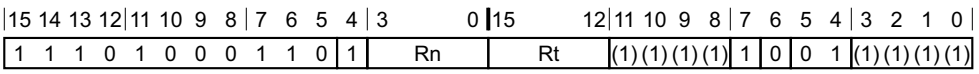

#### *A1 variant*

```
LDAH{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDAH{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = ZeroExtend(MemO[address, 2], 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.61 LDC, LDC2 (immediate)

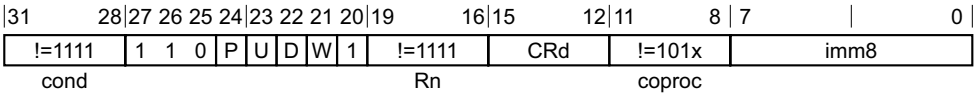
Load Coprocessor (immediate) loads data from consecutive memory addresses to a conceptual coprocessor.

This is a generic coprocessor instruction. The `coproc` field identifies the target conceptual coprocessor. This must be one of CP10, CP11, CP14, or CP15, and for these CP values, this manual defines the `imm8`, `CRd`, and `D` field values that are valid `LDC` and `LDC2` instructions. Other encodings are UNDEFINED. For more information see *Conceptual coprocessor support* on page E1-2353 and *General behavior of System registers* on page G4-4118.

In an implementation that includes EL2, the permitted `LDC` access to a System register can be trapped to Hyp mode, meaning that an attempt to execute an `LDC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *Trapping general Non-secure System register accesses to debug registers* on page G1-3871.

For simplicity, the `LDC` pseudocode does not show this possible trap to Hyp mode.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 1`.

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

#### *Unindexed variant*

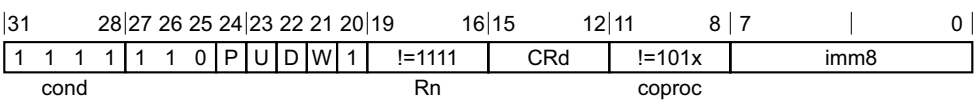
Applies when `P = 0 && U = 1 && W = 0`.

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
n = UInt(Rn);  cp = UInt(coproc);
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

### A2





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 1`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

#### *Unindexed variant*

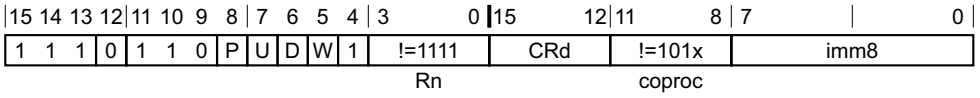
Applies when `P = 0 && U = 1 && W = 0`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then UNDEFINED;
n = UInt(Rn); cp = UInt(coproc);
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

### T1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 1`.

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

#### *Unindexed variant*

Applies when `P = 0 && U = 1 && W = 0`.

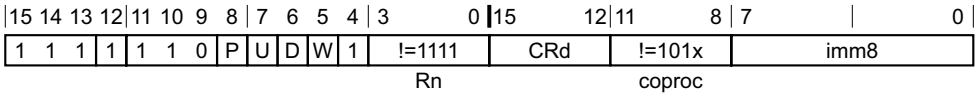
```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### Decode for all variants of this encoding

```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
n = UInt(Rn);  cp = UInt(coproc);
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

### T2




#### Offset variant

Applies when `P = 1 && W = 0`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

#### Post-indexed variant

Applies when `P = 0 && W = 1`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

Applies when `P = 1 && W = 1`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

#### Unindexed variant

Applies when `P = 0 && U = 1 && W = 0`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

#### Decode for all variants of this encoding

```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then UNDEFINED;
n = UInt(Rn); cp = UInt(coproc);
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

## Notes for all encodings

Advanced SIMD and floating-point: Extension register load/store instructions.

## Assembler symbols

| | |
|---|---|
| `L` | If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form. |
| `<c>` | For encoding A1, T1 and T2: see *Standard assembler syntax fields* on page F2-2446.<br>For encoding A2: see *Standard assembler syntax fields* on page F2-2446. `<c>` must be `AL` or omitted. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<coproc>` | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<CRd> Is the coprocessor register to be transferred, encoded in the "CRd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see LDC, LDC2 (literal).

<option> Is a coprocessor option, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0

+ when U = 1

<imm> Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Coproc_CheckInstr(cp, ThisInstr());
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    repeat
        Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());
        address = address + 4;
    until Coproc_DoneLoading(cp, ThisInstr());
    if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.62 LDC, LDC2 (literal)

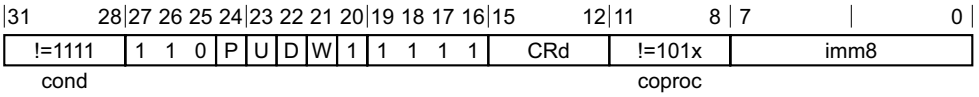
Load Coprocessor (literal) loads data from consecutive memory addresses to a conceptual coprocessor.

This is a generic coprocessor instruction. The `coproc` field identifies the target conceptual coprocessor. This must be one of CP10, CP11, CP14, or CP15, and for these CP values, this manual defines the `imm8`, `CRd`, and `D` field values that are valid `LDC` and `LDC2` instructions. Other encodings are UNDEFINED. For more information see *Conceptual coprocessor support* on page E1-2353 and *General behavior of System registers* on page G4-4118.

In an implementation that includes EL2, the permitted `LDC` access to a System register can be trapped to Hyp mode, meaning that an attempt to execute an `LDC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *Trapping general Non-secure System register accesses to debug registers* on page G1-3871.

For simplicity, the `LDC` pseudocode does not show this possible trap to Hyp mode.

### A1




#### A1 variant

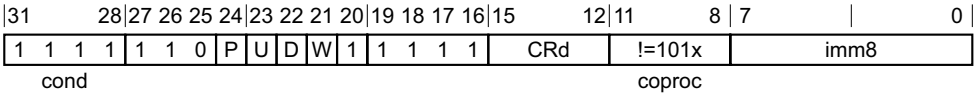
Applies when `!(P == 0 && U == 0 && W == 0)`.

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, <label>
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [PC, #{+/-}<imm>]
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [PC], <option>
```

#### Decode for this encoding

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
index = (P == '1');  add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

### A2




#### A2 variant

Applies when `!(P == 0 && U == 0 && W == 0)`.

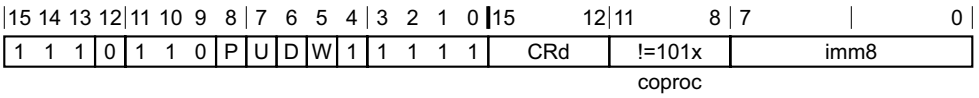
```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, <label>
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [PC, #{+/-}<imm>]
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [PC], <option>
```

#### Decode for this encoding

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then UNDEFINED;
index = (P == '1');  add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *T1 variant*

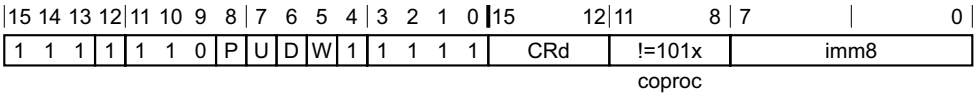
Applies when `!(P == 0 && U == 0 && W == 0)`.

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, <label>
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [PC, #{+/-}<imm>]
```

#### *Decode for this encoding*

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
index = (P == '1');  add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

### T2




#### *T2 variant*

Applies when `!(P == 0 && U == 0 && W == 0)`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, <label>
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [PC, #{+/-}<imm>]
```

#### *Decode for this encoding*

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then UNDEFINED;
index = (P == '1');  add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDC/LDC2 (literal)* on page K1-5436.

Advanced SIMD and floating-point: Extension register load/store instructions.

## Assembler symbols

| | |
|---|---|
| `L` | If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form. |
| `<c>` | For encoding A1, T1 and T2: see *Standard assembler syntax fields* on page F2-2446.<br>For encoding A2: see *Standard assembler syntax fields* on page F2-2446. `<c>` must be `AL` or omitted. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<coproc>` | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<CRd>` Is the coprocessor register to be transferred, encoded in the "CRd" field.

`<option>` Is a coprocessor option, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE` (encoded as U == 1). If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE (encoded as U == 0)`.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Coproc_CheckInstr(cp, ThisInstr());
    offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    address = if index then offset_addr else Align(PC,4);
    repeat
        Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());
        address = address + 4;
    until Coproc_DoneLoading(cp, ThisInstr());
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F6.1.63 LDM, LDMIA, LDMFD

Load Multiple (Increment After, Full Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register.

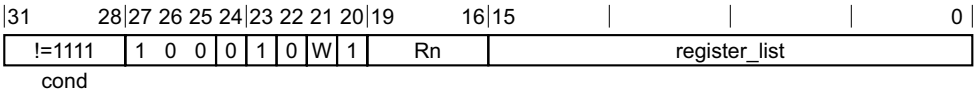
The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

Related system instructions are LDM (User registers) and LDM (exception return).

This instruction is used by the alias POP (multiple registers). See the *Alias conditions* on page F6-2712 table for details of when each alias is preferred.

### A1



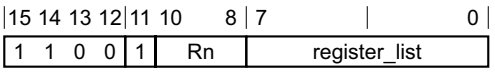

#### *A1 variant*

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

### T1



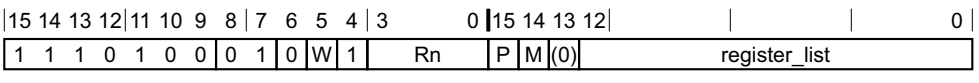

#### *T1 variant*

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

### T2





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *T2 variant*

```
LDM{IA}{<c>}.W <Rn>{!}, <registers> // Preferred syntax, if <Rn>, '!' and <registers> can be represented
in T1
LDMFD{<c>}.W <Rn>{!}, <registers> // Alternate syntax, Full Descending stack, if <Rn>, '!' and
<registers> can be represented in T1
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```

### *Decode for this encoding*

```
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDM/LDMIA/LDMFD (T32)* on page K1-5437 and .

## Alias conditions

| Alias | is preferred when |
|---|---|
| POP (multiple registers) | `W == '1' && Rn == '1101' && BitCount(register_list) > 1` |

## Assembler symbols

`IA` Is an optional suffix for the Increment After form.

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`!` For encoding A1 and T2: the address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

For encoding T1: the address adjusted by the size of the data loaded is written back to the base register. It is omitted if `<Rn>` is included in `<registers>`, otherwise it must be present.

`<registers>` For encoding A1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. ARM deprecates using these instructions with both the LR and the PC in the list.

For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field.

For encoding T2: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.64 LDM (exception return)

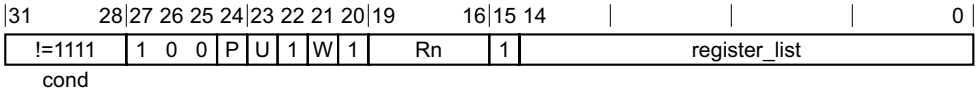
Load Multiple (exception return) loads multiple registers from consecutive memory locations using an address from a base register. The SPSR of the current mode is copied to the CPSR. An address adjusted by the size of the data loaded can optionally be written back to the base register.

The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

Load Multiple (exception return) is:

- UNDEFINED in Hyp mode.
- UNPREDICTABLE in debug state, and in User mode and System mode.

### A1




#### *A1 variant*

```
LDM{<amode>}{<c>}{<q>} <Rn>{!}, <registers_with_pc>^
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDM (exception return)* on page K1-5480.

### Assembler symbols

<amode> is one of:

| | |
|---|---|
| DA | Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0. |
| FA | Full Ascending. For this instruction, a synonym for DA. |
| DB | Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0. |
| EA | Empty Ascending. For this instruction, a synonym for DB. |
| IA | Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1. |
| FD | Full Descending. For this instruction, a synonym for IA. |
| IB | Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1. |
| ED | Empty Descending. For this instruction, a synonym for IB. |

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers_with_pc> Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must be specified in the register list, and the instruction causes a branch to the address (data) loaded into the PC. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

Instructions with similar syntax but without the PC included in the registers list are described in LDM (User registers).

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;                        // UNDEFINED or NOP
    else
        length = 4*BitCount(registers) + 4;
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;

        for i = 0 to 14
            if registers<i> == '1' then
                R[i] = MemA[address,4];  address = address + 4;
        new_pc_value = MemA[address,4];

        if wback && registers<n> == '0' then R[n] = if increment then R[n]+length else R[n]-length;
        if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;

        AArch32.ExceptionReturn(new_pc_value, SPSR[]);
```

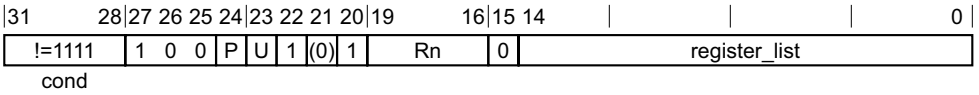

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.65 LDM (User registers)

In an EL1 mode other than System mode, Load Multiple (User registers) loads multiple User mode registers from consecutive memory locations using an address from a base register. The registers loaded cannot include the PC. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Load Multiple (User registers) is UNDEFINED in Hyp mode, and UNPREDICTABLE in User and System modes.

#### A1




##### *A1 variant*

```
LDM{<amode>}{<c>}{<q>} <Rn>, <registers_without_pc>^
```

##### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

#### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDM (User registers)* on page K1-5480.

#### Assembler symbols

<amode> is one of:

| | |
|---|---|
| DA | Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0. |
| FA | Full Ascending. For this instruction, a synonym for DA. |
| DB | Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0. |
| EA | Empty Ascending. For this instruction, a synonym for DB. |
| IA | Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1. |
| FD | Full Descending. For this instruction, a synonym for IA. |
| IB | Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1. |
| ED | Empty Descending. For this instruction, a synonym for IB. |

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<registers_without_pc>

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the LDM instruction. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must not be in the register list. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

Instructions with similar syntax but with the PC included in <registers_without_pc> are described in LDM (exception return).

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then UNPREDICTABLE;
    else
        length = 4*BitCount(registers);
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;
        for i = 0 to 14
            if registers<i> == '1' then  // Load User mode register
                Rmode[i, M32_User] = MemA[address,4];  address = address + 4;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.66 LDMDA, LDMFA

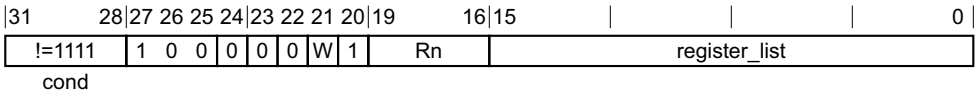
Load Multiple Decrement After (Full Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

Related system instructions are LDM (User registers) and LDM (exception return).

### A1




#### *A1 variant*

```
LDMDA{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMFA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Ascending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDMDA/LDMFA* on page K1-5438.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| ! | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. |
| <registers> | Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. ARM deprecates using these instructions with both the LR and the PC in the list. |

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    LoadWritePC(MemA[address,4]);
if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.67 LDMDB, LDMEA

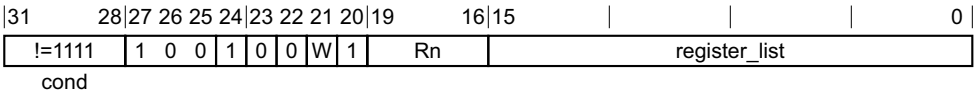
Load Multiple Decrement Before (Empty Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

Related system instructions are LDM (User registers) and LDM (exception return).

### A1



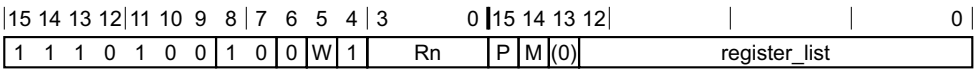

#### *A1 variant*

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMEA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMEA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDMDB/LDMEA* on page K1-5439.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> For encoding A1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. ARM deprecates using these instructions with both the LR and the PC in the list.

For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.68 LDMIB, LDMED

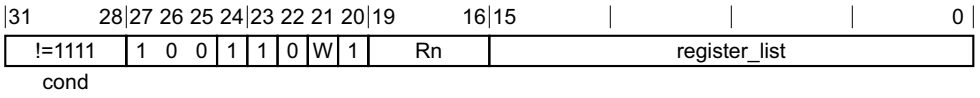
Load Multiple Increment Before (Empty Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

Related system instructions are LDM (User registers) and LDM (exception return).

### A1




#### *A1 variant*

```
LDMIB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMED{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Descending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDMIB/LDMED* on page K1-5438.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| ! | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. |
| <registers> | Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. ARM deprecates using these instructions with both the LR and the PC in the list. |

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    LoadWritePC(MemA[address,4]);
if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

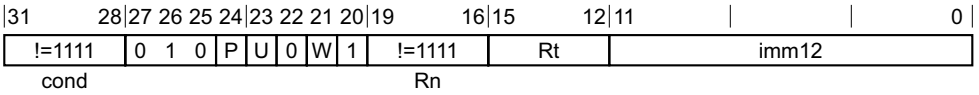

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.69 LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

This instruction is used by the alias POP (single register). See the *Alias conditions* on page F6-2726 table for details of when each alias is preferred.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

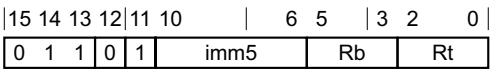
Applies when `P = 1 && W = 1`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE LDR (literal);
if P == '0' && W == '1' then SEE LDRT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && n == t then UNPREDICTABLE;
```

### T1




#### *T1 variant*

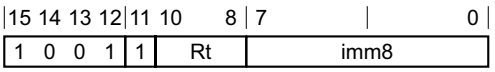
```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2



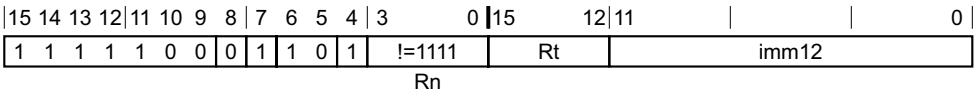

#### *T2 variant*

```
LDR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T3



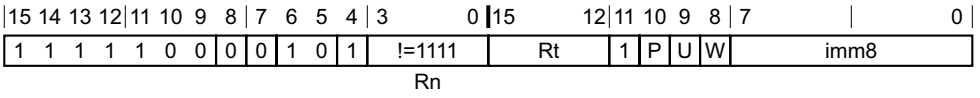

#### *T3 variant*

```
LDR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1 or T2
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32); index = TRUE;  add = TRUE;
wback = FALSE; if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T4




#### *Offset variant*

Applies when `P = 1 && U = 0 && W = 0`.

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 1`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.

```
LDR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE LDR (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);
imm32 = ZeroExtend(imm8, 32); index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDR (immediate, T32)* on page K1-5440 and .

### Alias conditions

| Alias | of variant | is preferred when |
|---|---|---|
| POP (single register) | A1 (post-indexed) | `P == '0' && U == '1' && W == '0' && Rn == '1101' && imm12 == '000000000100'` |
| POP (single register) | T4 (post-indexed) | `Rn == '1101' && U == '1' && imm8 == '00000100'` |

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T3 and T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

`<Rn>` For encoding A1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDR (literal).

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1 and T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.

For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T4: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

### Operation for all encodings

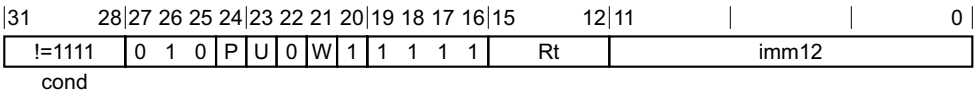
```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.70 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *A1 variant*

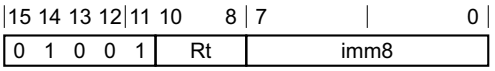
Applies when `!(P == 0 && W == 1)`.

```
LDR{<c>}{<q>} <Rt>, <label> // Normal form
LDR{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form
```

#### *Decode for this encoding*

```
if P == '0' && W == '1' then SEE LDRT;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if wback then UNPREDICTABLE;
```

### T1



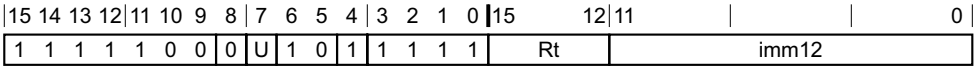

#### *T1 variant*

```
LDR{<c>}{<q>} <Rt>, <label> // Normal form
```

#### *Decode for this encoding*

```
t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

### T2




#### *T2 variant*

```
LDR{<c>}.W <Rt>, <label> // Preferred syntax, and <Rt>, <label> can be represented in T1
LDR{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDR{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

#### *Decode for this encoding*

```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDR (literal)* on page K1-5440.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

`<label>` For encoding A1 and T2: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are Multiples of four in the range 0 to 1020.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T2: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,4];
    if t == 15 then
        if address<1:0> == '00' then
            LoadWritePC(data);
        else
            UNPREDICTABLE;
    else
        R[t] = data;
```

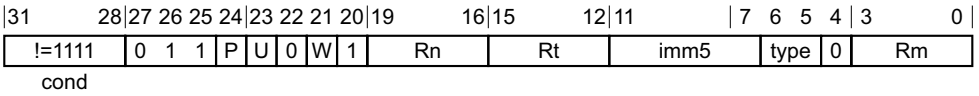

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.71 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses, see *Memory accesses* on page F2-2452.

The T32 form of LDR (register) does not support register writeback.

### A1




#### *Offset variant*

Applies when P = 1 && W = 0.

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

#### *Post-indexed variant*

Applies when P = 0 && W = 0.

```
LDR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

#### *Pre-indexed variant*

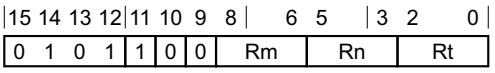
Applies when P = 1 && W = 1.

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE LDRT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

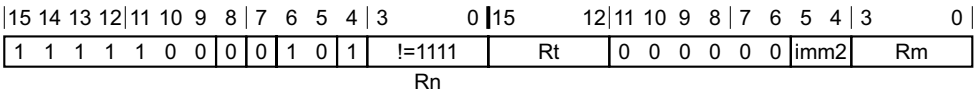
```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *T2 variant*

```
LDR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDR (register, A32)* on page K1-5441.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This branch is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see .

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

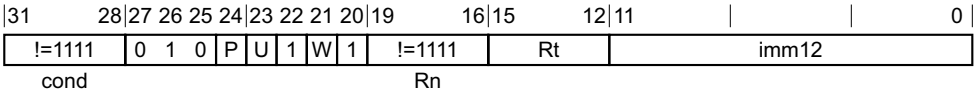
```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
        offset_addr = if add then (R[n] + offset) else (R[n] - offset);
        address = if index then offset_addr else R[n];
        data = MemU[address,4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
        offset_addr = (R[n] + offset);
        address = offset_addr;
        data = MemU[address,4];
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.72 LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

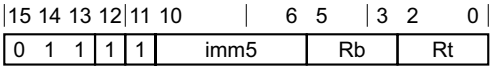
Applies when `P = 1 && W = 1`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE LDRB (literal);
if P == '0' && W == '1' then SEE LDRBT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

### T1



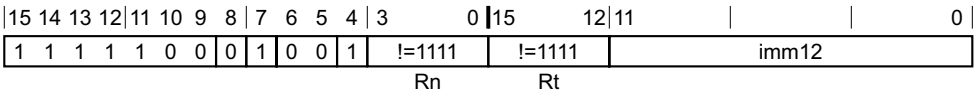

#### *T1 variant*

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T2





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

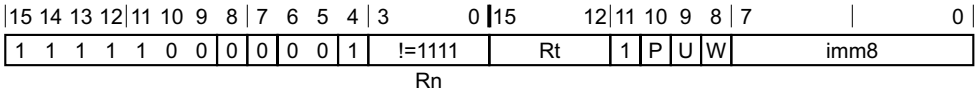
#### T2 variant

```
LDRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### Decode for this encoding

```
if Rt == '1111' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// ARMv8-A removes UNPREDICTABLE for R13
```

### T3




#### Offset variant

Applies when `Rt != 1111 && P = 1 && U = 0 && W = 0`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### Post-indexed variant

Applies when `P = 0 && W = 1`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

Applies when `P = 1 && W = 1`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### Decode for all variants of this encoding

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLD, PLDW (immediate);
if Rn == '1111' then SEE LDRB (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRBT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if  (t == 15 &&  W == '1') || (wback && n == t) then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRB (immediate, T32)* on page K1-5441 and .

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRB (literal). |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0

+ when U = 1

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

## Operation for all encodings

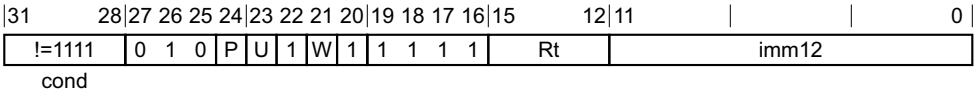
```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        R[t] = ZeroExtend(MemU[address,1], 32);
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        R[t] = ZeroExtend(MemU[address,1], 32);
        if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.73 LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *A1 variant*

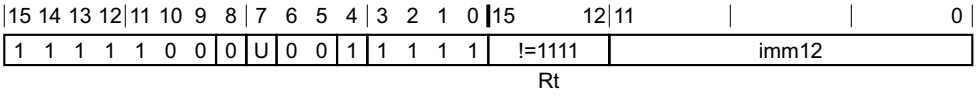
Applies when `!(P == 0 && W == 1)`.

```
LDRB{<c>}{<q>} <Rt>, <label> // Normal form
LDRB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form
```

#### *Decode for this encoding*

```
if P == '0' && W == '1' then SEE LDRBT;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDRB{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

#### *Decode for this encoding*

```
if Rt == '1111' then SEE PLD;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRB (literal)* on page K1-5442.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as U == 0.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.

## Operation for all encodings

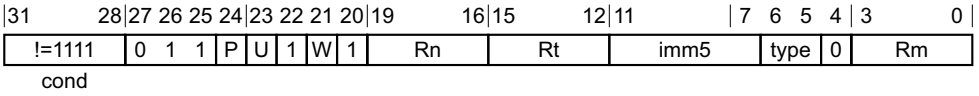
```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address,1], 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.74 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

#### *Pre-indexed variant*

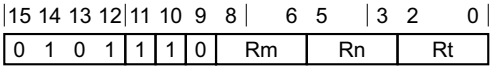
Applies when `P = 1 && W = 1`.

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE LDRBT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

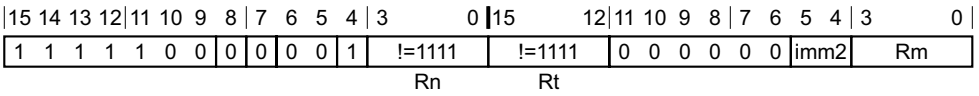
```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *T2 variant*

```
LDRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

#### *Decode for this encoding*

```
if Rt == '1111' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRB (register)* on page K1-5442.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.<br>For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field. |
| +/- | Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when `U = 0`<br>+ when `U = 1` |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <shift> | The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see . |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1],32);
    if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.75 LDRBT

Load Register Byte Unprivileged loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-2452.

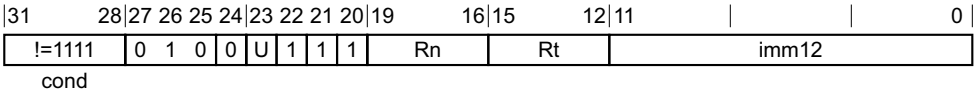
The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

### A1



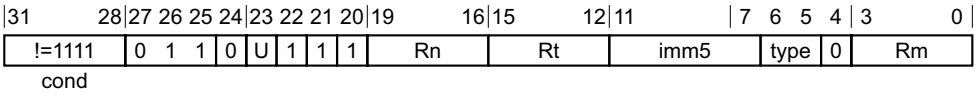

#### *A1 variant*

```
LDRBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

### A2



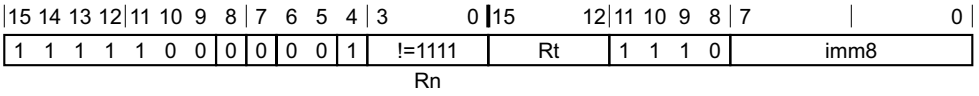

#### *A2 variant*

```
LDRBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for this encoding*

```
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRBT* on page K1-5442.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.<br>For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| +/- | For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1<br>For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <shift> | The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see . |
| + | Specifies the offset is added to the base register. |
| <imm> | For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.<br>For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

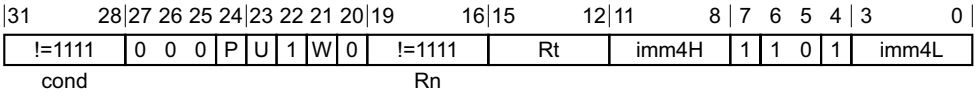
```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = ZeroExtend(MemU_unpriv[address,1],32);
    if postindex then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.76 LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

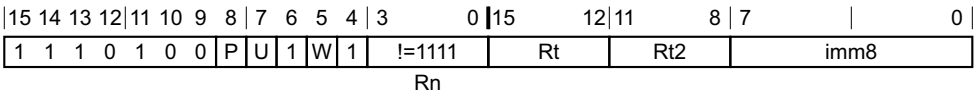
Applies when `P = 1 && W = 1`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE LDRD (literal);
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

### T1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 1`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

### *Decode for all variants of this encoding*

```
if P == '0' && W == '0' then SEE "Related encodings";
if Rn == '1111' then SEE LDRD (literal);
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRD (immediate)* on page K1-5443.

Related encodings: *Load/store dual, load/store exclusive, table branch* on page F3-2480.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Rt2>` For encoding A1: is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRD (literal).

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as <imm>/4.

## Operation for all encodings

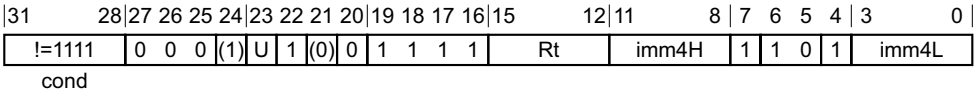
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian()  then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.77 LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



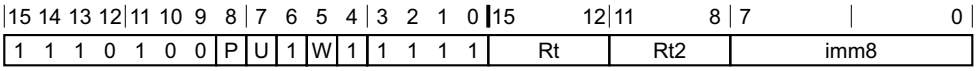

#### *A1 variant*

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // Normal form
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // Alternative form
```

#### *Decode for this encoding*

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t2 == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

Applies when `!(P == 0 && W == 0)`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // Normal form
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // Alternative form
```

#### *Decode for this encoding*

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
if W == '1' then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRD (literal)* on page K1-5444.

Related encodings: *Load/store dual, load/store exclusive, table branch* on page F3-2480.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Rt2>` For encoding A1: is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<label>` For encoding A1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.

## Operation for all encodings

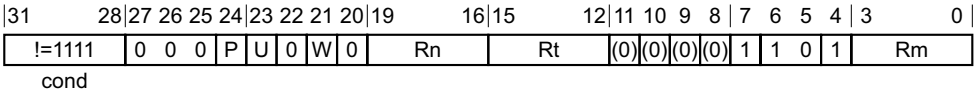
```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian() then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.78 LDRD (register)

Load Register Dual (register) calculates an address from a base register value and a register offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

#### *Decode for all variants of this encoding*

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 || m == t || m == t2 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRD (register)* on page K1-5444.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

`<Rt2>` Is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when `U = 0`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian() then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];

    if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

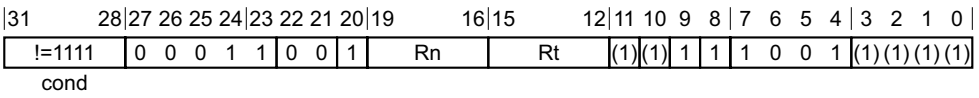
## F6.1.79 LDREX

Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



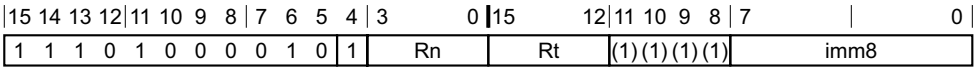

#### *A1 variant*

```
LDREX{<c>}{<q>} <Rt>, [<Rn> {, {#}<imm>}]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if t == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDREX* on page K1-5445.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| <imm> | For encoding A1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    AArch32.SetExclusiveMonitors(address,4);
    R[t] = MemA[address,4];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

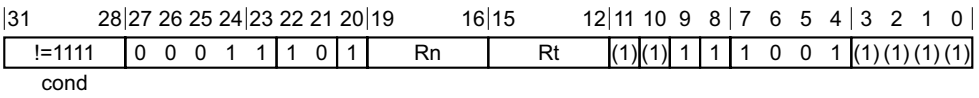
## F6.1.80 LDREXB

Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



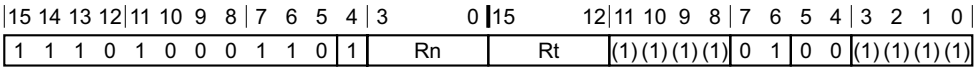

#### *A1 variant*

```
LDREXB{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDREXB{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDREXB* on page K1-5445.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

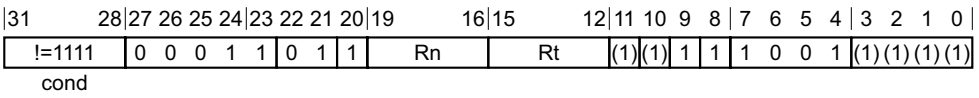
## F6.1.81 LDREXD

Load Register Exclusive Doubleword derives an address from a base register value, loads a 64-bit doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



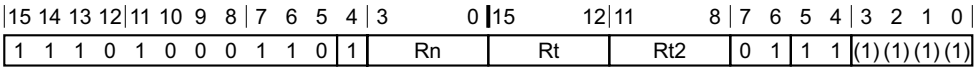

#### *A1 variant*

```
LDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  t2 = t + 1;  n = UInt(Rn);
if Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDREXD* on page K1-5446.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rt2> For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,8);
    value = MemA[address,8];
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R[t]  = if BigEndian() then value<63:32> else value<31:0>;
    R[t2] = if BigEndian() then value<31:0> else value<63:32>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

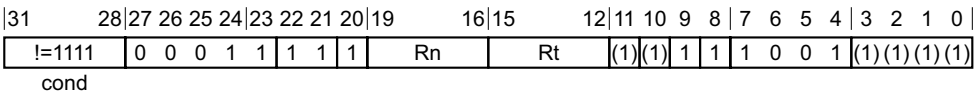
## F6.1.82 LDREXH

Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



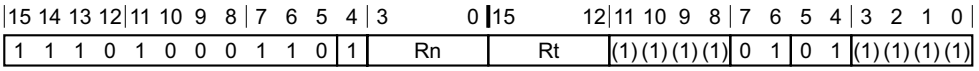

#### *A1 variant*

```
LDREXH{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDREXH{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDREXH* on page K1-5446.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

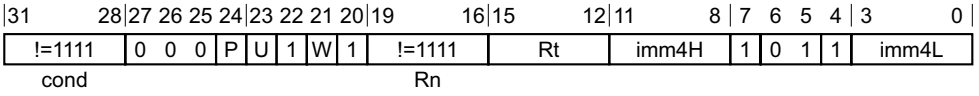
```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2], 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.83 LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

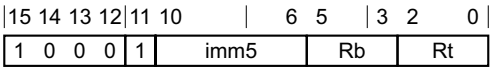
Applies when `P = 1 && W = 1`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE LDRH (literal);
if P == '0' && W == '1' then SEE LDRHT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

### T1



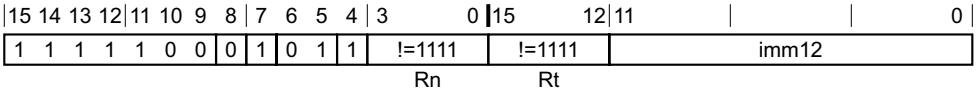

#### *T1 variant*

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T2





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

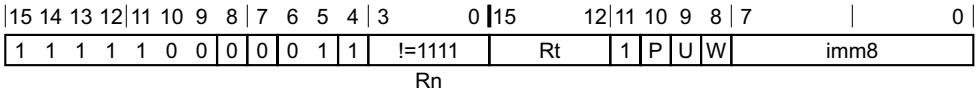
#### T2 variant

```
LDRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### Decode for this encoding

```
if Rt == '1111' then SEE PLD (immediate);
if Rn == '1111' then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// ARMv8-A removes UNPREDICTABLE for R13
```

### T3




#### Offset variant

Applies when `Rt != 1111 && P = 1 && U = 0 && W = 0`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### Post-indexed variant

Applies when `P = 0 && W = 1`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

Applies when `P = 1 && W = 1`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### Decode for all variants of this encoding

```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLDW (immediate);
if P == '1' && U == '1' && W == '0' then SEE LDRHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRH (immediate, T32)* on page K1-5447 and .

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRH (literal). |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

## Operation for all encodings

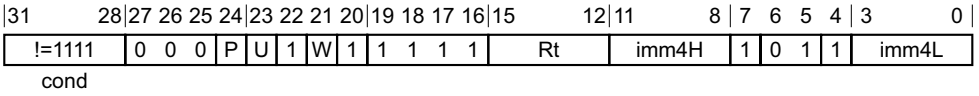
```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,2];
        if wback then R[n] = offset_addr;
        R[t] = ZeroExtend(data, 32);
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,2];
        if wback then R[n] = offset_addr;
        R[t] = ZeroExtend(data, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.84 LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *A1 variant*

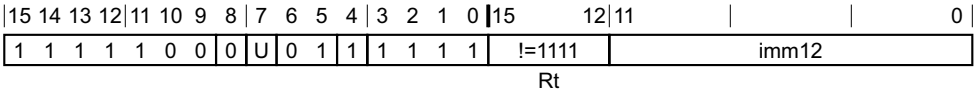
Applies when `!(P == 0 && W == 1)`.

```
LDRH{<c>}{<q>} <Rt>, <label> // Normal form
LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form
```

#### *Decode for this encoding*

```
if P == '0' && W == '1' then SEE LDRHT;
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDRH{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

#### *Decode for this encoding*

```
if Rt == '1111' then SEE PLD (literal);
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRH (literal)* on page K1-5447.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<label>` For encoding A1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.

## Operation for all encodings

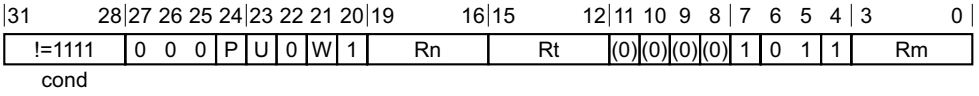
```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = ZeroExtend(data, 32);
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F6.1.85 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

#### *Pre-indexed variant*

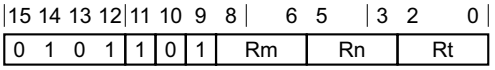
Applies when `P = 1 && W = 1`.

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE LDRHT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

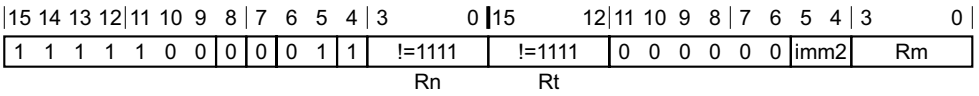
```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *T2 variant*

```
LDRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' then SEE PLDW (register);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRH (register)* on page K1-5447.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.<br>For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field. |
| +/- | Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.86 LDRHT

Load Register Halfword Unprivileged loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-2452.

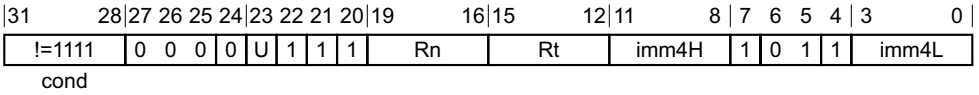
The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

### A1



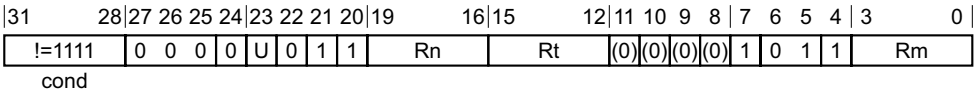

#### *A1 variant*

```
LDRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

### A2



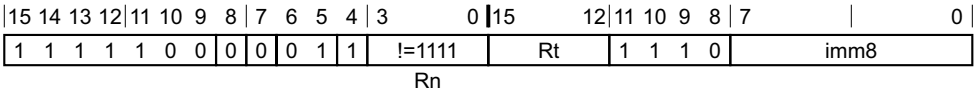

#### *A2 variant*

```
LDRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDRHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for this encoding*

```
if Rn == '1111' then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRHT* on page K1-5448.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

## Operation for all encodings

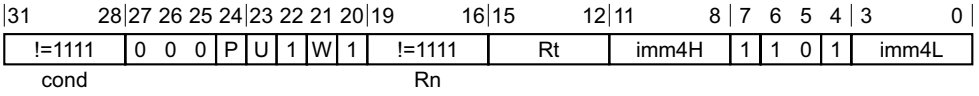
```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.87 LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

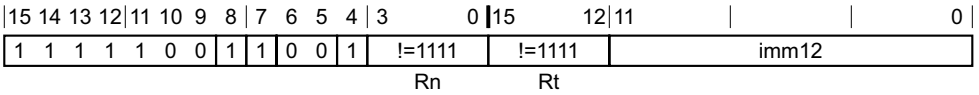
Applies when `P = 1 && W = 1`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE LDRSB (literal);
if P == '0' && W == '1' then SEE LDRSBT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

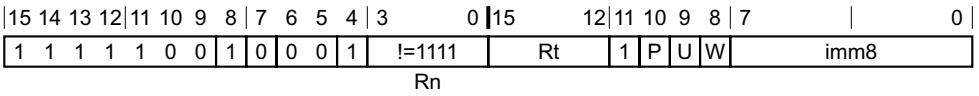
```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### *Decode for this encoding*

```
if Rt == '1111' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// ARMv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *Offset variant*

Applies when `P = 1 && U = 0 && W = 0`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 1`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRSBT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSB (immediate)* on page K1-5448.

## Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRSB (literal). |
| `+/-` | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>`-` when `U = 0`<br>`+` when `U = 1` |
| `+` | Specifies the offset is added to the base register. |
| `<imm>` | For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.<br>For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

### Operation for all encodings

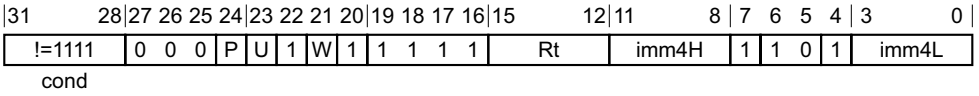
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.88 LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *A1 variant*

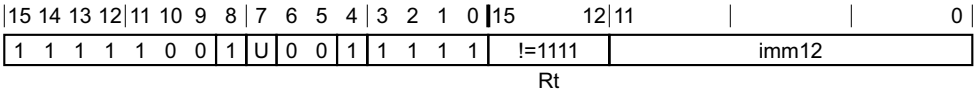
Applies when `!(P == 0 && W == 1)`.

```
LDRSB{<c>}{<q>} <Rt>, <label> // Normal form
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form
```

#### *Decode for this encoding*

```
if P == '0' && W == '1' then SEE LDRSBT;
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDRSB{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

#### *Decode for this encoding*

```
if Rt == '1111' then SEE PLI;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSB (literal)* on page K1-5448.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<label>` For encoding A1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.

## Operation for all encodings

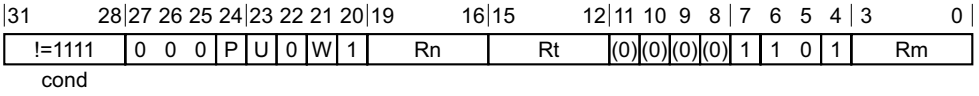
```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = SignExtend(MemU[address,1], 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.89 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

#### *Pre-indexed variant*

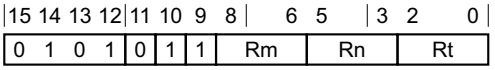
Applies when `P = 1 && W = 1`.

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE LDRSBT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

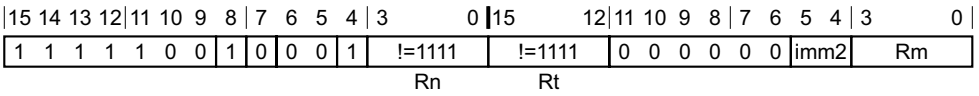
```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *T2 variant*

```
LDRSB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

#### *Decode for this encoding*

```
if Rt == '1111' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSB (register)* on page K1-5449.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.90 LDRSBT

Load Register Signed Byte Unprivileged loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-2452.

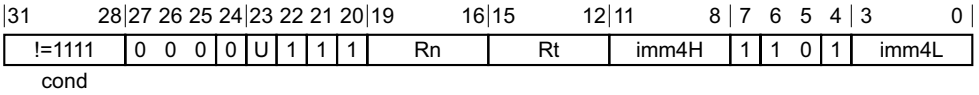
The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRSBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

### A1



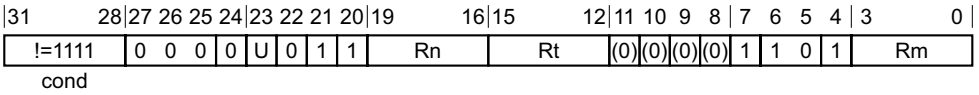

#### *A1 variant*

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

### A2



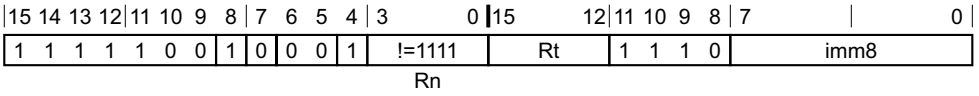

#### *A2 variant*

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for this encoding*

```
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSBT* on page K1-5449.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

## Operation for all encodings

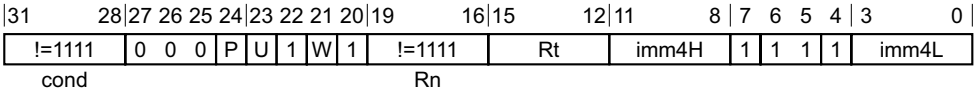
```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
    if postindex then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.91 LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

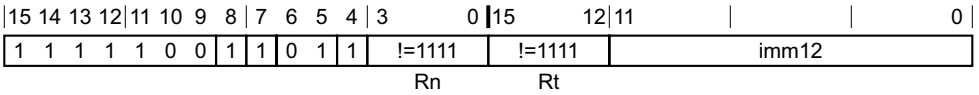
Applies when `P = 1 && W = 1`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE LDRSH (literal);
if P == '0' && W == '1' then SEE LDRSHT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

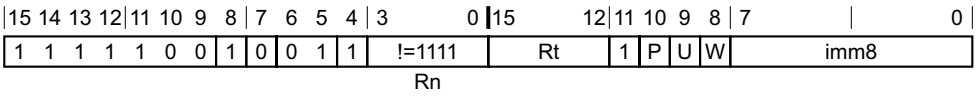
```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// ARMv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *Offset variant*

Applies when `P = 1 && U = 0 && W = 0`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 1`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related instructions";
if P == '1' && U == '1' && W == '0' then SEE LDRSHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSH (immediate)* on page K1-5449.

Related instructions: *Load/store single* on page F3-2492.

## Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRSH (literal). |
| `+/-` | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>`-` when `U = 0`<br>`+` when `U = 1` |
| `+` | Specifies the offset is added to the base register. |
| `<imm>` | For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

### Operation for all encodings

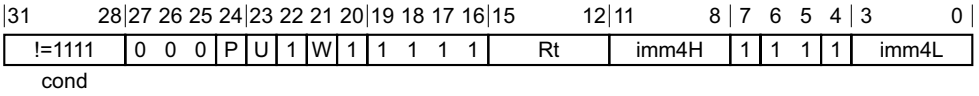
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.92 LDRSH (literal)

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *A1 variant*

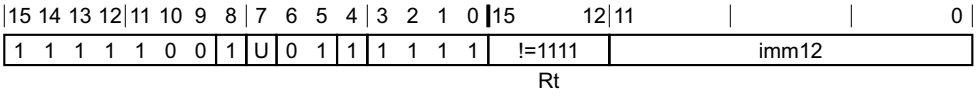
Applies when `!(P == 0 && W == 1)`.

```
LDRSH{<c>}{<q>} <Rt>, <label> // Normal form
LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form
```

#### *Decode for this encoding*

```
if P == '0' && W == '1' then SEE LDRSHT;
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDRSH{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

#### *Decode for this encoding*

```
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSH (literal)* on page K1-5450.

Related instructions: *Load literal* on page F3-2497.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<label>` For encoding A1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.

## Operation for all encodings

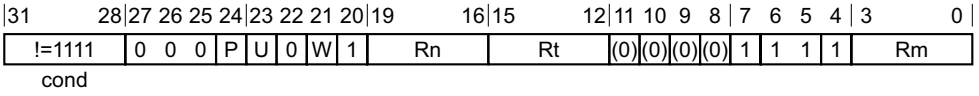
```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = SignExtend(data, 32);
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F6.1.93 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses on page F2-2452*.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

#### *Pre-indexed variant*

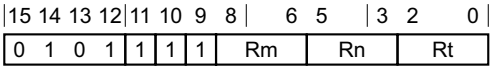
Applies when `P = 1 && W = 1`.

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE LDRSHT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

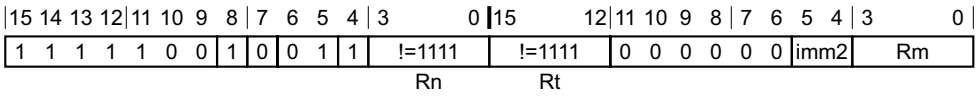
```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *T2 variant*

```
LDRSH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSH (register)* on page K1-5450.

Related instructions: *Load/store (register offset)* on page F3-2469.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.94 LDRSHT

Load Register Signed Halfword Unprivileged loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-2452.

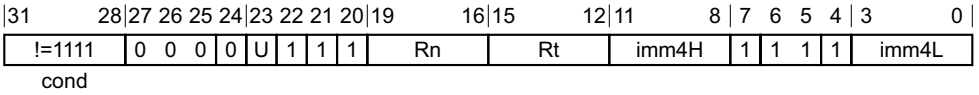
The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRSHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

### A1



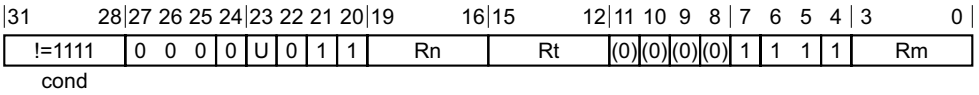

#### *A1 variant*

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

### A2



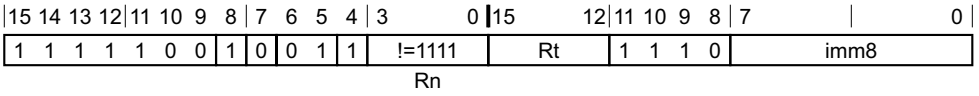

#### *A2 variant*

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for this encoding*

```
if Rn == '1111' then SEE LDRSH (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRSHT* on page K1-5450.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

## Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.95 LDRT

Load Register Unprivileged loads a word from memory, and writes it to a register. For information about memory accesses see *Memory accesses* on page F2-2452.

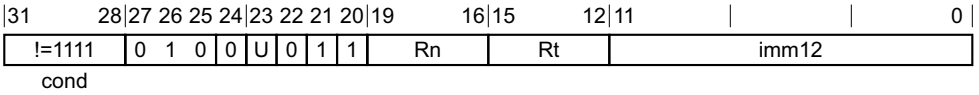
The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

### A1



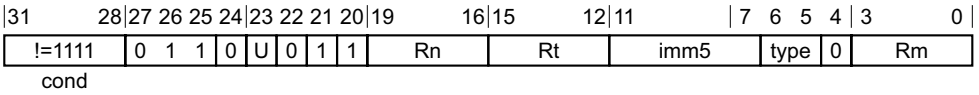

#### *A1 variant*

```
LDRT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

### A2



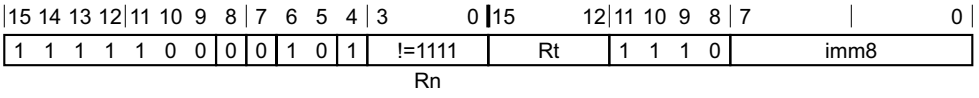

#### *A2 variant*

```
LDRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
LDRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for this encoding*

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *LDRT* on page K1-5451.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.<br>For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| +/- | For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1<br>For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <shift> | The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see . |
| + | Specifies the offset is added to the base register. |
| <imm> | For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.<br>For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,4];
    if postindex then R[n] = offset_addr;
    R[t] = data;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

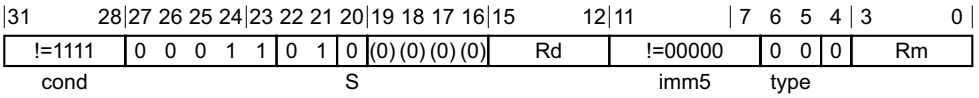
## F6.1.96 LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

### A1




#### *MOV, shift or rotate by value variant*

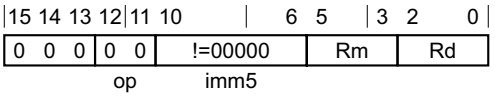
```
LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

### T2




#### *T2 variant*

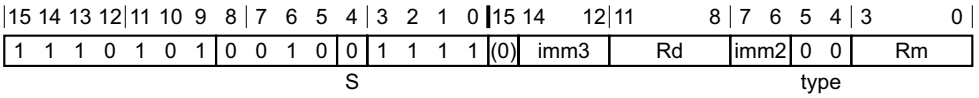
```
LSL<c>{<q>} {<Rd>,} <Rm>, #<imm> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when `InITBlock()`.

### T3




#### *MOV, shift or rotate by value variant*

```
LSL<c>.W {<Rd>,} <Rm>, #<imm> // Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

### Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

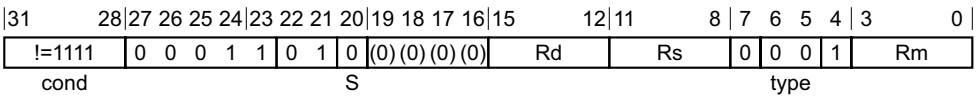
## F6.1.97 LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### A1




#### *Not flag setting variant*

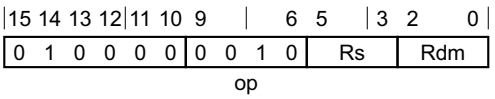
```
LSL{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

### T1




#### *Logical shift left variant*

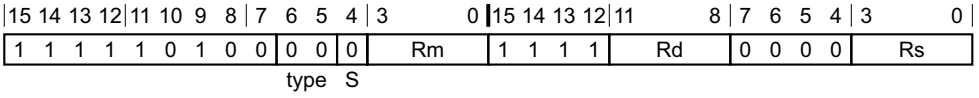
```
LSL<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

and is the preferred disassembly when `InITBlock()`.

### T2




#### *Not flag setting variant*

```
LSL<c>.W {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Rdm>` | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| `<Rd>` | Is the general-purpose destination register, encoded in the "Rd" field. |
| `<Rm>` | Is the first general-purpose source register, encoded in the "Rm" field. |
| `<Rs>` | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.98 LSLS (immediate)

Logical Shift Left, setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

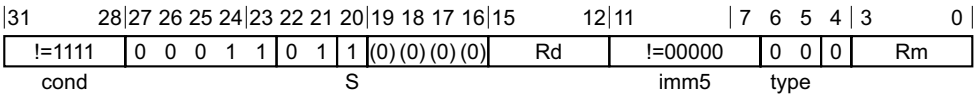
The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

### A1




#### *MOVS, shift or rotate by value variant*

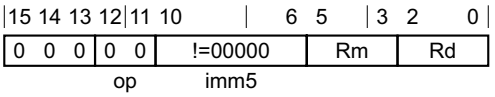
```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

### T2




#### *T2 variant*

```
LSLS{<q>} {<Rd>,} <Rm>, #<imm> // Outside IT block
```

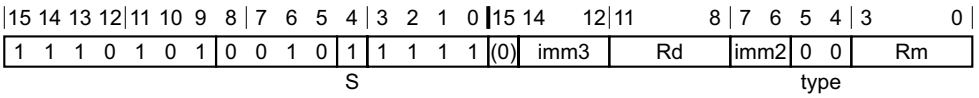
is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when `!InITBlock()`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T3




#### *MOVS, shift or rotate by value variant*

```
LSLS.W {<Rd>,} <Rm>, #<imm> // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

## Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

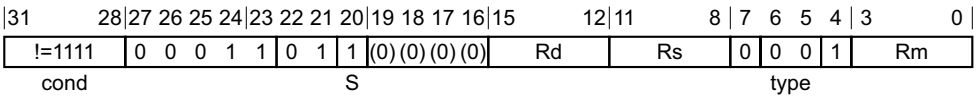
## F6.1.99 LSLS (register)

Logical Shift Left, setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### A1




#### *Flag setting variant*

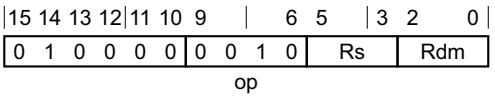
```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

### T1




#### *Logical shift left variant*

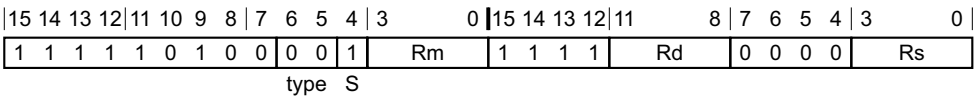
```
LSLS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

### T2




#### *Flag setting variant*

```
LSLS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Rdm>` | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| `<Rd>` | Is the general-purpose destination register, encoded in the "Rd" field. |
| `<Rm>` | Is the first general-purpose source register, encoded in the "Rm" field. |
| `<Rs>` | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

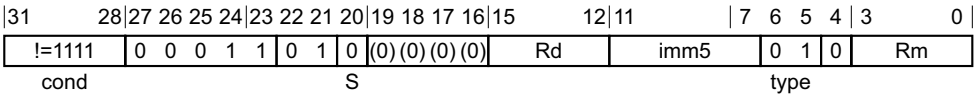
## F6.1.100 LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

### A1




#### *MOV, shift or rotate by value variant*

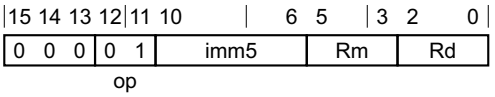
```
LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

### T2




#### *T2 variant*

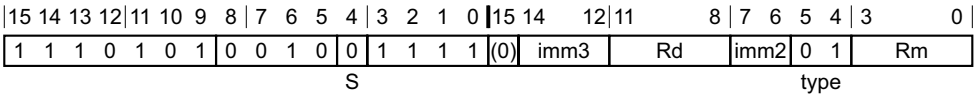
```
LSR<c>{<q>} {<Rd>,} <Rm>, #<imm> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is the preferred disassembly when `InITBlock()`.

### T3




#### *MOV, shift or rotate by value variant*

```
LSR<c>.W {<Rd>,} <Rm>, #<imm> // Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

```
LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

### Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

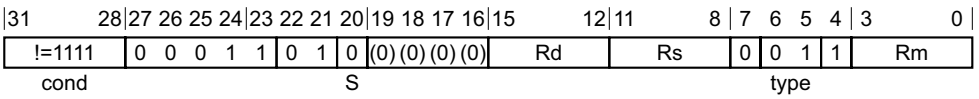
## F6.1.101 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### A1




#### *Not flag setting variant*

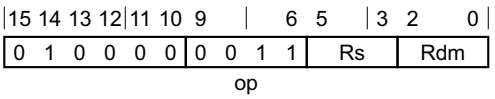
```
LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

### T1




#### *Logical shift right variant*

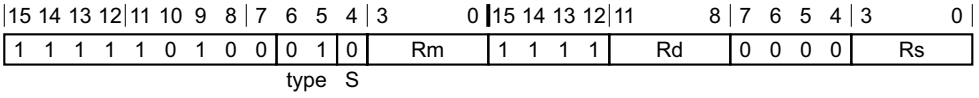
```
LSR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

### T2




#### *Not flag setting variant*

```
LSR<c>.W {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Rdm>` | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| `<Rd>` | Is the general-purpose destination register, encoded in the "Rd" field. |
| `<Rm>` | Is the first general-purpose source register, encoded in the "Rm" field. |
| `<Rs>` | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.102 LSRS (immediate)

Logical Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

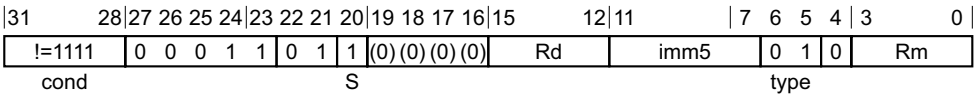
The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

### A1




#### *MOVS, shift or rotate by value variant*

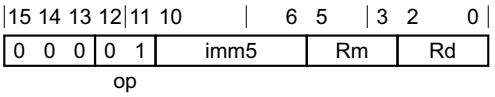
```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

### T2




#### *T2 variant*

```
LSRS{<q>} {<Rd>,} <Rm>, #<imm> // Outside IT block
```

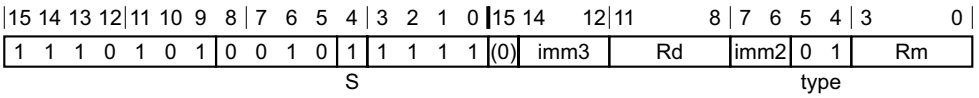
is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is the preferred disassembly when `!InITBlock()`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T3




#### *MOVS, shift or rotate by value variant*

```
LSRS.W {<Rd>,} <Rm>, #<imm> // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

## Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

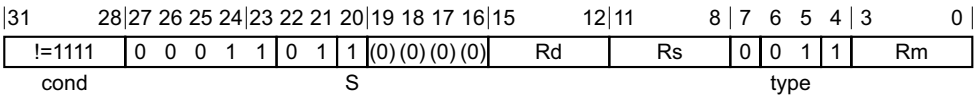
## F6.1.103 LSRS (register)

Logical Shift Right, setting flags (register) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### A1




#### *Flag setting variant*

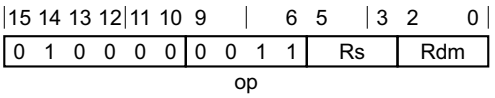
```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

### T1




#### *Logical shift right variant*

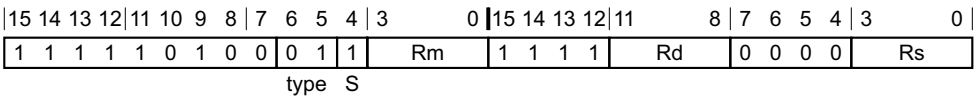
```
LSRS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

### T2




#### *Flag setting variant*

```
LSRS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

`LSRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>`

is equivalent to

`MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>`

and is always the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rdm>` Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the first general-purpose source register, encoded in the "Rm" field.

`<Rs>` Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.104 MCR, MCR2

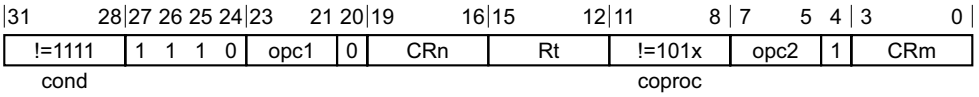
Move to Coprocessor from general-purpose register passes the value of a general-purpose register to a conceptual coprocessor.

This is a generic coprocessor instruction. The coproc field identifies the target conceptual coprocessor. This must be one of CP10, CP11, CP14, or CP15, and for these CP values, this manual defines the opc1, opc2, CRn, and CRd field values that are valid MCR and MCR2 instructions. Other encodings are UNDEFINED. For more information see *Conceptual coprocessor support* on page E1-2353 and *General behavior of System registers* on page G4-4118.

In an implementation that includes EL2, MCR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see .

Because of the range of possible traps to Hyp mode, the MCR pseudocode does not show these possible traps.

### A1



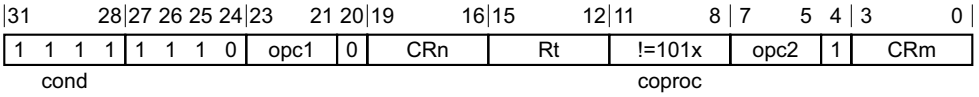

#### *A1 variant*

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

#### *Decode for this encoding*

```
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
t = UInt(Rt);  cp = UInt(coproc);
if t == 15  then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### A2



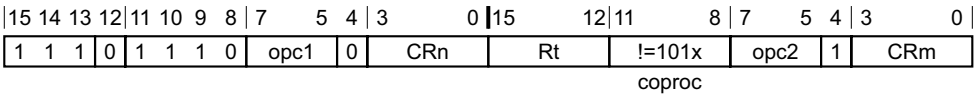

#### *A2 variant*

```
MCR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

#### *Decode for this encoding*

```
if coproc == '101x' then UNDEFINED;
t = UInt(Rt);  cp = UInt(coproc);
if t == 15  then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T1




#### *T1 variant*

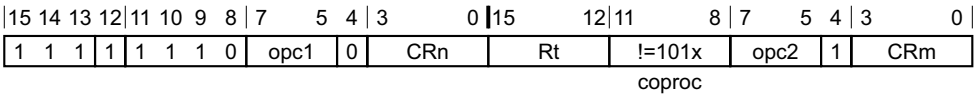
```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### Decode for this encoding

```
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
t = UInt(Rt);  cp = UInt(coproc);
if t == 15  then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T2




#### T2 variant

```
MCR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

#### Decode for this encoding

```
if coproc == '101x' then UNDEFINED;
t = UInt(Rt);  cp = UInt(coproc);
if t == 15  then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Advanced SIMD and floating-point: 8, 16, and 32-bit transfer between ARM core and extension registers.

## Assembler symbols

| | |
|---|---|
| <c> | For encoding A1, T1 and T2: see *Standard assembler syntax fields* on page F2-2446.<br>For encoding A2: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| <opc1> | Is a coprocessor-specific opcode in the range 0 to 7, encoded in the "opc1" field. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <CRn> | Is the first coprocessor register, encoded in the "CRn" field. |
| <CRm> | Is the second coprocessor register, encoded in the "CRm" field. |
| <opc2> | Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Coproc_CheckInstr(cp, ThisInstr());
    Coproc_SendOneWord(R[t], cp, ThisInstr());
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.105 MCRR, MCRR2

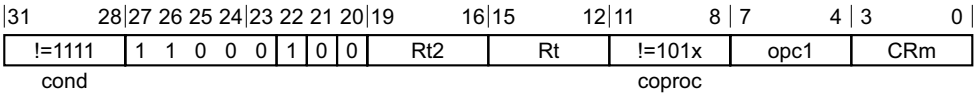
Move to Coprocessor from two general-purpose registers passes the values of two general-purpose registers to a conceptual coprocessor.

This is a generic coprocessor instruction. The coproc field identifies the target conceptual coprocessor. This must be one of CP10, CP11, CP14, or CP15, and for these CP values, this manual defines the opc1, and CRm field values that are valid MCRR and MCRR2 instructions. Other encodings are UNDEFINED. For more information see *Conceptual coprocessor support* on page E1-2353 and *General behavior of System registers* on page G4-4118.

In an implementation that includes EL2, MCRR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCRR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see .

Because of the range of possible traps to Hyp mode, the MCRR pseudocode does not show these possible traps.

### A1



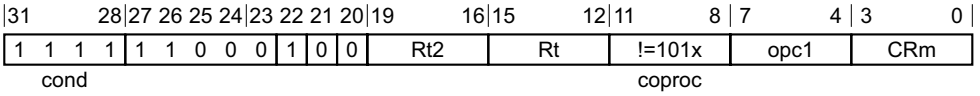

#### *A1 variant*

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

#### *Decode for this encoding*

```
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

### A2



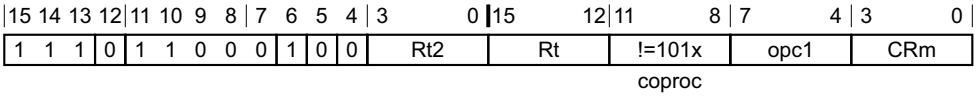

#### *A2 variant*

```
MCRR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

#### *Decode for this encoding*

```
if coproc == '101x' then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

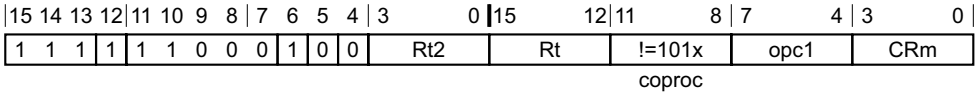
#### T1 variant

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

#### Decode for this encoding

```
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

### T2




#### T2 variant

```
MCRR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

#### Decode for this encoding

```
if coproc == '101x' then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Advanced SIMD and floating-point: 64-bit transfers between ARM core and extension registers.

## Assembler symbols

| | |
|---|---|
| <c> | For encoding A1, T1 and T2: see *Standard assembler syntax fields* on page F2-2446.<br>For encoding A2: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| <opc1> | Is a coprocessor-specific opcode in the range 0 to 15, encoded in the "opc1" field. |
| <Rt> | Is the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rt2> | Is the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <CRm> | Is a coprocessor register, encoded in the "CRm" field. |

The relative significance of <Rt2> and <Rt> is IMPLEMENTATION DEFINED for IMPLEMENTATION DEFINED uses of the MCRR and MRRC instructions. For the architected uses, as described in this manual, <Rt2> transfers bits[63:32] of the selected coprocessor register, while <Rt> transfers bits[31:0].


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Coproc_CheckInstr(cp, ThisInstr());
    Coproc_SendTwoWords(R[t2], R[t], cp, ThisInstr());
```

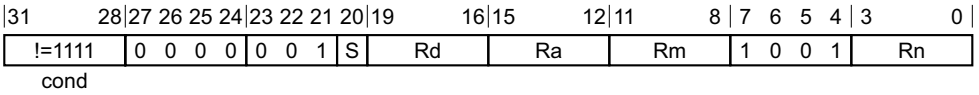

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.106 MLA, MLAS

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

In an A32 instruction, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
MLAS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Not flag setting variant*

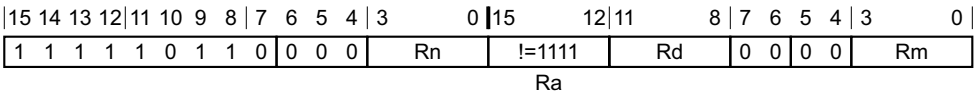
Applies when `S = 0`.

```
MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for this encoding*

```
if Ra == '1111' then SEE MUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = FALSE;
if d == 15 || n == 15 || m  == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

## Operation for all encodings

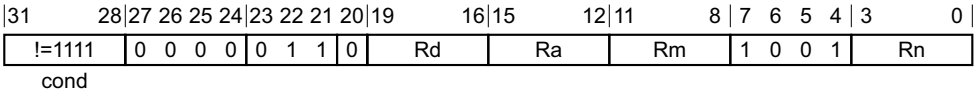
```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = operand1 * operand2 + addend;
    R[d] = result<31:0>;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result<31:0>);
        // PSTATE.C, PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.107 MLS

Multiply and Subtract multiplies two register values, and subtracts the product from a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

### A1



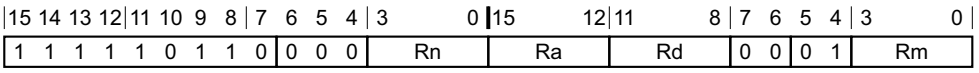

#### *A1 variant*

```
MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the minuend, encoded in the "Ra" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = addend - operand1 * operand2;
    R[d] = result<31:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.108 MOV, MOVS (immediate)

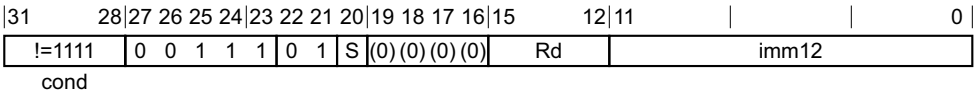
Move (immediate) writes an immediate value to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MOV variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *MOV variant*

Applies when S = 0.

```
MOV{<c>}{<q>} <Rd>, #<const>
```

##### *MOVS variant*

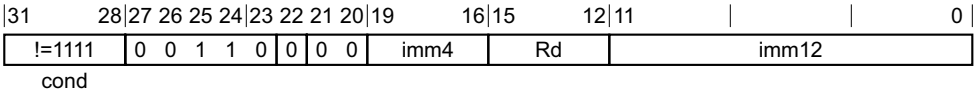
Applies when S = 1.

```
MOVS{<c>}{<q>} <Rd>, #<const>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

#### A2




##### *A2 variant*

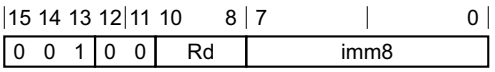
```
MOV{<c>}{<q>} <Rd>, #<imm16> // <imm16> can not be represented in A1
MOVW{<c>}{<q>} <Rd>, #<imm16> // <imm16> can be represented in A1
```

##### *Decode for this encoding*

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:imm12, 32);
if d == 15 then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



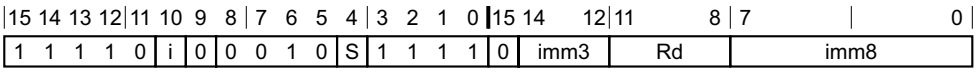

#### *T1 variant*

```
MOV<c>{<q>} <Rd>, #<imm8> // Inside IT block
MOVS{<q>} <Rd>, #<imm8> // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = PSTATE.C;
```

### T2




#### *MOV variant*

Applies when `S = 0`.

```
MOV<c>.W <Rd>, #<const> // Inside IT block, and <Rd>, <const> can be represented in T1
MOV{<c>}{<q>} <Rd>, #<const>
```

#### *MOVS variant*

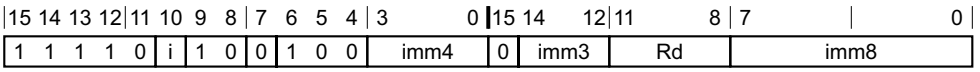
Applies when `S = 1`.

```
MOVS.W <Rd>, #<const> // Outside IT block, and <Rd>, <const> can be represented in T1
MOVS{<c>}{<q>} <Rd>, #<const>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T3




#### *T3 variant*

```
MOV{<c>}{<q>} <Rd>, #<imm16> // <imm16> cannot be represented in T1 or T2
MOVW{<c>}{<q>} <Rd>, #<imm16> // <imm16> can be represented in T1 or T2
```

#### *Decode for this encoding*

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the MOVS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding A2, T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<imm8> Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

<imm16> For encoding A2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.

For encoding T3: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    if d == 15 then          // Can only occur for encoding A1
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.109 MOV, MOVS (register)

Move (register) copies a value from a register to the destination register.

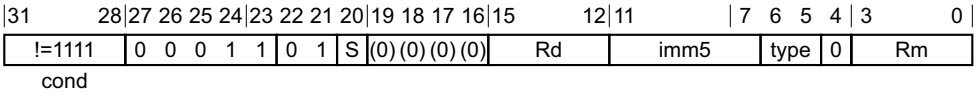
If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The MOV variant of the instruction is a branch. In the T32 instruction set (encoding T1) this is a simple branch, and in the A32 instruction set it is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is used by the aliases ASRS (immediate), ASR (immediate), LSLS (immediate), LSL (immediate), LSRS (immediate), LSR (immediate), RORS (immediate), ROR (immediate), RRXS, and RRX. See the *Alias conditions* on page F6-2816 table for details of when each alias is preferred.

### A1




#### *MOV, rotate right with extend variant*

Applies when S = 0 && imm5 = 00000 && type = 11.

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

#### *MOV, shift or rotate by value variant*

Applies when S = 0 && !(imm5 == 00000 && type == 11).

```
MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

#### *MOVS, rotate right with extend variant*

Applies when S = 1 && imm5 = 00000 && type = 11.

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

#### *MOVS, shift or rotate by value variant*

Applies when S = 1 && !(imm5 == 00000 && type == 11).

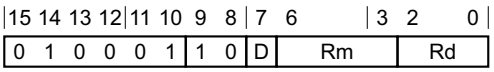
```
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



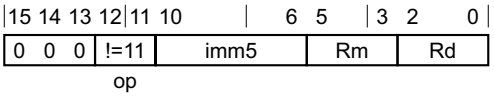

#### *T1 variant*

```
MOV{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T2



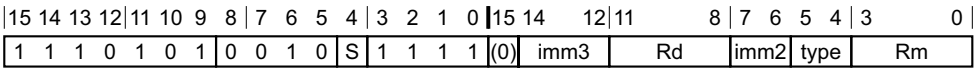

#### *T2 variant*

```
MOV<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>} // Inside IT block
MOVS{<q>} <Rd>, <Rm> {, <shift> #<amount>} // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
(shift_t, shift_n) = DecodeImmShift(op, imm5);
if op == '00' && imm5 == '00000' && InITBlock() then UNPREDICTABLE;
```

### T3




#### *MOV, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

#### *MOV, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
MOV<c>.W <Rd>, <Rm> {, <shift> #<amount>} // Inside IT block, and <Rd>, <Rm>, <shift>, <amount> can be
represented in T1 or T2
MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

#### *MOVS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && imm2 = 00 && type = 11`.

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *MOVS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
MOVS.W <Rd>, <Rm> {, <shift> #<amount>} // Outside IT block, and <Rd>, <Rm>, <shift>, <amount> can be
represented in T1 or T2
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

### *Decode for all variants of this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *MOV (register, T32)* on page K1-5451.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Alias conditions

| Alias | of variant | is preferred when |
|---|---|---|
| ASRS (immediate) | T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value) | `S == '1' && type == '10'` |
| ASRS (immediate) | T2 | `op == '10' && !InITBlock()` |
| ASR (immediate) | T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value) | `S == '0' && type == '10'` |
| ASR (immediate) | T2 | `op == '10' && InITBlock()` |
| LSLS (immediate) | T3 (MOVS, shift or rotate by value) | `S == '1' && imm3:Rd:imm2 != '000xxxx00' && type == '00'` |
| LSLS (immediate) | A1 (MOVS, shift or rotate by value) | `S == '1' && imm5 != '00000' && type == '00'` |
| LSLS (immediate) | T2 | `op == '00' && imm5 != '00000' && !InITBlock()` |
| LSL (immediate) | T3 (MOV, shift or rotate by value) | `S == '0' && imm3:Rd:imm2 != '000xxxx00' && type == '00'` |
| LSL (immediate) | A1 (MOV, shift or rotate by value) | `S == '0' && imm5 != '00000' && type == '00'` |
| LSL (immediate) | T2 | `op == '00' && imm5 != '00000' && InITBlock()` |
| LSRS (immediate) | T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value) | `S == '1' && type == '01'` |
| LSRS (immediate) | T2 | `op == '01' && !InITBlock()` |
| LSR (immediate) | T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value) | `S == '0' && type == '01'` |
| LSR (immediate) | T2 | `op == '01' && InITBlock()` |
| RORS (immediate) | T3 (MOVS, shift or rotate by value) | `S == '1' && imm3:Rd:imm2 != '000xxxx00' && type == '11'` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Alias | of variant | is preferred when |
|---|---|---|
| RORS (immediate) | A1 (MOVS, shift or rotate by value) | `S == '1' && imm5 != '00000' && type == '11'` |
| ROR (immediate) | T3 (MOV, shift or rotate by value) | `S == '0' && imm3:Rd:imm2 != '000xxxx00' && type == '11'` |
| ROR (immediate) | A1 (MOV, shift or rotate by value) | `S == '0' && imm5 != '00000' && type == '11'` |
| RRXS | T3 (MOVS, rotate right with extend) | `S == '1' && imm3 == '000' && imm2 == '00' && type == '11'` |
| RRXS | A1 (MOVS, rotate right with extend) | `S == '1' && imm5 == '00000' && type == '11'` |
| RRX | T3 (MOV, rotate right with extend) | `S == '0' && imm3 == '000' && imm2 == '00' && type == '11'` |
| RRX | A1 (MOV, rotate right with extend) | `S == '0' && imm5 == '00000' && type == '11'` |

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the MOVS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. ARM deprecates use of the instruction if <Rn> is not the LR, or if the optional shift or `RRX` argument is specified.

For encoding T1: is the general-purpose destination register, encoded in the "D:Rd" field. If the PC is used:

- The instruction causes a branch to the address moved to the PC. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The instruction must either be outside an IT block or the last instruction of an IT block.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<shift> For encoding A1 and T3: is the type of shift to be applied to the source register, encoded in the "type" field. It can have the following values:

LSL when type = 00

LSR when type = 01

ASR when type = 10

ROR when type = 11

For encoding T2: is the type of shift to be applied to the source register, encoded in the "op" field. It can have the following values:

LSL when op = 00

LSR when op = 01

ASR when op = 10

<amount> For encoding A1 and T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = shifted;
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

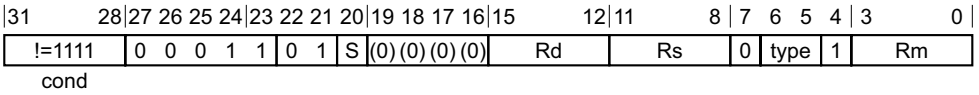

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.110 MOV, MOVS (register-shifted register)

Move (register-shifted register) copies a register-shifted register value to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases ASRS (register), ASR (register), LSLS (register), LSL (register), LSRS (register), LSR (register), RORS (register), and ROR (register). See the *Alias conditions* on page F6-2821 table for details of when each alias is preferred.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
MOVS{<c>}{<q>} <Rd>, <Rm>, <type> <Rs>
```

#### *Not flag setting variant*

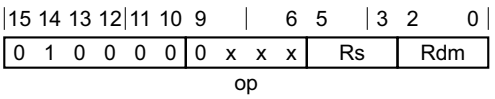
Applies when `S = 0`.

```
MOV{<c>}{<q>} <Rd>, <Rm>, <type> <Rs>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### T1




#### *Arithmetic shift right variant*

Applies when `op = 0100`.

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs> // Outside IT block
```

#### *Logical shift left variant*

Applies when `op = 0010`.

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs> // Outside IT block
```

#### *Logical shift right variant*

Applies when `op = 0011`.

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs> // Outside IT block
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Rotate right variant*

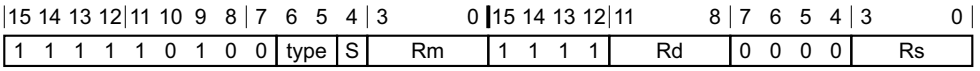
Applies when `op = 0111`.

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs> // Outside IT block
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rdn);  m = UInt(Rdn);  s = UInt(Rs);
setflags = !InITBlock();  shift_t = DecodeRegShift(op<2>:op<0>);
```

### T2




#### *Flag setting variant*

Applies when `S = 1`.

```
MOVS.W <Rd>, <Rm>, <type> <Rs> // Outside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
MOVS{<c>}{<q>} <Rd>, <Rm>, <type> <Rs>
```

#### *Not flag setting variant*

Applies when `S = 0`.

```
MOV<c>.W <Rd>, <Rm>, <type> <Rs> // Inside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in
T1
MOV{<c>}{<q>} <Rd>, <Rm>, <type> <Rs>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

Related encodings: In encoding T1, for a `op` field value that is not described above, see *Data-processing (two low registers)* on page F3-2467.

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Alias conditions

| Alias | of variant | is preferred when |
|---|---|---|
| ASRS (register) | A1 (flag setting) | `S == '1' && type == '10'` |
| ASRS (register) | T1 (arithmetic shift right) | `op == '0100' && !InITBlock()` |
| ASRS (register) | T2 (flag setting) | `type == '10' && S == '1'` |
| ASR (register) | A1 (not flag setting) | `S == '0' && type == '10'` |
| ASR (register) | T1 (arithmetic shift right) | `op == '0100' && InITBlock()` |
| ASR (register) | T2 (not flag setting) | `type == '10' && S == '0'` |
| LSLS (register) | A1 (flag setting) | `S == '1' && type == '00'` |
| LSLS (register) | T1 (logical shift left) | `op == '0010' && !InITBlock()` |
| LSLS (register) | T2 (flag setting) | `type == '00' && S == '1'` |
| LSL (register) | A1 (not flag setting) | `S == '0' && type == '00'` |
| LSL (register) | T1 (logical shift left) | `op == '0010' && InITBlock()` |
| LSL (register) | T2 (not flag setting) | `type == '00' && S == '0'` |
| LSRS (register) | A1 (flag setting) | `S == '1' && type == '01'` |
| LSRS (register) | T1 (logical shift right) | `op == '0011' && !InITBlock()` |
| LSRS (register) | T2 (flag setting) | `type == '01' && S == '1'` |
| LSR (register) | A1 (not flag setting) | `S == '0' && type == '01'` |
| LSR (register) | T1 (logical shift right) | `op == '0011' && InITBlock()` |
| LSR (register) | T2 (not flag setting) | `type == '01' && S == '0'` |
| RORS (register) | A1 (flag setting) | `S == '1' && type == '11'` |
| RORS (register) | T1 (rotate right) | `op == '0111' && !InITBlock()` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Alias | of variant | is preferred when |
|---|---|---|
| RORS (register) | T2 (flag setting) | `type == '11' && S == '1'` |
| ROR (register) | A1 (not flag setting) | `S == '0' && type == '11'` |
| ROR (register) | T1 (rotate right) | `op == '0111' && InITBlock()` |
| ROR (register) | T2 (not flag setting) | `type == '11' && S == '0'` |

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rdm>` Is the general-purpose source register and the destination register, encoded in the "Rdm" field.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

- `LSL` when `type = 00`
- `LSR` when `type = 01`
- `ASR` when `type = 10`
- `ROR` when `type = 11`

`<Rs>` Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

## Operation for all encodings

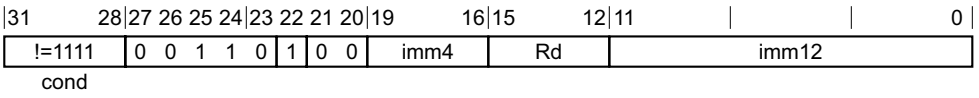
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (result, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.111 MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

### A1



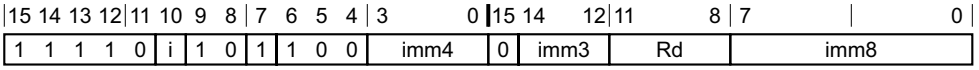

#### *A1 variant*

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  imm16 = imm4:imm12;
if d == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if d == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <imm16> | For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.<br>For encoding T1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.112 MRC, MRC2

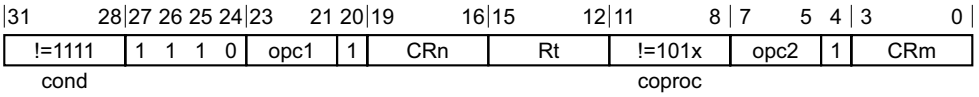
Move to general-purpose register from Coprocessor causes a conceptual coprocessor to transfer a value to a general-purpose register or to the condition flags.

This is a generic coprocessor instruction. The `coproc` field identifies the target conceptual coprocessor. This must be one of CP10, CP11, CP14, or CP15, and for these CP values, this manual defines the `opc1`, `opc2`, `CRn`, and `CRd` field values that are valid `MRC` and `MRC2` instructions. Other encodings are UNDEFINED. For more information see *Conceptual coprocessor support* on page E1-2353 and *General behavior of System registers* on page G4-4118.

In an implementation that includes EL2, `MRC` accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an `MRC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see .

Because of the range of possible traps to Hyp mode, the `MRC` pseudocode does not show these possible traps.

### A1



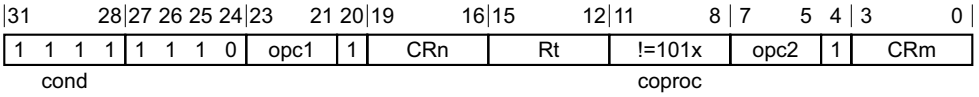

#### *A1 variant*

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

#### *Decode for this encoding*

```
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
t = UInt(Rt);  cp = UInt(coproc);
// ARMv8-A removes UNPREDICTABLE for R13
```

### A2



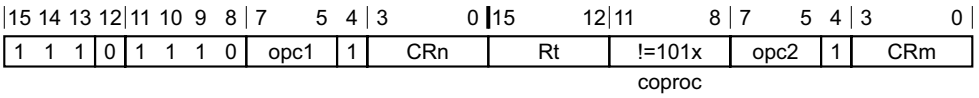

#### *A2 variant*

```
MRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

#### *Decode for this encoding*

```
if coproc == '101x' then UNDEFINED;
t = UInt(Rt);  cp = UInt(coproc);
// ARMv8-A removes UNPREDICTABLE for R13
```

### T1




#### *T1 variant*

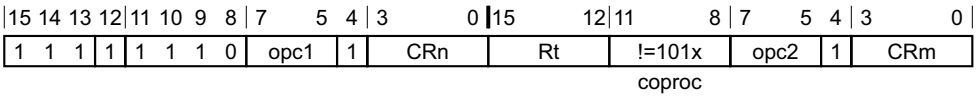
```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Decode for this encoding*

```
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
t = UInt(Rt);  cp = UInt(coproc);
// ARMv8-A removes UNPREDICTABLE for R13
```

### T2




#### *T2 variant*

```
MRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

#### *Decode for this encoding*

```
if coproc == '101x' then UNDEFINED;
t = UInt(Rt);  cp = UInt(coproc);
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Advanced SIMD and floating-point: 8, 16, and 32-bit transfer between ARM core and extension registers.

## Assembler symbols

| | |
|---|---|
| <c> | For encoding A1, T1 and T2: see *Standard assembler syntax fields* on page F2-2446.<br>For encoding A2: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| <opc1> | Is a coprocessor-specific opcode in the range 0 to 7, encoded in the "opc1" field. |
| <Rt> | Is the general-purpose register to be transferred or APSR_nzcv (encoded as 0b1111), encoded in the "Rt" field. If APSR_nzcv is used, bits [31:28] of the transferred value are written to the PSTATE condition flags. |
| <CRn> | Is the first coprocessor register, encoded in the "CRn" field. |
| <CRm> | Is the second coprocessor register, encoded in the "CRm" field. |
| <opc2> | Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Coproc_CheckInstr(cp, ThisInstr());
    value = Coproc_GetOneWord(cp, ThisInstr());
    if t != 15 then
        R[t] = value;
    elsif Coproc_CanWriteAPSR(cp, ThisInstr()) then
        PSTATE.<N,Z,C,V> = value<31:28>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // value<27:0> are not used.
else
    PSTATE.<N,Z,C,V> = bits(4) UNKNOWN;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.113 MRRC, MRRC2

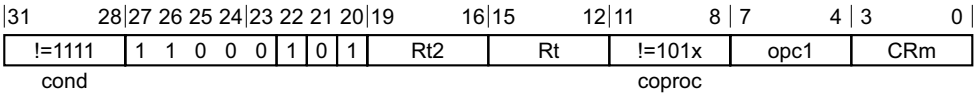
Move to two general-purpose registers from Coprocessor causes a conceptual coprocessor to transfer values to two general-purpose registers.

This is a generic coprocessor instruction. The `coproc` field identifies the target conceptual coprocessor. This must be one of CP10, CP11, CP14, or CP15, and for these CP values, this manual defines the `opc1`, and `CRm` field values that are valid `MRRC` and `MRRC2` instructions. Other encodings are UNDEFINED. For more information see *Conceptual coprocessor support* on page E1-2353 and *General behavior of System registers* on page G4-4118.

In an implementation that includes EL2, `MRRC` accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an `MRRC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see .

Because of the range of possible traps to Hyp mode, the `MRRC` pseudocode does not show these possible traps.

### A1



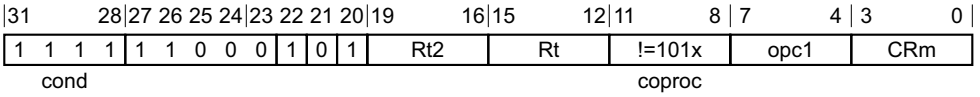

#### *A1 variant*

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

#### *Decode for this encoding*

```
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### A2



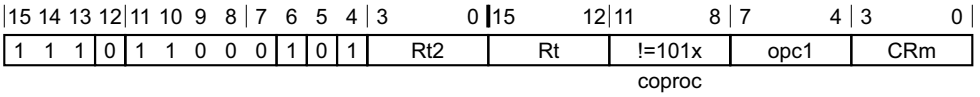

#### *A2 variant*

```
MRRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

#### *Decode for this encoding*

```
if coproc == '101x' then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

### T1




#### *T1 variant*

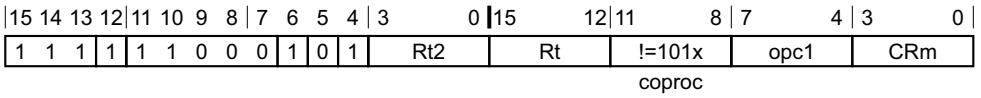
```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Decode for this encoding*

```
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T2




#### *T2 variant*

```
MRRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

#### *Decode for this encoding*

```
if coproc == '101x' then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

Advanced SIMD and floating-point: 64-bit transfers between ARM core and extension registers.

## Assembler symbols

| | |
|---|---|
| <c> | For encoding A1, T1 and T2: see *Standard assembler syntax fields* on page F2-2446.<br>For encoding A2: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| <opc1> | Is a coprocessor-specific opcode in the range 0 to 15, encoded in the "opc1" field. |
| <Rt> | Is the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rt2> | Is the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <CRm> | Is a coprocessor register, encoded in the "CRm" field. |

The relative significance of <Rt2> and {syntax{<Rt>} is IMPLEMENTATION DEFINED for IMPLEMENTATION DEFINED uses of the MCRR and MRRC instructions. For the architected uses, as described in this manual, <Rt2> transfers bits[63:32] of the selected coprocessor register, while <Rt> transfers bits[31:0].

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Coproc_CheckInstr(cp, ThisInstr());
    (R[t2], R[t]) = Coproc_GetTwoWords(cp, ThisInstr());
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.114 MRS

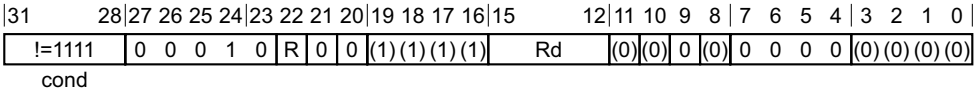
Move Special register to general-purpose register moves the value of the APSR, CPSR, or SPSR_<current_mode> into a general-purpose register.

ARM recommends the APSR form when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see APSR.

An MRS that accesses the SPSR is UNPREDICTABLE if executed in User mode or System mode.

An MRS that is executed in User mode and accesses the CPSR returns an UNKNOWN value for the CPSR.{E, A, I, F, M} fields.

#### A1



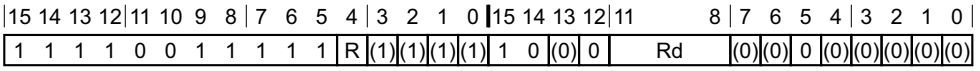

##### *A1 variant*

```
MRS{<c>}{<q>} <Rd>, <spec_reg>
```

##### *Decode for this encoding*

```
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
```

#### T1




##### *T1 variant*

```
MRS{<c>}{<q>} <Rd>, <spec_reg>
```

##### *Decode for this encoding*

```
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *MRS* on page K1-5481.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<spec_reg> Is the special register to be accessed, encoded in the "R" field. It can have the following values:

CPSR|APSR when R = 0

SPSR when R = 1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if read_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        else
            R[d] = SPSR[];
    else
        // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T bits masked out.
        psr_val = GetPSRFromPSTATE() AND '11111000 00001111 00000011 11011111';
        if PSTATE.EL == EL0 then
            // If accessed from User mode return UNKNOWN values for E, A, I, F bits, bits<9:6>,
            // and for the M field, bits<4:0>
            psr_val<9:6> = bits(4) UNKNOWN;
            psr_val<4:0> = bits(5) UNKNOWN;
        R[d] = psr_val;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

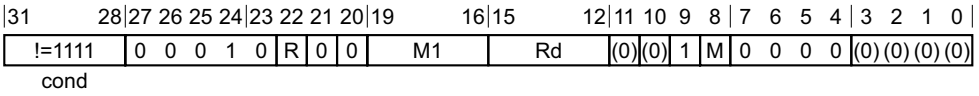
### F6.1.115 MRS (Banked register)

Move to Register from Banked or Special register moves the value from the Banked general-purpose register or SPSR of the specified mode, or the value of ELR_hyp, to a general-purpose register.

MRS (Banked register) is UNPREDICTABLE if executed in User mode.

The effect of using an MRS (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see *Usage restrictions on the Banked register transfer instructions* on page F6-3203.

#### A1



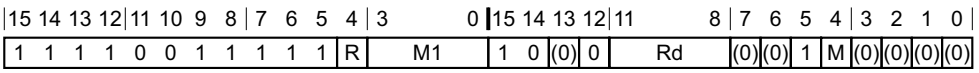

##### *A1 variant*

```
MRS{<c>}{<q>} <Rd>, <banked_reg>
```

##### *Decode for this encoding*

```
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
SYSm = M:M1;
```

#### T1




##### *T1 variant*

```
MRS{<c>}{<q>} <Rd>, <banked_reg>
```

##### *Decode for this encoding*

```
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
SYSm = M:M1;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<banked_reg> Is the name of the banked register to be transferred to or from, encoded in the "R:M:M1" field. It can have the following values:

| | |
|---|---|
| R8_usr | when R = 0, M = 0, M1 = 0000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| | |
|---|---|
| R9_usr | when R = 0, M = 0, M1 = 0001 |
| R10_usr | when R = 0, M = 0, M1 = 0010 |
| R11_usr | when R = 0, M = 0, M1 = 0011 |
| R12_usr | when R = 0, M = 0, M1 = 0100 |
| SP_usr | when R = 0, M = 0, M1 = 0101 |
| LR_usr | when R = 0, M = 0, M1 = 0110 |
| UNPREDICTABLE | when R = 0, M = 0, M1 = 0111 |
| R8_fiq | when R = 0, M = 0, M1 = 1000 |
| R9_fiq | when R = 0, M = 0, M1 = 1001 |
| R10_fiq | when R = 0, M = 0, M1 = 1010 |
| R11_fiq | when R = 0, M = 0, M1 = 1011 |
| R12_fiq | when R = 0, M = 0, M1 = 1100 |
| SP_fiq | when R = 0, M = 0, M1 = 1101 |
| LR_fiq | when R = 0, M = 0, M1 = 1110 |
| UNPREDICTABLE | when R = 0, M = 0, M1 = 1111 |
| LR_irq | when R = 0, M = 1, M1 = 0000 |
| SP_irq | when R = 0, M = 1, M1 = 0001 |
| LR_svc | when R = 0, M = 1, M1 = 0010 |
| SP_svc | when R = 0, M = 1, M1 = 0011 |
| LR_abt | when R = 0, M = 1, M1 = 0100 |
| SP_abt | when R = 0, M = 1, M1 = 0101 |
| LR_und | when R = 0, M = 1, M1 = 0110 |
| SP_und | when R = 0, M = 1, M1 = 0111 |
| UNPREDICTABLE | when R = 0, M = 1, M1 = 10xx |
| LR_mon | when R = 0, M = 1, M1 = 1100 |
| SP_mon | when R = 0, M = 1, M1 = 1101 |
| ELR_hyp | when R = 0, M = 1, M1 = 1110 |
| SP_hyp | when R = 0, M = 1, M1 = 1111 |
| UNPREDICTABLE | when R = 1, M = 0, M1 = 0xxx |
| UNPREDICTABLE | when R = 1, M = 0, M1 = 10xx |
| UNPREDICTABLE | when R = 1, M = 0, M1 = 110x |
| SPSR_fiq | when R = 1, M = 0, M1 = 1110 |
| UNPREDICTABLE | when R = 1, M = 0, M1 = 1111 |
| SPSR_irq | when R = 1, M = 1, M1 = 0000 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 0001 |
| SPSR_svc | when R = 1, M = 1, M1 = 0010 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 0011 |
| SPSR_abt | when R = 1, M = 1, M1 = 0100 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 0101 |
| SPSR_und | when R = 1, M = 1, M1 = 0110 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 0111 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 10xx |
| SPSR_mon | when R = 1, M = 1, M1 = 1100 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 1101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| | |
|---|---|
| SPSR_hyp | when R = 1, M = 1, M1 = 1110 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 1111 |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
       UNPREDICTABLE;
    else
        mode = PSTATE.M;
        if read_spsr then
            SPSRaccessValid(SYSm, mode);           // Check for UNPREDICTABLE cases
            case SYSm of
                when '01110'  R[d] = SPSR_fiq;
                when '10000'  R[d] = SPSR_irq;
                when '10010'  R[d] = SPSR_svc;
                when '10100'  R[d] = SPSR_abt;
                when '10110'  R[d] = SPSR_und;
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    R[d] = SPSR_mon;
                when '11110'  R[d] = SPSR_hyp;
        else
            BankedRegisterAccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
            case SYSm of
                when '00xxx'                       // Access the User mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    R[d] = Rmode[m,M32_User];
                when '01xxx'                       // Access the FIQ mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    R[d] = Rmode[m,M32_FIQ];
                when '1000x'                       // Access the IRQ mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_IRQ];
                when '1001x'                       // Access the Supervisor mode regsiters
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Svc];
                when '1010x'                       // Access the Abort mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Abort];
                when '1011x'                       // Access the Undefined mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Undef];
                when '1110x'                       // Access Monitor registers
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Monitor];
                when '11110'                       // Access ELR_hyp register
                    R[d] = ELR_hyp;
                when '11111'                       // Access SP_hyp register
                    R[d] = Rmode[13,M32_Hyp];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

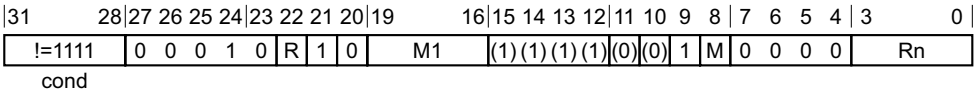
## F6.1.116 MSR (Banked register)

Move to Banked or Special register from general-purpose register moves the value of a general-purpose register to the Banked general-purpose register or SPSR of the specified mode, or to ELR_hyp.

`MSR` (Banked register) is UNPREDICTABLE if executed in User mode.

The effect of using an `MSR` (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see *Usage restrictions on the Banked register transfer instructions* on page F6-3203.

### A1



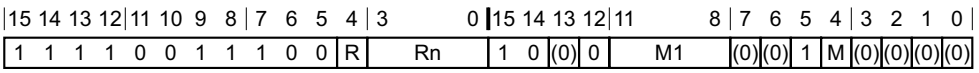

#### *A1 variant*

```
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

#### *Decode for this encoding*

```
n = UInt(Rn);  write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE;
SYSm = M:M1;
```

### T1




#### *T1 variant*

```
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

#### *Decode for this encoding*

```
n = UInt(Rn);  write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
SYSm = M:M1;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<banked_reg> Is the name of the banked register to be transferred to or from, encoded in the "R:M:M1" field. It can have the following values:

| | |
|---|---|
| `R8_usr` | when `R = 0, M = 0, M1 = 0000` |
| `R9_usr` | when `R = 0, M = 0, M1 = 0001` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| | |
|---|---|
| R10_usr | when R = 0, M = 0, M1 = 0010 |
| R11_usr | when R = 0, M = 0, M1 = 0011 |
| R12_usr | when R = 0, M = 0, M1 = 0100 |
| SP_usr | when R = 0, M = 0, M1 = 0101 |
| LR_usr | when R = 0, M = 0, M1 = 0110 |
| UNPREDICTABLE | when R = 0, M = 0, M1 = 0111 |
| R8_fiq | when R = 0, M = 0, M1 = 1000 |
| R9_fiq | when R = 0, M = 0, M1 = 1001 |
| R10_fiq | when R = 0, M = 0, M1 = 1010 |
| R11_fiq | when R = 0, M = 0, M1 = 1011 |
| R12_fiq | when R = 0, M = 0, M1 = 1100 |
| SP_fiq | when R = 0, M = 0, M1 = 1101 |
| LR_fiq | when R = 0, M = 0, M1 = 1110 |
| UNPREDICTABLE | when R = 0, M = 0, M1 = 1111 |
| LR_irq | when R = 0, M = 1, M1 = 0000 |
| SP_irq | when R = 0, M = 1, M1 = 0001 |
| LR_svc | when R = 0, M = 1, M1 = 0010 |
| SP_svc | when R = 0, M = 1, M1 = 0011 |
| LR_abt | when R = 0, M = 1, M1 = 0100 |
| SP_abt | when R = 0, M = 1, M1 = 0101 |
| LR_und | when R = 0, M = 1, M1 = 0110 |
| SP_und | when R = 0, M = 1, M1 = 0111 |
| UNPREDICTABLE | when R = 0, M = 1, M1 = 10xx |
| LR_mon | when R = 0, M = 1, M1 = 1100 |
| SP_mon | when R = 0, M = 1, M1 = 1101 |
| ELR_hyp | when R = 0, M = 1, M1 = 1110 |
| SP_hyp | when R = 0, M = 1, M1 = 1111 |
| UNPREDICTABLE | when R = 1, M = 0, M1 = 0xxx |
| UNPREDICTABLE | when R = 1, M = 0, M1 = 10xx |
| UNPREDICTABLE | when R = 1, M = 0, M1 = 110x |
| SPSR_fiq | when R = 1, M = 0, M1 = 1110 |
| UNPREDICTABLE | when R = 1, M = 0, M1 = 1111 |
| SPSR_irq | when R = 1, M = 1, M1 = 0000 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 0001 |
| SPSR_svc | when R = 1, M = 1, M1 = 0010 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 0011 |
| SPSR_abt | when R = 1, M = 1, M1 = 0100 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 0101 |
| SPSR_und | when R = 1, M = 1, M1 = 0110 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 0111 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 10xx |
| SPSR_mon | when R = 1, M = 1, M1 = 1100 |
| UNPREDICTABLE | when R = 1, M = 1, M1 = 1101 |
| SPSR_hyp | when R = 1, M = 1, M1 = 1110 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

UNPREDICTABLE when R = 1, M = 1, M1 = 1111

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        UNPREDICTABLE;
    else
        mode = PSTATE.M;
        if write_spsr then
            SPSRaccessValid(SYSm, mode);             // Check for UNPREDICTABLE cases
            case SYSm of
                when '01110'  SPSR_fiq = R[n];
                when '10000'  SPSR_irq = R[n];
                when '10010'  SPSR_svc = R[n];
                when '10100'  SPSR_abt = R[n];
                when '10110'  SPSR_und = R[n];
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    SPSR_mon = R[n];
                when '11110'  SPSR_hyp = R[n];
        else
            BankedRegisterAccessValid(SYSm, mode);  // Check for UNPREDICTABLE cases
            case SYSm of
                when '00xxx'                        // Access the User mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    Rmode[m,M32_User] = R[n];
                when '01xxx'                        // Access the FIQ mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    Rmode[m,M32_FIQ] = R[n];
                when '1000x'                        // Access the IRQ mode registers
                    m = 14 - UInt(SYSm<0>);         // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_IRQ] = R[n];
                when '1001x'                        // Access the Supervisor mode regsiters
                    m = 14 - UInt(SYSm<0>);         // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Svc] = R[n];
                when '1010x'                        // Access the Abort mode registers
                    m = 14 - UInt(SYSm<0>);         // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Abort] = R[n];
                when '1011x'                        // Access the Undefined mode registers
                    m = 14 - UInt(SYSm<0>);         // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Undef] = R[n];
                when '1110x'                        // Access Monitor registers
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    m = 14 - UInt(SYSm<0>);         // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Monitor] = R[n];
                when '11110'                        // Access ELR_hyp register
                    ELR_hyp = R[n];
                when '11111'                        // Access SP_hyp register
                    Rmode[13,M32_Hyp] = R[n];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.117 MSR (immediate)

Move immediate value to Special register moves selected bits of an immediate value to the corresponding bits in the APSR, CPSR, or SPSR_<current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, the immediate form of MSR is normally only useful at the Application level for writing to APSR_nzcvq (CPSR_f).

If an MSR (immediate) moves selected bits of an immediate value to the CPSR, the PE checks whether the value being written to PSTATE.M is legal. See *Illegal changes to PSTATE.M* on page G1-3771.

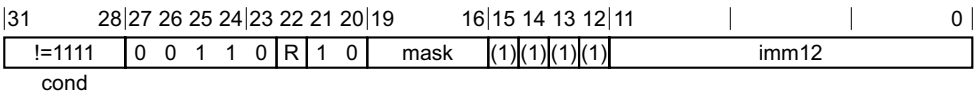
An MSR (immediate) executed in User mode:

- Is CONSTRAINED UNPREDICTABLE if it attempts to update the SPSR.
- Otherwise, does not update any CPSR field that is accessible only at EL1 or higher,

An MSR (immediate) executed in System mode is CONSTRAINED UNPREDICTABLE if it attempts to update the SPSR.

The CPSR.E bit is writable from any mode using an MSR instruction. ARM deprecates using this to change its value.

#### A1




##### *A1 variant*

Applies when !(R == 0 && mask == 0000).

```
MSR{<c>}{<q>} <spec_reg>, #<imm>
```

##### *Decode for this encoding*

```
if mask == '0000' && R == '0' then SEE "Related encodings";
imm32 = A32ExpandImm(imm12);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
```

#### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *MSR (immediate)* on page K1-5481.

Related encodings: *Move Special Register & Hints (immediate)* on page F4-2523.

#### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<spec_reg> Is one of:

- APSR_<bits>.
- CPSR_<fields>.
- SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

c mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.

x mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.

s mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`f` mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of `nzcvq`, `g`, or `nzcvqg`. These map to the following CPSR_<fields> values:

- `APSR_nzcvq` is the same as `CPSR_f` (mask== '1000').
- `APSR_g` is the same as `CPSR_s` (mask == '0100').
- `APSR_nzcvqg` is the same as `CPSR_fs` (mask == '1100').

ARM recommends the `APSR_<bits>` forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see APSR.

`<imm>` Is an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        else
            SPSRWriteByInstr(imm32, mask);
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(imm32, mask);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.118 MSR (register)

Move general-purpose register to Special register moves selected bits of a general-purpose register to the APSR, CPSR or SPSR_<current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, a read-modify-write sequence is normally required when the MSR instruction is being used at Application level and its destination is not APSR_nzcvq (CPSR_f).

If an MSR (register) moves selected bits of an immediate value to the CPSR, the PE checks whether the value being written to PSTATE.M is legal. See *Illegal changes to PSTATE.M* on page G1-3771.

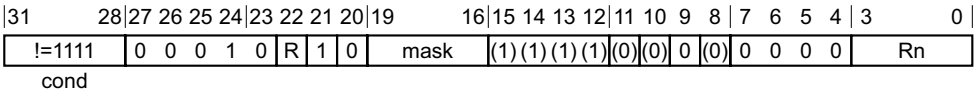
An MSR (register) executed in User mode:

- Is UNPREDICTABLE if it attempts to update the SPSR.
- Otherwise, does not update any CPSR field that is accessible only at EL1 or higher.

An MSR (register) executed in System mode is UNPREDICTABLE if it attempts to update the SPSR.

The CPSR.E bit is writable from any mode using an MSR instruction. ARM deprecates using this to change its value.

#### A1



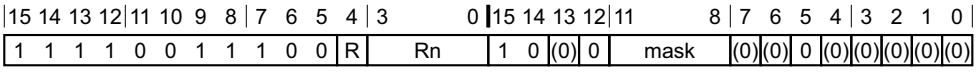

##### *A1 variant*

```
MSR{<c>}{<q>} <spec_reg>, <Rn>
```

##### *Decode for this encoding*

```
n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

#### T1




##### *T1 variant*

```
MSR{<c>}{<q>} <spec_reg>, <Rn>
```

##### *Decode for this encoding*

```
n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *MSR (register)* on page K1-5452.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<spec_reg> Is one of:

- APSR_<bits>.
- CPSR_<fields>.
- SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

c mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.

x mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.

s mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.

f mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of nzcvq, g, or nzcvqg. These map to the following CPSR_<fields> values:

- APSR_nzcvq is the same as CPSR_f (mask== '1000').
- APSR_g is the same as CPSR_s (mask == '0100').
- APSR_nzcvqg is the same as CPSR_fs (mask == '1100').

ARM recommends the APSR_<bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see APSR.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        else
            SPSRWriteByInstr(R[n], mask);
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(R[n], mask);
```

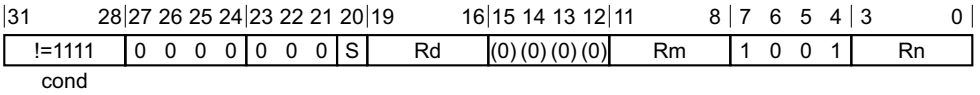

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.119 MUL, MULS

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

Optionally, it can update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
MULS{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

#### *Not flag setting variant*

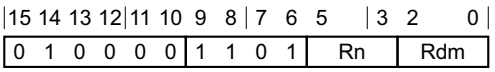
Applies when `S = 0`.

```
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1



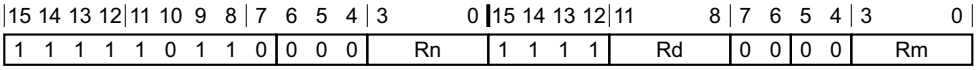

#### *T1 variant*

```
MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>} // Inside IT block
MULS{<q>} <Rdm>, <Rn>{, <Rdm>} // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
```

### T2




#### *T2 variant*

```
MUL<c>.W <Rd>, <Rn>{, <Rm>} // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rdm> | Is the second general-purpose source register holding the multiplier and the destination register, encoded in the "Rdm" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. If omitted, <Rd> is used. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result<31:0>);
        // PSTATE.C, PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.120 MVN, MVNS (immediate)

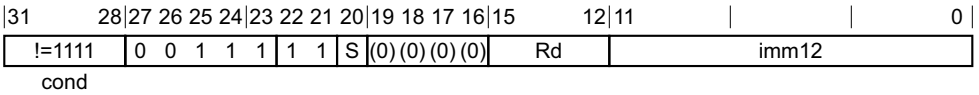
Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register.

If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1




#### *MVN variant*

Applies when S = 0.

```
MVN{<c>}{<q>} <Rd>, #<const>
```

#### *MVNS variant*

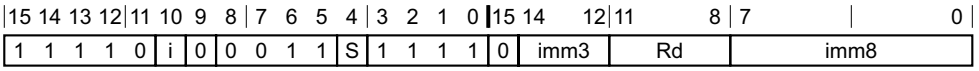
Applies when S = 1.

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

### T1




#### *MVN variant*

Applies when S = 0.

```
MVN{<c>}{<q>} <Rd>, #<const>
```

#### *MVNS variant*

Applies when S = 1.

```
MVNS{<c>}{<q>} <Rd>, #<const>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = NOT(imm32);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.121 MVN, MVNS (register)

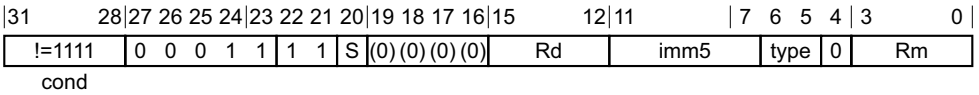
Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register.

If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *MVN, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

##### *MVN, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

##### *MVNS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

##### *MVNS, shift or rotate by value variant*

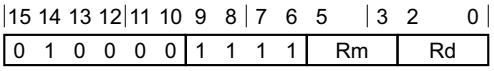
Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

```
MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

#### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

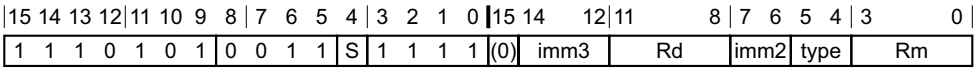
#### T1 variant

```
MVN<c>{<q>} <Rd>, <Rm> // Inside IT block
MVNS{<q>} <Rd>, <Rm> // Outside IT block
```

#### Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### MVN, rotate right with extend variant

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

#### MVN, shift or rotate by value variant

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
MVN<c>.W <Rd>, <Rm> // Inside IT block, and <Rd>, <Rm> can be represented in T1
MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

#### MVNS, rotate right with extend variant

Applies when `S = 1 && imm3 = 000 && imm2 = 00 && type = 11`.

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

#### MVNS, shift or rotate by value variant

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
MVNS.W <Rd>, <Rm> // Outside IT block, and <Rd>, <Rm> can be represented in T1
MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

#### Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T2: is the general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the source register, encoded in the "type" field. It can have the following values:

LSL when type = 00

LSR when type = 01

ASR when type = 10

ROR when type = 11

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

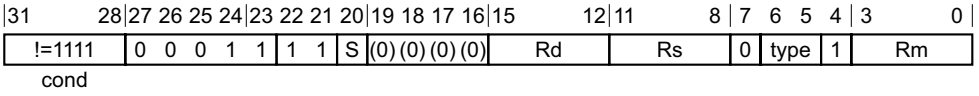
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.122 MVN, MVNS (register-shifted register)

Bitwise NOT (register-shifted register) writes the bitwise inverse of a register-shifted register value to the destination register. It can optionally update the condition flags based on the result.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
MVNS{<c>}{<q>} <Rd>, <Rm>, <type> <Rs>
```

#### *Not flag setting variant*

Applies when `S = 0`.

```
MVN{<c>}{<q>} <Rd>, <Rm>, <type> <Rs>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<Rs>` Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
result = NOT(shifted);
R[d] = result;
if setflags then
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

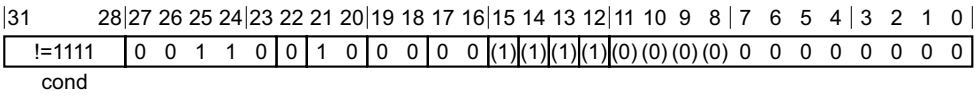
## F6.1.123 NOP

No Operation does nothing. This instruction can be used for instruction alignment purposes.

**Note**

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

### A1



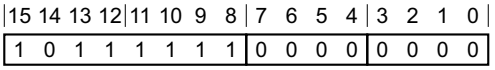

#### *A1 variant*

```
NOP{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T1



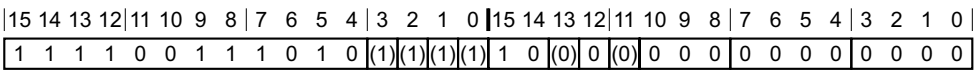

#### *T1 variant*

```
NOP{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T2




#### *T2 variant*

```
NOP{<c>}.W
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

### Operation for all encodings

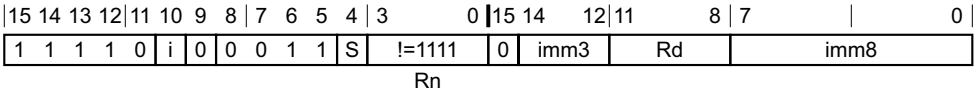
```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.124 ORN, ORNS (immediate)

Bitwise OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1




#### *Flag setting variant*

Applies when `S = 1`.

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *Not flag setting variant*

Applies when `S = 0`.

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE MVN (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | An immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values. |

### Operation

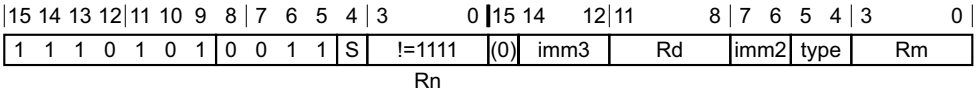
```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR NOT(imm32);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.125 ORN, ORNS (register)

Bitwise OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1




#### *ORN, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *ORN, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *ORNS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && imm2 = 00 && type = 11`.

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *ORNS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE MVN (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<shift> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

LSL when type = 00

LSR when type = 01

ASR when type = 10

ROR when type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.126 ORR, ORRS (immediate)

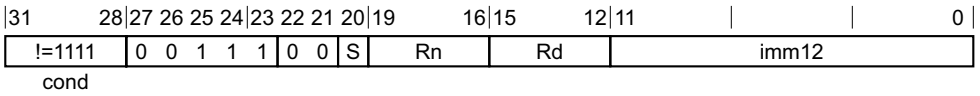
Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1




#### *ORR variant*

Applies when S = 0.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *ORRS variant*

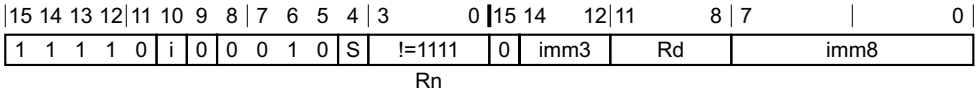
Applies when S = 1.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

### T1




#### *ORR variant*

Applies when S = 0.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *ORRS variant*

Applies when S = 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE MOV (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.127 ORR, ORRS (register)

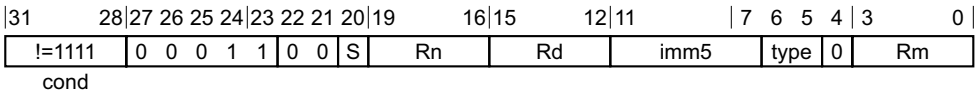
Bitwise OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1




#### *ORR, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *ORR, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *ORRS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *ORRS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

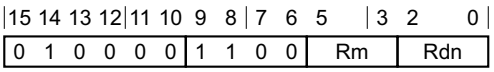
```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



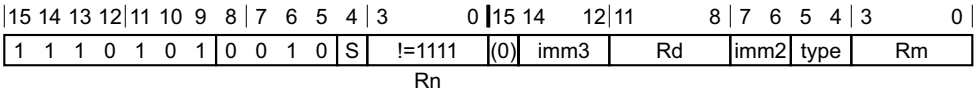

#### *T1 variant*

```
ORR<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
ORRS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *ORR, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *ORR, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
ORR<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *ORRS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && imm2 = 00 && type = 11`.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *ORRS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
ORRS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE "Related encodings";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related Encodings: *Data-processing (shifted register)* on page F3-2483


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| LSL | when type = 00 |
| LSR | when type = 01 |
| ASR | when type = 10 |
| ROR | when type = 11 |

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if ORRS <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORRS <Rd>, <Rn> had been written.
- Inside an IT block, if ORR<c> <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORR<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

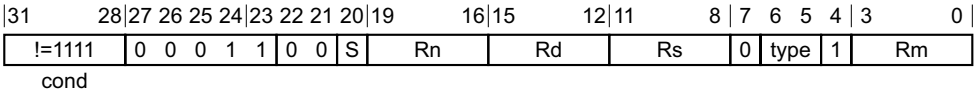
```
        ALUWritePC(result);
else
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.128 ORR, ORRS (register-shifted register)

Bitwise OR (register-shifted register) performs a bitwise (inclusive) OR of a register value and a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### A1




##### *Flag setting variant*

Applies when `S = 1`.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Not flag setting variant*

Applies when `S = 0`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

#### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

#### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<Rs>` Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

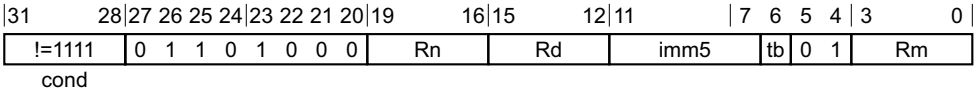
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.129 PKHBT, PKHTB

Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand.

### A1




#### *PKHBT variant*

Applies when `tb = 0`.

```
PKHBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, LSL #<imm>}
```

#### *PKHTB variant*

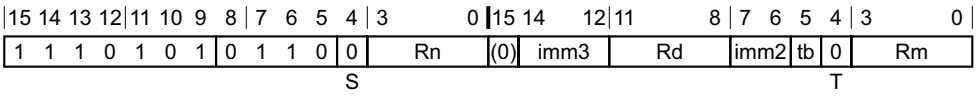
Applies when `tb = 1`.

```
PKHTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ASR #<imm>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm5);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *PKHBT variant*

Applies when `tb = 0`.

```
PKHBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, LSL #<imm>} // tbform == FALSE
```

#### *PKHTB variant*

Applies when `tb = 1`.

```
PKHTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ASR #<imm>} // tbform == TRUE
```

#### *Decode for all variants of this encoding*

```
if S == '1' || T == '1' then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1: the shift to apply to the value read from `<Rm>`, encoded in imm5. Is one of:

`omitted` No shift, encoded as `0b00000`.

`1-31` Left shift by specified number of bits, encoded as a binary number.

For encoding A1: the shift to apply to the value read from `<Rm>`, encoded in imm5. Is one of:

`omitted` Instruction is a pseudo-instruction and is assembled as though `PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn>` had been written.

`1-32` Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as `0b00000`. Other shift amounts are encoded as binary numbers.

**Note**

An assembler can permit `<imm>` = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

For encoding T1: the shift to apply to the value read from `<Rm>`, encoded in imm3:imm2. For `PKHBT`, it is one of:

`omitted` No shift, encoded as `0b00000`.

`1-31` Left shift by specified number of bits, encoded as a binary number.

For `PKHTB`, it is one of:

`omitted` Instruction is a pseudo-instruction and is assembled as though `PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn>` had been written.

`1-32` Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as `0b00000`. Other shift amounts are encoded as binary numbers.

**Note**

An assembler can permit `<imm>` = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = Shift(R[m], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    R[d]<15:0>  = if tbform then operand2<15:0> else R[n]<15:0>;
    R[d]<31:16> = if tbform then R[n]<31:16>    else operand2<31:16>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

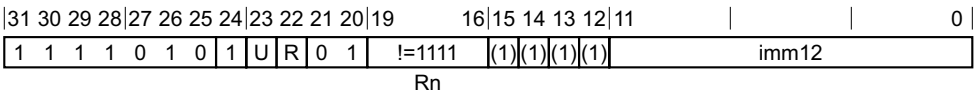
## F6.1.130 PLD, PLDW (immediate)

Preload Data (immediate) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.

The effect of a PLD or PLDW instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches* on page E2-2361.

### A1




#### *Preload read variant*

Applies when R = 1.

```
PLD{<c>}{<q>} [<Rn> {, #{+/-}<imm>}]
```

#### *Preload write variant*

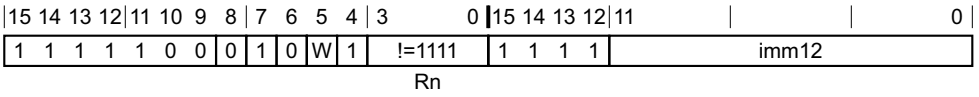
Applies when R = 0.

```
PLDW{<c>}{<q>} [<Rn> {, #{+/-}<imm>}]
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');  is_pldw = (R == '0');
```

### T1




#### *Preload read variant*

Applies when W = 0.

```
PLD{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

#### *Preload write variant*

Applies when W = 1.

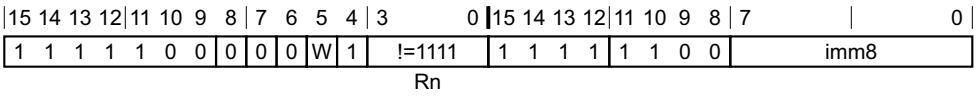
```
PLDW{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;  is_pldw = (W == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *Preload read variant*

Applies when `W = 0`.

```
PLD{<c>}{<q>} [<Rn> {, #-<imm>}]
```

#### *Preload write variant*

Applies when `W = 1`.

```
PLDW{<c>}{<q>} [<Rn> {, #-<imm>}]
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;  is_pldw = (W == '1');
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-2446. Must be `AL` or omitted.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see PLD (literal).

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

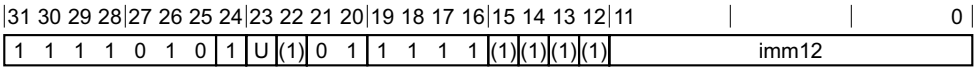

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.131 PLD (literal)

Preload Data (literal) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The effect of a PLD instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches* on page E2-2361.

#### A1



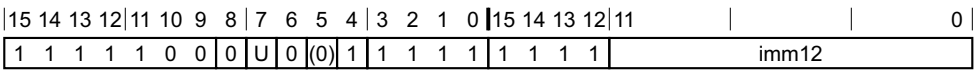

##### *A1 variant*

```
PLD{<c>}{<q>} <label> // Normal form
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative form
```

##### *Decode for this encoding*

```
imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

#### T1




##### *T1 variant*

```
PLD{<c>}{<q>} <label> // Preferred syntax
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative syntax
```

##### *Decode for this encoding*

```
imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-2446. Must be AL or omitted.
For encoding T1: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<label> The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range –4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint_PreloadData(address);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

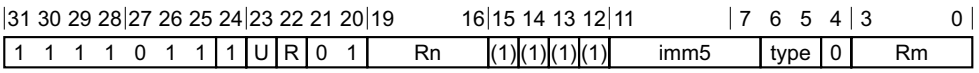
## F6.1.132 PLD, PLDW (register)

Preload Data (register) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.

The effect of a PLD or PLDW instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches* on page E2-2361.

### A1




#### *Preload read, optional shift or rotate variant*

Applies when `R = 1 && !(imm5 == 00000 && type == 11)`.

```
PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

#### *Preload read, rotate right with extend variant*

Applies when `R = 1 && imm5 = 00000 && type = 11`.

```
PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

#### *Preload write, optional shift or rotate variant*

Applies when `R = 0 && !(imm5 == 00000 && type == 11)`.

```
PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

#### *Preload write, rotate right with extend variant*

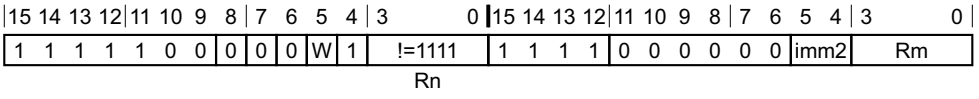
Applies when `R = 0 && imm5 = 00000 && type = 11`.

```
PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

#### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');  is_pldw = (R == '0');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 || (n == 15 && is_pldw) then UNPREDICTABLE;
```

### T1




#### *Preload read variant*

Applies when `W = 0`.

```
PLD{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

#### *Preload write variant*

Applies when `W = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
PLDW{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

### Decode for all variants of this encoding

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;  is_pldw = (W == '1');
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted.

For encoding T1: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0

+ when U = 1

\+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the index register, encoded in the "type" field. It can have the following values:

LSL when type = 00

LSR when type = 01

ASR when type = 10

ROR when type = 11

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

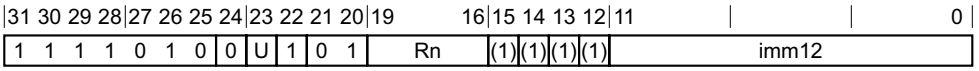

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.133 PLI (immediate, literal)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches* on page E2-2361.

#### A1



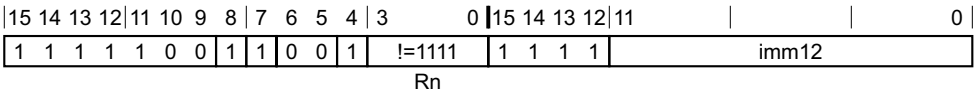

##### *A1 variant*

```
PLI{<c>}{<q>} [<Rn> {, #{+/-}<imm>}]
PLI{<c>}{<q>} <label> // Normal form
PLI{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative form
```

##### *Decode for this encoding*

```
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

#### T1



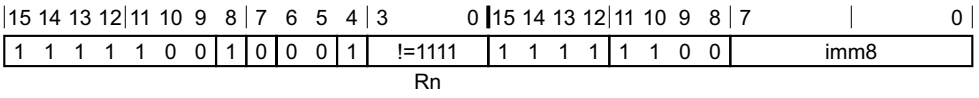

##### *T1 variant*

```
PLI{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

##### *Decode for this encoding*

```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;
```

#### T2




##### *T2 variant*

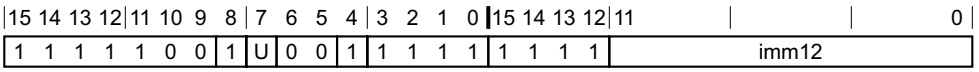
```
PLI{<c>}{<q>} [<Rn> {, #-<imm>}]
```

##### *Decode for this encoding*

```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T3




#### *T3 variant*

```
PLI{<c>}{<q>} <label> // Preferred syntax
PLI{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative syntax
```

#### *Decode for this encoding*

```
n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

`<c>` For encoding A1: see *Standard assembler syntax fields* on page F2-2446. Must be `AL` or omitted.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<label>` The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. The offset must be in the range –4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`. If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

For encoding T3: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For the literal forms of the instruction, encoding T3 is used, or Rn is encoded as `0b1111` in encoding A1, to indicate that the PC is the base register.

The alternative literal syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
```

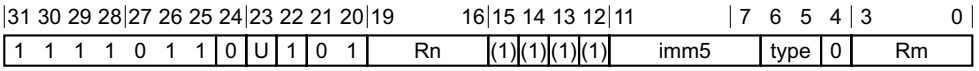

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.134 PLI (register)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches* on page E2-2361.

#### A1




##### *Rotate right with extend variant*

Applies when `imm5 = 00000 && type = 11`.

```
PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

##### *Shift or rotate by value variant*

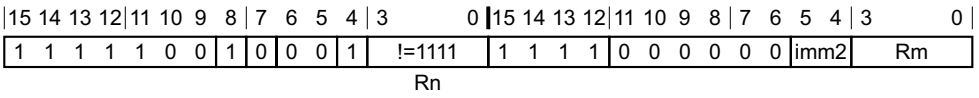
Applies when `!(imm5 == 00000 && type == 11)`.

```
PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

##### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
```

#### T1




##### *T1 variant*

```
PLI{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

##### *Decode for this encoding*

```
if Rn == '1111' then SEE PLI (immediate, literal);
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: see *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the index register, encoded in the "type" field. It can have the following values:

- `LSL` when `type = 00`
- `LSR` when `type = 01`
- `ASR` when `type = 10`
- `ROR` when `type = 11`

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadInstr(address);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

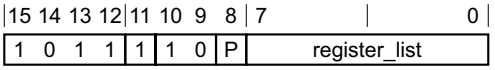
## F6.1.135 POP

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

### T1




#### *T1 variant*

```
POP{<c>}{<q>} <registers> // Preferred syntax
LDM{<c>}{<q>} SP!, <registers> // Alternate syntax
```

#### *Decode for this encoding*

```
registers = P:'0000000':register_list;   UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *POP (T32)* on page K1-5452 and .

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the PC. If the PC is in the list, the "P" field is set to 1, otherwise this field defaults to 0. If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block.

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = if UnalignedAllowed then MemU[address,4] else MemA[address,4];
            address = address + 4;
    if registers<15> == '1' then
        if UnalignedAllowed then
            if address<1:0> == '00' then
                LoadWritePC(MemU[address,4]);
            else
                UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        LoadWritePC(MemA[address,4]);
if registers<13> == '0' then SP = SP + 4*BitCount(registers);
if registers<13> == '1' then SP = bits(32) UNKNOWN;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

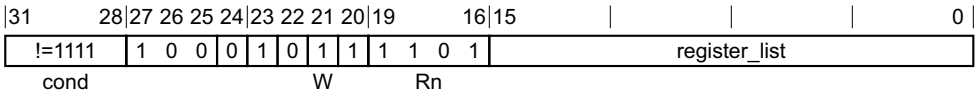
### F6.1.136 POP (multiple registers)

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data

This instruction is an alias of the LDM, LDMIA, LDMFD instruction. This means that:

- The encodings in this description are named to match the encodings of LDM, LDMIA, LDMFD.
- The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.

#### A1




##### *A1 variant*

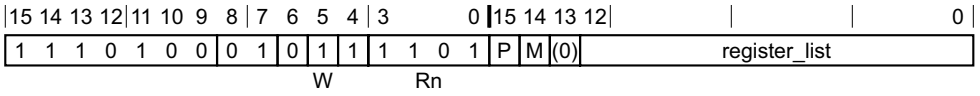
```
POP{<c>}{<q>} <registers>
```

is equivalent to

```
LDM{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

#### T2




##### *T2 variant*

```
POP{<c>}{<q>} <registers>
```

is equivalent to

```
LDM{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

#### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<registers>` For encoding A1: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453. If the SP is in the list, the value of the SP after such an instruction is UNKNOWN. The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318. ARM deprecates the use of this instruction with both the LR and the PC in the list.

For encoding T2: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of*


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*lists of general-purpose registers and the PC* on page F2-2453. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318. If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

### Operation for all encodings

The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

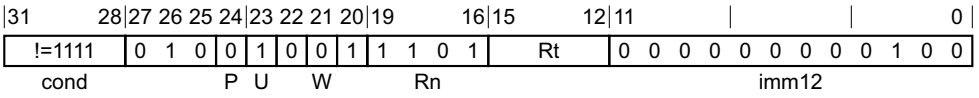
## F6.1.137 POP (single register)

Pop Single Register from Stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point just above the loaded data

This instruction is an alias of the LDR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of LDR (immediate).
- The description of LDR (immediate) gives the operational pseudocode for this instruction.

### A1




#### *Post-indexed variant*

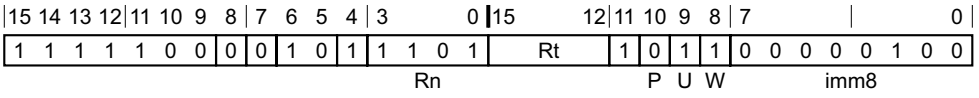
```
POP{<c>}{<q>} <single_register_list>
```

is equivalent to

```
LDR{<c>}{<q>} <Rt>, [SP], #4
```

and is always the preferred disassembly.

### T4




#### *Post-indexed variant*

```
POP{<c>}{<q>} <single_register_list>
```

is equivalent to

```
LDR{<c>}{<q>} <Rt>, [SP], #4
```

and is always the preferred disassembly.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<single_register_list>

Is the general-purpose register <Rt> to be loaded surrounded by { and }.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

### Operation for all encodings

The description of LDR (immediate) gives the operational pseudocode for this instruction.

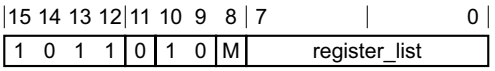

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.138 PUSH

Push Multiple Registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

### T1




#### *T1 variant*

```
PUSH{<c>}{<q>} <registers> // Preferred syntax
STMDB{<c>}{<q>} SP!, <registers> // Alternate syntax
```

#### *Decode for this encoding*

```
registers = '0':M:'000000':register_list;  UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *PUSH* on page K1-5453.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<registers> Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0.

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == 13 && i != LowestSetBit(registers) then  // Only possible for encoding A1
                MemA[address,4] = bits(32) UNKNOWN;
            else
                if UnalignedAllowed then
                    MemU[address,4] = R[i];
                else
                    MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1 or A2
        if UnalignedAllowed then
            MemU[address,4] = PCStoreValue();
        else
            MemA[address,4] = PCStoreValue();
    SP = SP - 4*BitCount(registers);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

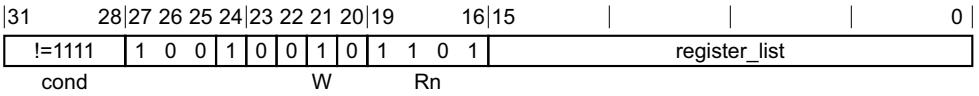
### F6.1.139 PUSH (multiple registers)

Push multiple registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data

This instruction is an alias of the STMDB, STMFD instruction. This means that:

- The encodings in this description are named to match the encodings of STMDB, STMFD.
- The description of STMDB, STMFD gives the operational pseudocode for this instruction.

#### A1




##### *A1 variant*

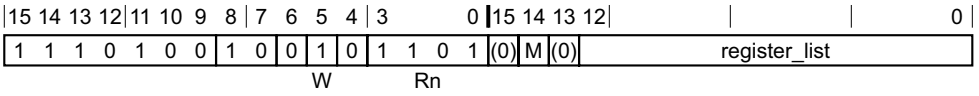
`PUSH{<c>}{<q>} <registers>`

is equivalent to

`STMDB{<c>}{<q>} SP!, <registers>`

and is the preferred disassembly when `BitCount(register_list) > 1`.

#### T1




##### *T1 variant*

`PUSH{<c>}{<q>} <registers>`

is equivalent to

`STMDB{<c>}{<q>} SP!, <registers>`

and is the preferred disassembly when `BitCount(register_list) > 1`.

#### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<registers>` For encoding A1: is a list of two or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453. The SP and PC can be in the list. However:

- ARM deprecates the use of instructions that include the PC in the list.
- If the SP is in the list, and it is not the lowest-numbered register in the list, the instruction stores an UNKNOWN value for the SP.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

### Operation for all encodings

The description of STMDB, STMFD gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

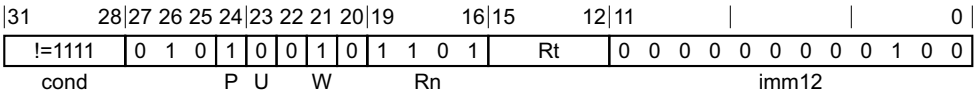
### F6.1.140 PUSH (single register)

Push Single Register to Stack stores a single general-purpose register to the stack, storing to the 32-bit word below the address in SP, and updates SP to point to the start of the stored data

This instruction is an alias of the STR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of STR (immediate).
- The description of STR (immediate) gives the operational pseudocode for this instruction.

#### A1




##### *Pre-indexed variant*

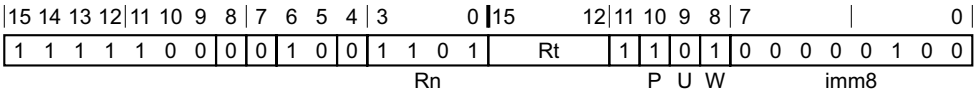
```
PUSH{<c>}{<q>} <single_register_list>
```

is equivalent to

```
STR{<c>}{<q>} <Rt>, [SP, #4]!
```

and is always the preferred disassembly.

#### T4




##### *Pre-indexed variant*

```
PUSH{<c>}{<q>} <single_register_list> // Standard syntax
```

is equivalent to

```
STR{<c>}{<q>} <Rt>, [SP, #-4]!
```

and is always the preferred disassembly.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<single_register_list>

Is the general-purpose register <Rt> to be stored surrounded by { and }.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

### Operation for all encodings

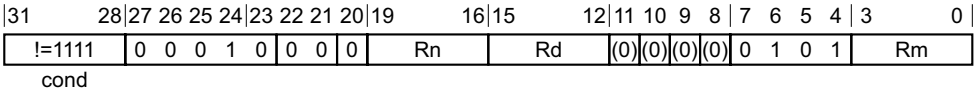
The description of STR (immediate) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.141 QADD

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range $-2^{31}$ to $(2^{31} - 1)$, and writes the result to the destination register. If saturation occurs, it sets PSTATE.Q to 1.

### A1



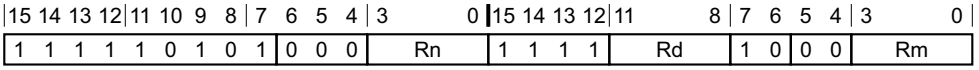

#### *A1 variant*

```
QADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
QADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rn> | Is the second general-purpose source register, encoded in the "Rn" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

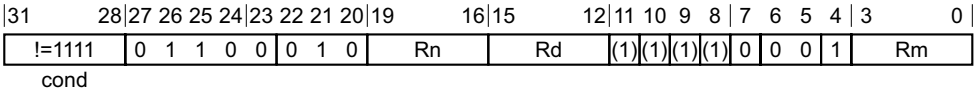
```
[n]), 32);
     if sat then
         PSTATE.Q = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.142 QADD16

Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

### A1



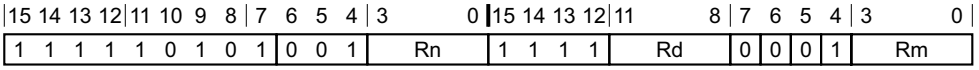

#### *A1 variant*

```
QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

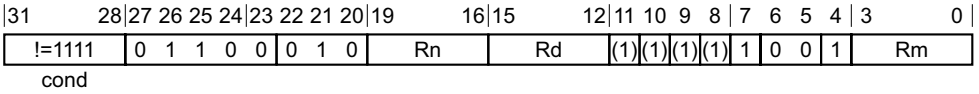
```
sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
R[d]<15:0>  = SignedSat(sum1, 16);
R[d]<31:16> = SignedSat(sum2, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.143 QADD8

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range $-2^7 <= x <= 2^7 - 1$, and writes the results to the destination register.

### A1



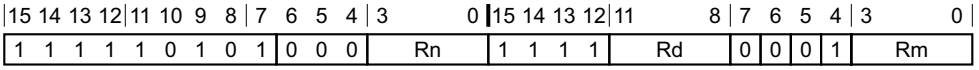

#### *A1 variant*

```
QADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
QADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

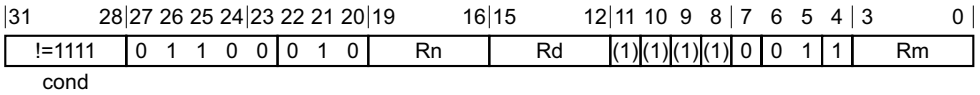
```
sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
R[d]<7:0>   = SignedSat(sum1, 8);
R[d]<15:8>  = SignedSat(sum2, 8);
R[d]<23:16> = SignedSat(sum3, 8);
R[d]<31:24> = SignedSat(sum4, 8);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.144 QASX

Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

### A1



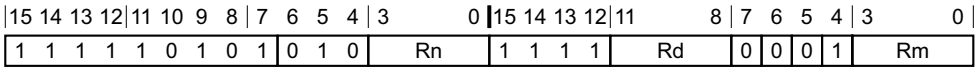

#### *A1 variant*

```
QASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
QASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

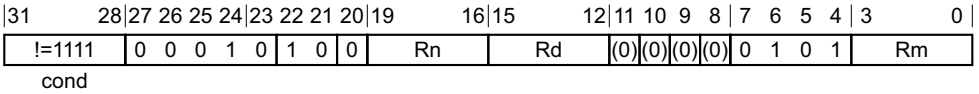
```
sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
R[d]<15:0>  = SignedSat(diff, 16);
R[d]<31:16> = SignedSat(sum, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.145 QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$. If saturation occurs in either operation, it sets PSTATE.Q to 1.

### A1



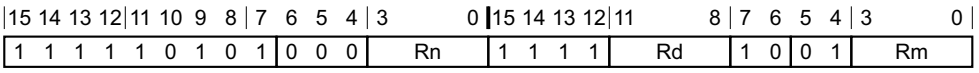

#### *A1 variant*

```
QDADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
QDADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rn> | Is the second general-purpose source register, encoded in the "Rn" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

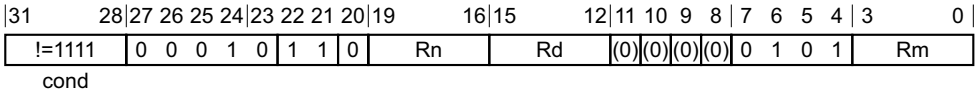
```
(R[d], sat2)  = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
if sat1 || sat2 then
    PSTATE.Q = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.146 QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$. If saturation occurs in either operation, it sets PSTATE.Q to 1.

### A1



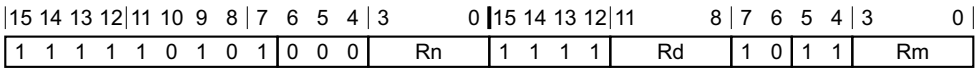

#### *A1 variant*

```
QDSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
QDSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rn> | Is the second general-purpose source register, encoded in the "Rn" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

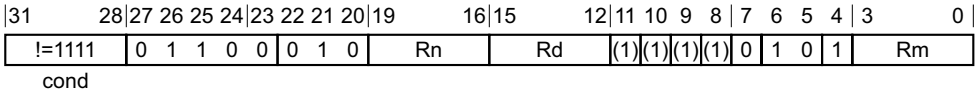
```
(R[d], sat2)  = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
if sat1 || sat2 then
    PSTATE.Q = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.147 QSAX

Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

### A1



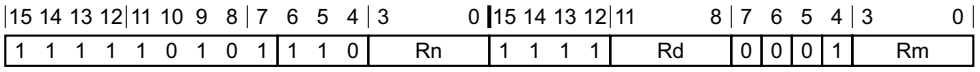

#### *A1 variant*

```
QSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
QSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

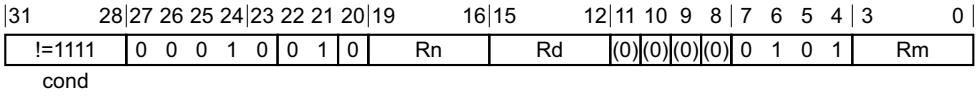
```
diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
R[d]<15:0>  = SignedSat(sum, 16);
R[d]<31:16> = SignedSat(diff, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.148 QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$, and writes the result to the destination register. If saturation occurs, it sets PSTATE.Q to 1.

### A1



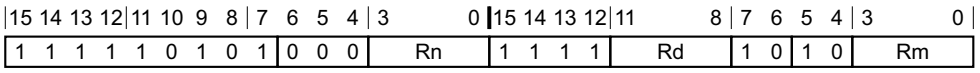

#### *A1 variant*

```
QSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
QSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rn> | Is the second general-purpose source register, encoded in the "Rn" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

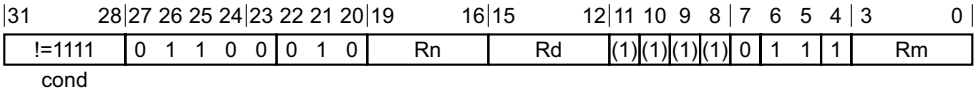
```
[n]), 32);
     if sat then
         PSTATE.Q = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.149 QSUB16

Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

### A1



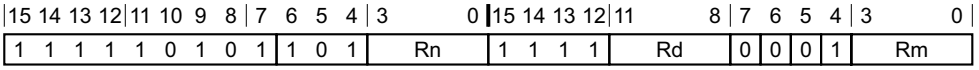

#### *A1 variant*

```
QSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
QSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

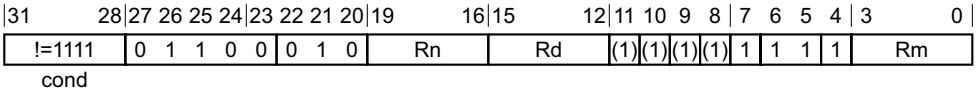
```
diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
R[d]<15:0>  = SignedSat(diff1, 16);
R[d]<31:16> = SignedSat(diff2, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.150 QSUB8

Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range $-2^7 <= x <= 2^7 - 1$, and writes the results to the destination register.

### A1



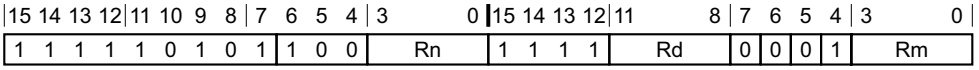

#### *A1 variant*

```
QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

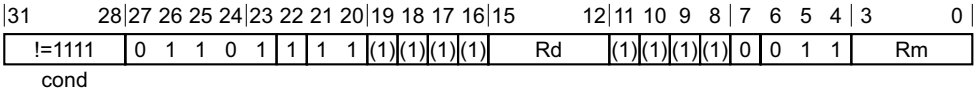
```
diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
R[d]<7:0>   = SignedSat(diff1, 8);
R[d]<15:8>  = SignedSat(diff2, 8);
R[d]<23:16> = SignedSat(diff3, 8);
R[d]<31:24> = SignedSat(diff4, 8);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.151 RBIT

Reverse Bits reverses the bit order in a 32-bit register.

### A1



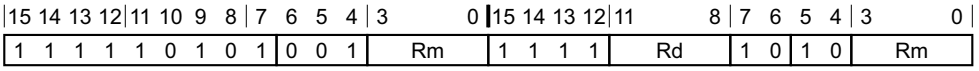

#### *A1 variant*

```
RBIT{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
RBIT{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *RBIT* on page K1-5454.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | For encoding A1: is the general-purpose source register, encoded in the "Rm" field.<br>For encoding T1: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

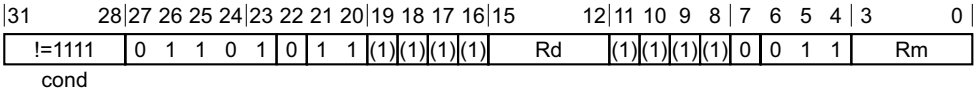
```
for i = 0 to 31
    result<31-i> = R[m]<i>;
R[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.152 REV

Byte-Reverse Word reverses the byte order in a 32-bit register.

### A1



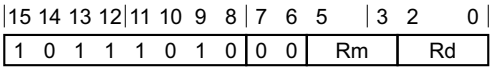

#### *A1 variant*

```
REV{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1



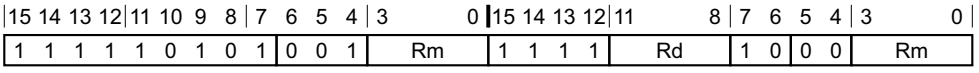

#### *T1 variant*

```
REV{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);
```

### T2




#### *T2 variant*

```
REV{<c>}.W <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1
REV{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *REV* on page K1-5455.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice.

### Operation for all encodings

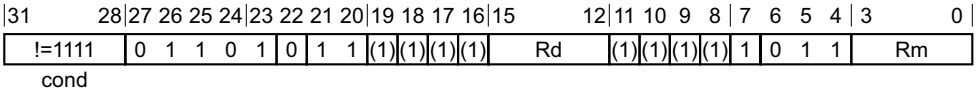
```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.153 REV16

Byte-Reverse Packed Halfword reverses the byte order in each16-bit halfword of a 32-bit register.

### A1



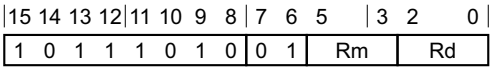

#### *A1 variant*

```
REV16{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1



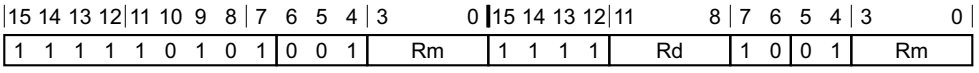

#### *T1 variant*

```
REV16{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);
```

### T2




#### *T2 variant*

```
REV16{<c>}.W <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1
REV16{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *REV16* on page K1-5455.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice.

### Operation for all encodings

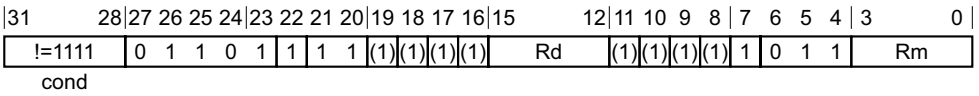
```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.154 REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign-extends the result to 32 bits.

### A1



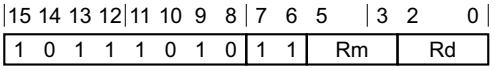

#### *A1 variant*

```
REVSH{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1



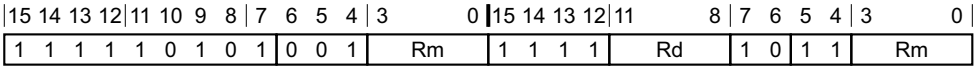

#### *T1 variant*

```
REVSH{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);
```

### T2




#### *T2 variant*

```
REVSH{<c>}.W <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1
REVSH{<c>}{<q>} <Rd>, <Rm>
```

#### *Decode for this encoding*

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *REVSH* on page K1-5455.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:8>  = SignExtend(R[m]<7:0>, 24);
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.155 RFE, RFEDA, RFEDB, RFEIA, RFEIB

Return From Exception loads two consecutive memory locations using an address in a base register:

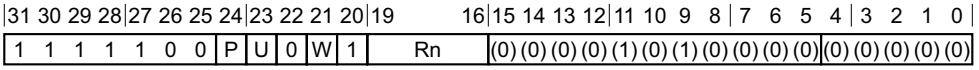
- The word loaded from the lower address is treated as an instruction address. The PE branches to it.
- The word loaded from the higher address is used to restore PSTATE. This word must be in the format of an SPSR.

An address adjusted by the size of the data loaded can optionally be written back to the base register.

The PE checks the value of the word loaded from the higher address for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.

RFE is UNDEFINED in Hyp mode and CONSTRAINED UNPREDICTABLE in User mode.

### A1




#### *Decrement After variant*

Applies when `P = 0 && U = 0`.

```
RFEDA{<c>}{<q>} <Rn>{!}
```

#### *Decrement Before variant*

Applies when `P = 1 && U = 0`.

```
RFEDB{<c>}{<q>} <Rn>{!}
```

#### *Increment After variant*

Applies when `P = 0 && U = 1`.

```
RFE{IA}{<c>}{<q>} <Rn>{!}
```

#### *Increment Before variant*

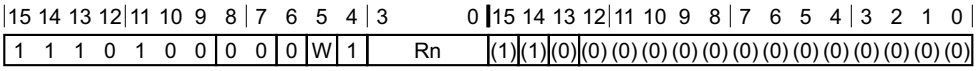
Applies when `P = 1 && U = 1`.

```
RFEIB{<c>}{<q>} <Rn>{!}
```

#### *Decode for all variants of this encoding*

```
n = UInt(Rn);
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

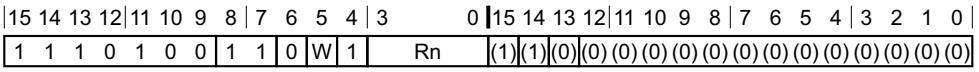
```
RFEDB{<c>}{<q>} <Rn>{!} // Outside or last in IT block
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Decode for this encoding*

```
n = UInt(Rn);  wback = (W == '1');  increment = FALSE;  wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T2




#### *T2 variant*

```
RFE{IA}{<c>}{<q>} <Rn>{!} // Outside or last in IT block
```

#### *Decode for this encoding*

```
n = UInt(Rn);  wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *RFE* on page K1-5482.

## Assembler symbols

| | |
|---|---|
| IA | For encoding A1: is an optional suffix to indicate the Increment After variant.<br>For encoding T2: is an optional suffix for the Increment After form. |
| <c> | For encoding A1: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted.<br>For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| ! | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. |

RFEFA, RFEEA, RFEFD, and RFEED are pseudo-instructions for RFEDA, RFEDB, RFEIA, and RFEIB respectively, referring to their use for popping data from Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.EL == EL0 then
        UNPREDICTABLE;                          // UNDEFINED or NOP
    else
        address = if increment then R[n] else R[n]-8;
        if wordhigher then address = address+4;
        new_pc_value = MemA[address,4];
        spsr = MemA[address+4,4];
        if wback then R[n] = if increment then R[n]+8 else R[n]-8;
        AArch32.ExceptionReturn(new_pc_value, spsr);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

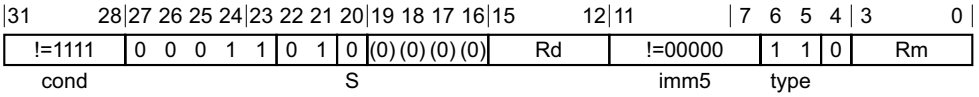
### F6.1.156 ROR (immediate)

Rotate Right (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

#### A1




##### *MOV, shift or rotate by value variant*

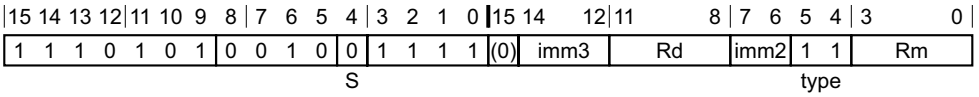
```
ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

#### T3




##### *MOV, shift or rotate by value variant*

Applies when `!(imm3 == 000 && imm2 == 00)`.

```
ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<imm> For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

### Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

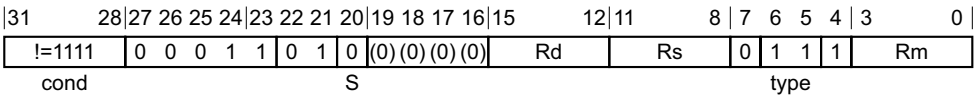
## F6.1.157 ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### A1




#### *Not flag setting variant*

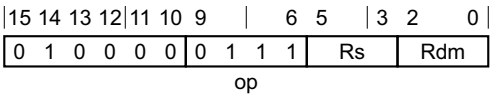
```
ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

### T1




#### *Rotate right variant*

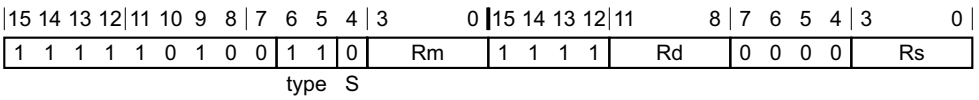
```
ROR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

### T2




#### *Not flag setting variant*

```
ROR<c>.W {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

`ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>`

is equivalent to

`MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>`

and is always the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rdm>` Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the first general-purpose source register, encoded in the "Rm" field.

`<Rs>` Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.158 RORS (immediate)

Rotate Right, setting flags (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

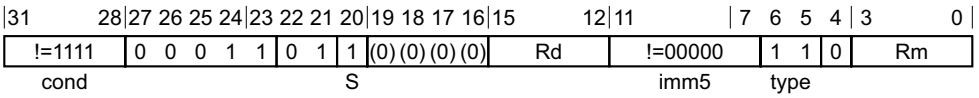
The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

### A1




#### *MOVS, shift or rotate by value variant*

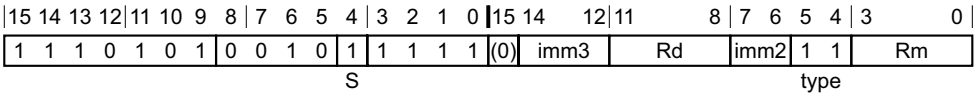
```
RORS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

### T3




#### *MOVS, shift or rotate by value variant*

Applies when `!(imm3 == 000 && imm2 == 00)`.

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

`<imm>` For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

### Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

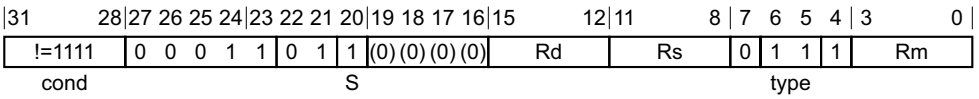
## F6.1.159 RORS (register)

Rotate Right, setting flags (register) provides the value of the contents of a register rotated by a variable number of bits, and updates the condition flags based on the result. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### A1




#### *Flag setting variant*

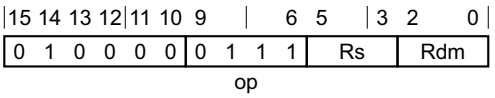
```
RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

### T1




#### *Rotate right variant*

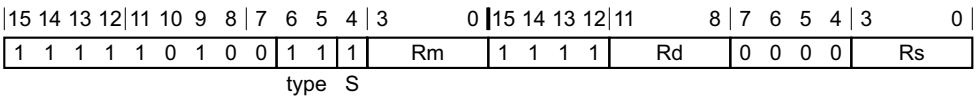
```
RORS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

### T2




#### *Flag setting variant*

```
RORS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

and is always the preferred disassembly.

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Rdm>` | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| `<Rd>` | Is the general-purpose destination register, encoded in the "Rd" field. |
| `<Rm>` | Is the first general-purpose source register, encoded in the "Rm" field. |
| `<Rs>` | Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

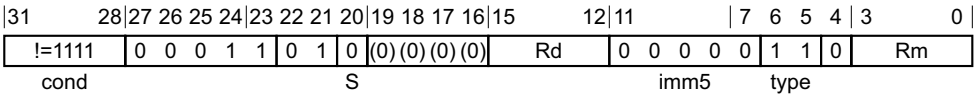
## F6.1.160 RRX

Rotate Right with Extend provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

### A1




#### *MOV, rotate right with extend variant*

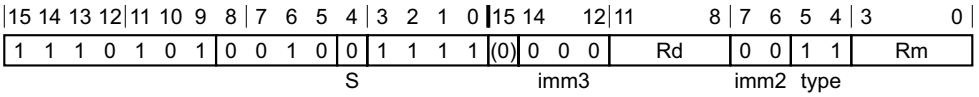
```
RRX{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

### T3




#### *MOV, rotate right with extend variant*

```
RRX{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.161 RRXS

Rotate Right with Extend, setting flags provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

If the destination register is not the PC, this instruction updates the condition flags based on the result, and bit[0] is shifted into the Carry flag.

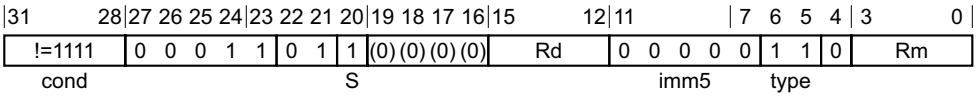
The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

### A1




#### *MOVS, rotate right with extend variant*

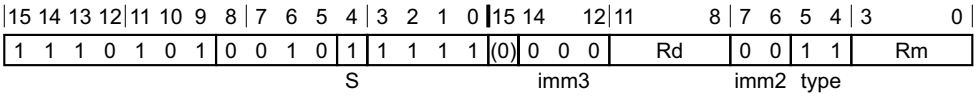
```
RRXS{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

### T3




#### *MOVS, rotate right with extend variant*

```
RRXS{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. ARM deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.162 RSB, RSBS (immediate)

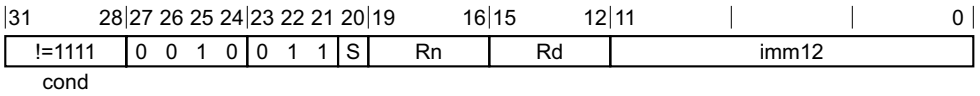
Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1




#### *RSB variant*

Applies when S = 0.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *RSBS variant*

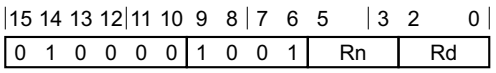
Applies when S = 1.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

### T1




#### *T1 variant*

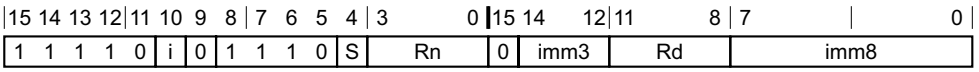
```
RSB<c>{<q>} {<Rd>, }<Rn>, #0 // Inside IT block
RSBS{<q>} {<Rd>, }<Rn>, #0 // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *RSB variant*

Applies when `S = 0`.

```
RSB<c>.W {<Rd>,} <Rn>, #0 // Inside IT block
RSB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *RSBS variant*

Applies when `S = 1`.

```
RSBS.W {<Rd>,} <Rn>, #0 // Outside IT block
RSBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the RSBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1 and T2: is the general-purpose source register, encoded in the "Rn" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(NOT(R[n]), imm32, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.163 RSB, RSBS (register)

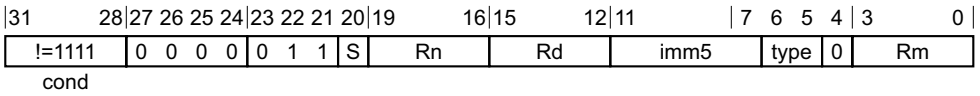
Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *RSB, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *RSB, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *RSBS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *RSBS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

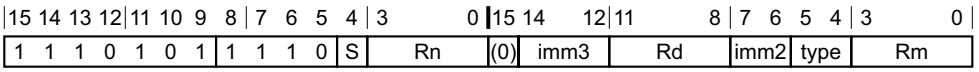
```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *RSB, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *RSB, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *RSBS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && imm2 = 00 && type = 11`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *RSBS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the RSBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

LSL when type = 00

LSR when type = 01

ASR when type = 10

ROR when type = 11

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

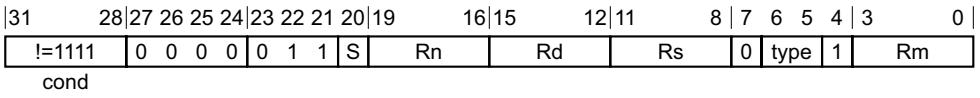
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.164 RSB, RSBS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

#### *Not flag setting variant*

Applies when `S = 0`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, '1');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.165 RSC, RSCS (immediate)

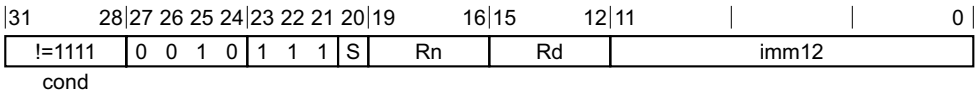
Reverse Subtract with Carry (immediate) subtracts a register value and the value of NOT (Carry flag) from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *RSC variant*

Applies when S = 0.

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *RSCS variant*

Applies when S = 1.

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

#### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the RSCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

<Rn> Is the general-purpose source register, encoded in the "Rn" field. The PC can be used.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<const> An immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(NOT(R[n]), imm32, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.166 RSC, RSCS (register)

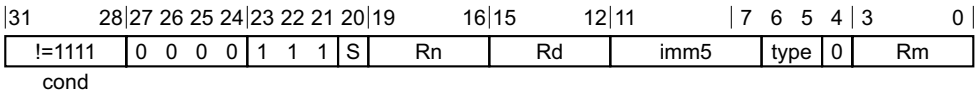
Reverse Subtract with Carry (register) subtracts a register value and the value of NOT (Carry flag) from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *RSC, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *RSC, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *RSCS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *RSCS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the RSCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

<shift> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

LSL when type = 00

LSR when type = 01

ASR when type = 10

ROR when type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

## Operation

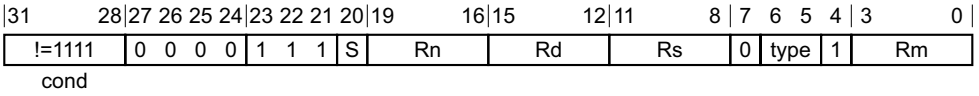
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.167 RSC, RSCS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value and the value of NOT (Carry flag) from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

#### *Not flag setting variant*

Applies when `S = 0`.

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

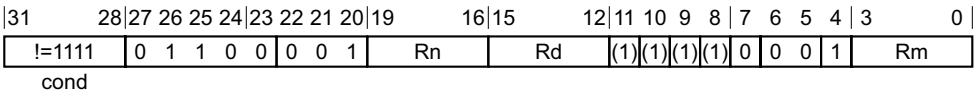
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.168 SADD16

Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

### A1



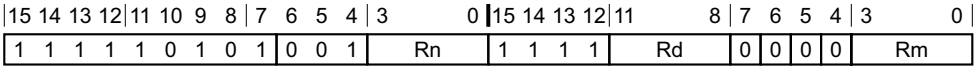

#### *A1 variant*

```
SADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

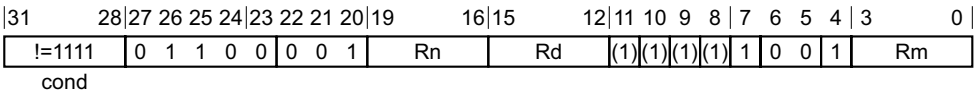
```
R[d]<31:16> = sum2<15:0>;
PSTATE.GE<1:0> = if sum1 >= 0 then '11' else '00';
PSTATE.GE<3:2> = if sum2 >= 0 then '11' else '00';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.169 SADD8

Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

### A1



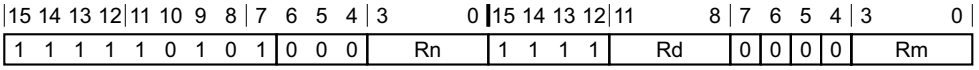

#### *A1 variant*

```
SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

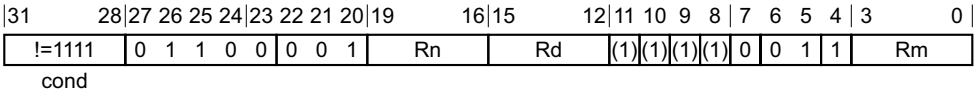
```
sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
R[d]<7:0>   = sum1<7:0>;
R[d]<15:8>  = sum2<7:0>;
R[d]<23:16> = sum3<7:0>;
R[d]<31:24> = sum4<7:0>;
PSTATE.GE<0>  = if sum1 >= 0 then '1' else '0';
PSTATE.GE<1>  = if sum2 >= 0 then '1' else '0';
PSTATE.GE<2>  = if sum3 >= 0 then '1' else '0';
PSTATE.GE<3>  = if sum4 >= 0 then '1' else '0';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.170 SASX

Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets PSTATE.GE according to the results.

### A1



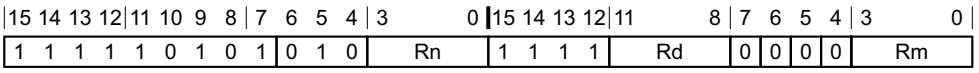

#### *A1 variant*

```
SASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
R[d]<15:0>  = diff<15:0>;
R[d]<31:16> = sum<15:0>;
PSTATE.GE<1:0> = if diff >= 0 then '11' else '00';
PSTATE.GE<3:2> = if sum  >= 0 then '11' else '00';
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F6.1.171 SBC, SBCS (immediate)

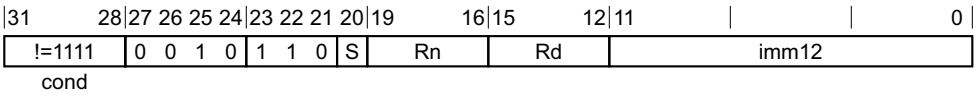
Subtract with Carry (immediate) subtracts an immediate value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1




#### *SBC variant*

Applies when S = 0.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *SBCS variant*

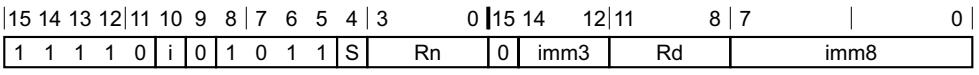
Applies when S = 1.

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

### T1




#### *SBC variant*

Applies when S = 0.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *SBCS variant*

Applies when S = 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.172 SBC, SBCS (register)

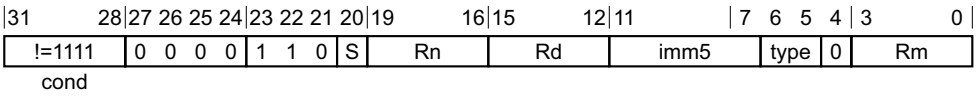
Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1




#### *SBC, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *SBC, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *SBCS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *SBCS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

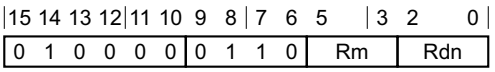
```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



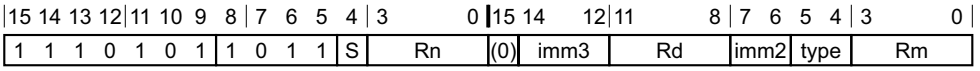

#### *T1 variant*

```
SBC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
SBCS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *SBC, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *SBC, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
SBC<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *SBCS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && imm2 = 00 && type = 11`.

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *SBCS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
SBCS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| LSL | when type = 00 |
| LSR | when type = 01 |
| ASR | when type = 10 |
| ROR | when type = 11 |

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

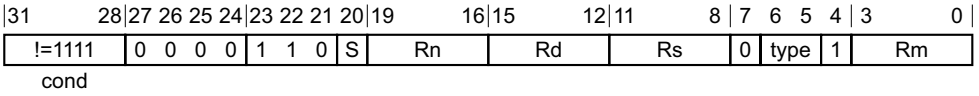
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.173 SBC, SBCS (register-shifted register)

Subtract with Carry (register-shifted register) subtracts a register-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### A1




##### *Flag setting variant*

Applies when `S = 1`.

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Not flag setting variant*

Applies when `S = 0`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

#### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

#### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

- `LSL` when `type = 00`
- `LSR` when `type = 01`
- `ASR` when `type = 10`
- `ROR` when `type = 11`

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Operation

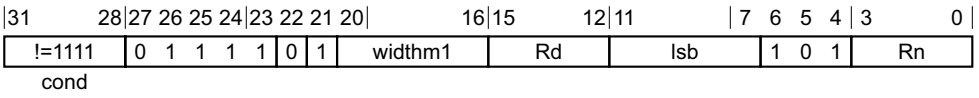
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.174 SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from a register, sign-extends them to 32 bits, and writes the result to the destination register.

### A1



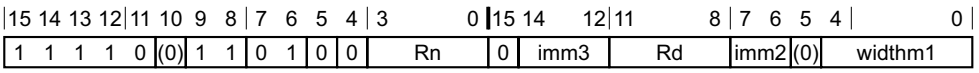

#### *A1 variant*

```
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *SBFX* on page K1-5455.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <lsb> | For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.<br>For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field. |
| <width> | Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

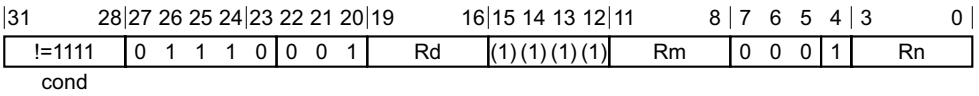
```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.175 SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition flags are not affected.

### A1



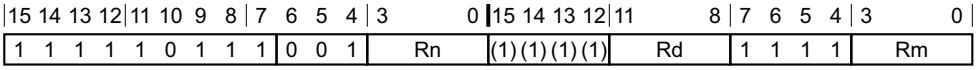

#### *A1 variant*

```
SDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see .

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the dividend, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the divisor, encoded in the "Rm" field. |

## Overflow

If the signed integer division `0x80000000` / `0xFFFFFFFF` is performed, the pseudocode produces the intermediate integer result $+2^{31}$, that overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to <Rd> must be the bottom 32 bits of the binary representation of $+2^{31}$. So the result of the division is `0x80000000`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

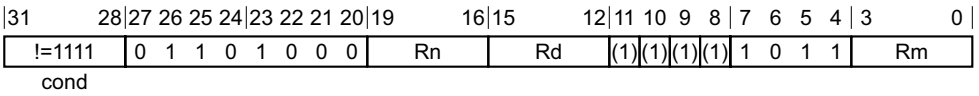
```
if ConditionPassed() then
    EncodingSpecificOperations();
    if SInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(Real(SInt(R[n])) / Real(SInt(R[m])));
    R[d] = result<31:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.176 SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the PSTATE.GE flags.

### A1



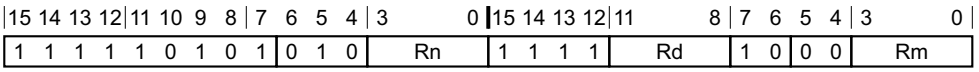

#### *A1 variant*

```
SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<7:0>   = if PSTATE.GE<0> == '1' then R[n]<7:0>   else R[m]<7:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

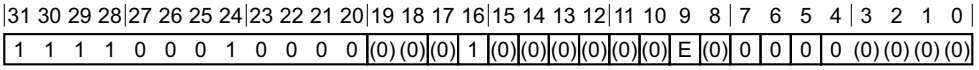
```
R[d]<15:8>  = if PSTATE.GE<1> == '1' then R[n]<15:8>  else R[m]<15:8>;
R[d]<23:16> = if PSTATE.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
R[d]<31:24> = if PSTATE.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.177 SETEND

Set Endianness writes a new value to PSTATE.E.

### A1



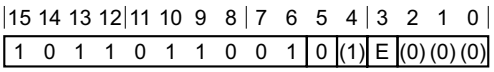

#### *A1 variant*

```
SETEND{<q>} <endian_specifier> // Cannot be conditional
```

#### *Decode for this encoding*

```
set_bigend = (E == '1');
```

### T1




#### *T1 variant*

```
SETEND{<q>} <endian_specifier> // Not permitted in IT block
```

#### *Decode for this encoding*

```
set_bigend = (E == '1');
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446.

<endian_specifier> Is the endianness to be selected, and the value to be set in PSTATE.E, encoded in the "E" field. It can have the following values:

LE when E = 0

BE when E = 1

### Operation for all encodings

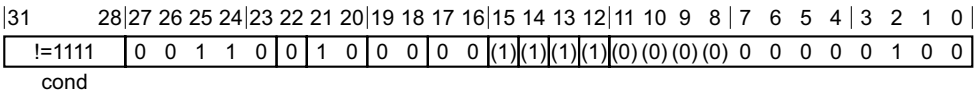
```
EncodingSpecificOperations();
AArch32.CheckSETENDEnabled();
PSTATE.E = if set_bigend then '1' else '0';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.178 SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait For Event and Send Event* on page G1-3834.

### A1



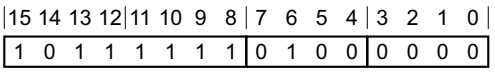

#### *A1 variant*

```
SEV{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T1



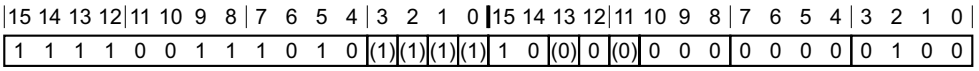

#### *T1 variant*

```
SEV{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T2




#### *T2 variant*

```
SEV{<c>}.W
```

#### *Decode for this encoding*

```
// No additional decoding required
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

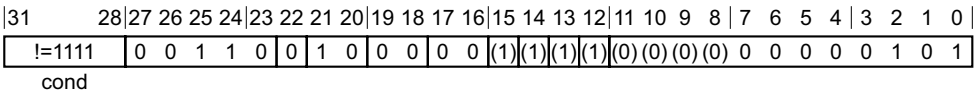
```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.179 SEVL

Send Event Local is a hint instruction. It causes an event to be signaled locally without the requirement to affect other PEs in the multiprocessor system. It can prime a wait-loop which starts with a WFE instruction.

### A1



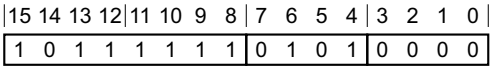

#### *A1 variant*

```
SEVL{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T1



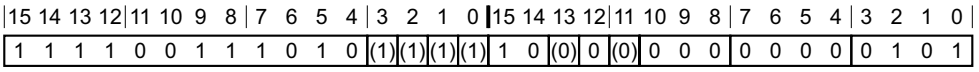

#### *T1 variant*

```
SEVL{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T2




#### *T2 variant*

```
SEVL{<c>}.W
```

#### *Decode for this encoding*

```
// No additional decoding required
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

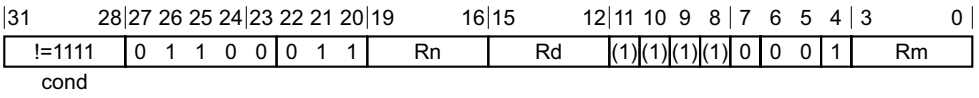
```
if ConditionPassed() then
    EncodingSpecificOperations();
    EventRegisterSet();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.180 SHADD16

Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

### A1



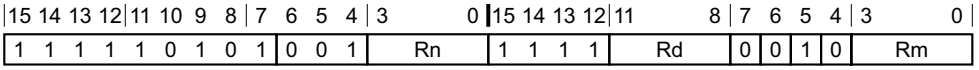

#### *A1 variant*

```
SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

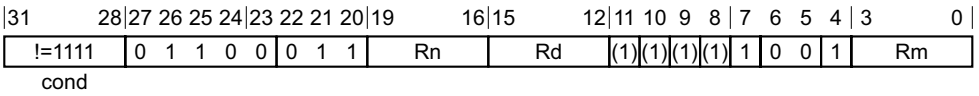
```
sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
R[d]<15:0>  = sum1<16:1>;
R[d]<31:16> = sum2<16:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.181 SHADD8

Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

### A1



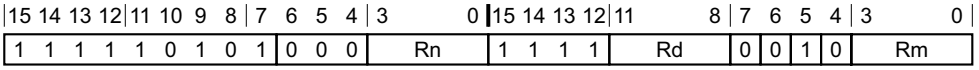

#### *A1 variant*

```
SHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

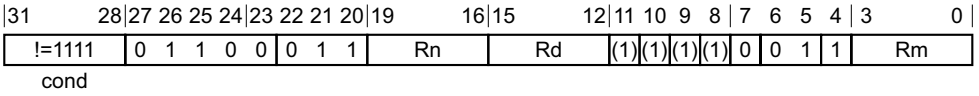
```
sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
R[d]<7:0>   = sum1<8:1>;
R[d]<15:8>  = sum2<8:1>;
R[d]<23:16> = sum3<8:1>;
R[d]<31:24> = sum4<8:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.182 SHASX

Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

### A1



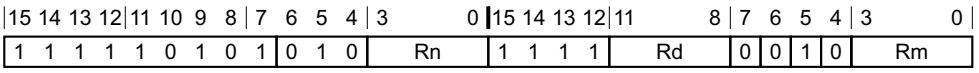

#### *A1 variant*

```
SHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

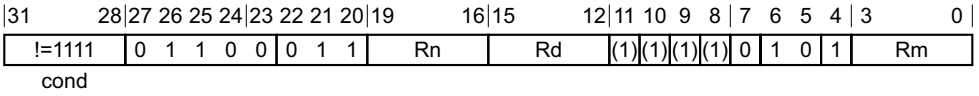
```
sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
R[d]<15:0>  = diff<16:1>;
R[d]<31:16> = sum<16:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.183 SHSAX

Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

### A1



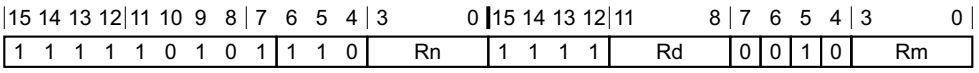

#### *A1 variant*

```
SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

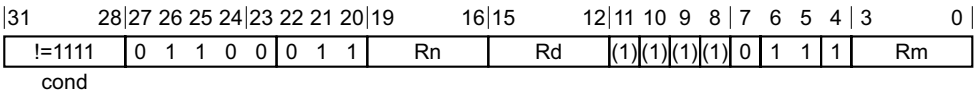
```
diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
R[d]<15:0>  = sum<16:1>;
R[d]<31:16> = diff<16:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.184 SHSUB16

Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

### A1



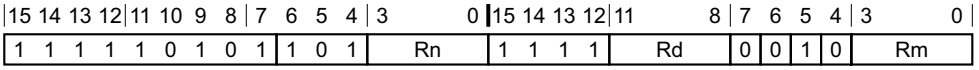

#### *A1 variant*

```
SHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

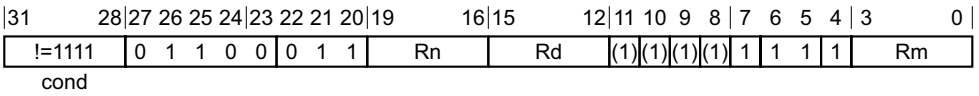
```
diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
R[d]<15:0>  = diff1<16:1>;
R[d]<31:16> = diff2<16:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.185 SHSUB8

Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register.

### A1



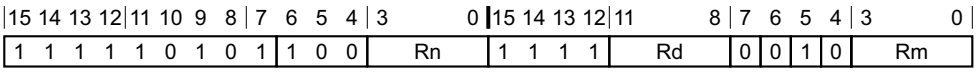

#### *A1 variant*

```
SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
R[d]<7:0>   = diff1<8:1>;
R[d]<15:8>  = diff2<8:1>;
R[d]<23:16> = diff3<8:1>;
R[d]<31:24> = diff4<8:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.186 SMC

Secure Monitor Call causes a Secure Monitor Call exception. For more information see *Secure Monitor Call (SMC) exception* on page G1-3816.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in User mode.

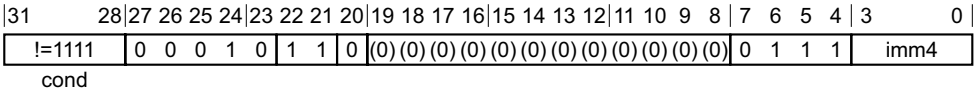
If the values of .TSC and .SCD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception that is taken to EL3. When EL3 is using AArch32 this exception is taken to Monitor mode. When EL3 is using AArch64, it is the SCR_EL3.SMD bit, rather than the .SCD bit, that can change the effect of executing an SMC instruction.

If the value of .TSC is 1, execution of an SMC instruction in a Non-secure EL1 mode generates an exception that is taken to EL2, regardless of the value of .SCD. When EL2 is using AArch32, this is a Hyp Trap exception that is taken to Hyp mode. For more information see .

If the value of .TSC is 0 and the value of .SCD is 1, the SMC instruction is:

- UNDEFINED in Non-secure state.
- CONSTRAINED UNPREDICTABLE if executed in Secure state at EL1 or higher.

#### A1



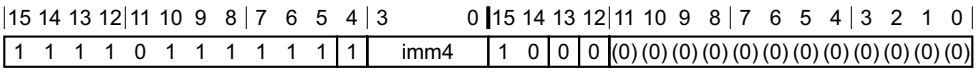

##### *A1 variant*

```
SMC{<c>}{<q>} {#}<imm4>
```

##### *Decode for this encoding*

```
// imm4 is for assembly/disassembly only and is ignored by hardware
```

#### T1




##### *T1 variant*

```
SMC{<c>}{<q>} {#}<imm4>
```

##### *Decode for this encoding*

```
// imm4 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<imm4> Is a 4-bit unsigned immediate value, in the range 0 to 15, encoded in the "imm4" field. This is ignored by the PE. The Secure Monitor Call exception handler (Secure Monitor code) can use this value to determine what service is being requested, but ARM does not recommend this.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;

    AArch32.CheckForSMCTrap();

    if !ELUsingAArch32(EL3) then
        if SCR_EL3.SMD == '1' then
            // SMC disabled.
            UNDEFINED;
    else
        if SCR.SCD == '1' then
            // SMC disabled
            if IsSecure() then
                // Executes either as a NOP or UNALLOCATED.
                c = ConstrainUnpredictable(Unpredictable_SMD);
                assert c IN {Constraint_NOP, Constraint_UNDEF};
                if c == Constraint_NOP then EndOfInstruction();
            UNDEFINED;

    if !ELUsingAArch32(EL3) then
        AArch64.CallSecureMonitor(Zeros(16));
    else
        AArch32.TakeSMCException();
```

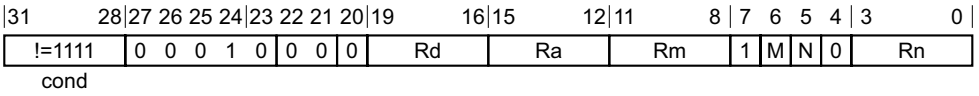

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.187 SMLABB, SMLABT, SMLATB, SMLATT

Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets PSTATE.Q to 1. It is not possible for overflow to occur during the multiplication.

### A1




#### *SMLABB variant*

Applies when `M = 0 && N = 0`.

```
SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLABT variant*

Applies when `M = 1 && N = 0`.

```
SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLATB variant*

Applies when `M = 0 && N = 1`.

```
SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLATT variant*

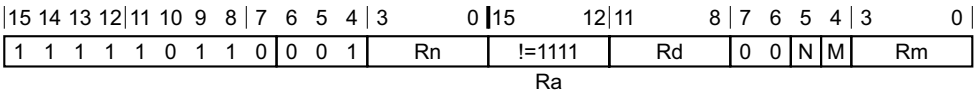
Applies when `M = 1 && N = 1`.

```
SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

### T1




#### *SMLABB variant*

Applies when `N = 0 && M = 0`.

```
SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLABT variant*

Applies when `N = 0 && M = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLATB variant*

Applies when `N = 1 && M = 0`.

```
SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLATT variant*

Applies when `N = 1 && M = 1`.

```
SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
if Ra == '1111' then SEE SMULBB, SMULBT, SMULTB, SMULTT;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

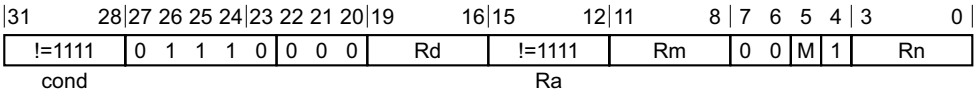
## F6.1.188 SMLAD, SMLADX

Signed Multiply Accumulate Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications.

### A1




#### *SMLAD variant*

Applies when `M = 0`.

```
SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLADX variant*

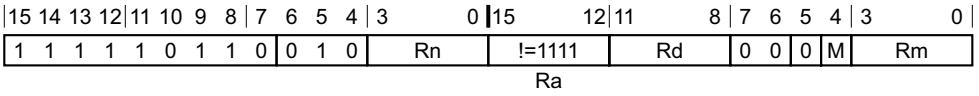
Applies when `M = 1`.

```
SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
if Ra == '1111' then SEE SMUAD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *SMLAD variant*

Applies when `M = 0`.

```
SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLADX variant*

Applies when `M = 1`.

```
SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
if Ra == '1111' then SEE SMUAD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```

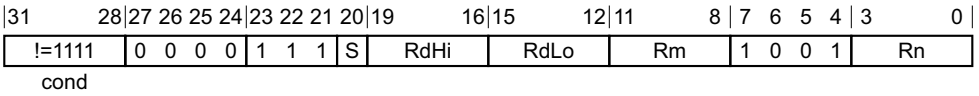

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.189 SMLAL, SMLALS

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
SMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Not flag setting variant*

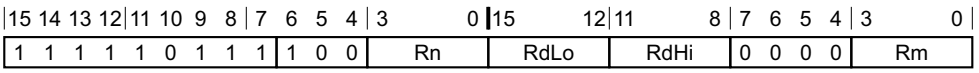
Applies when `S = 0`.

```
SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *SMLAL* on page K1-5456.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<RdLo> Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();
     result = SInt(R[n]) * SInt(R[m]) + SInt(R

[dHi]:R[dLo]);
     R[dHi] = result<63:32>;
     R[dLo] = result<31:0>;
     if setflags then
         PSTATE.N = result<63>;
         PSTATE.Z = IsZeroBit(result<63:0>);
         // PSTATE.C, PSTATE.V unchanged
```

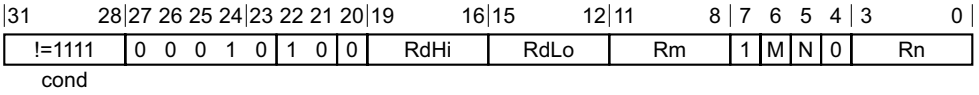

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.190 SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo $2^{64}$.

### A1




#### *SMLALBB variant*

Applies when `M = 0 && N = 0`.

```
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *SMLALBT variant*

Applies when `M = 1 && N = 0`.

```
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *SMLALTB variant*

Applies when `M = 0 && N = 1`.

```
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *SMLALTT variant*

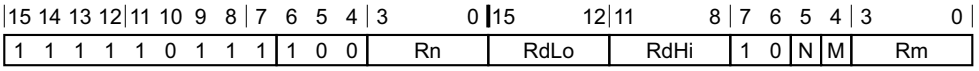
Applies when `M = 1 && N = 1`.

```
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

### T1




#### *SMLALBB variant*

Applies when `N = 0 && M = 0`.

```
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### SMLALBT variant

Applies when N = 0 && M = 1.

```
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### SMLALTB variant

Applies when N = 1 && M = 0.

```
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### SMLALTT variant

Applies when N = 1 && M = 1.

```
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### Decode for all variants of this encoding

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *SMLALBB, SMLALBT, SMLALTB, SMLALTT* on page K1-5456.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<RdLo> Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> For encoding A1: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.

For encoding T1: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

For encoding T1: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <x>), encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

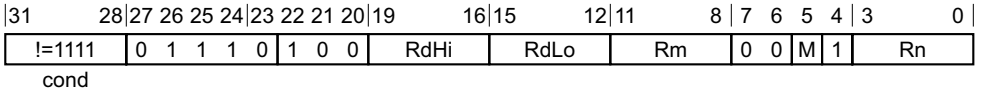
## F6.1.191 SMLALD, SMLALDX

Signed Multiply Accumulate Long Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo $2^{64}$.

### A1




#### *SMLALD variant*

Applies when `M = 0`.

```
SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *SMLALDX variant*

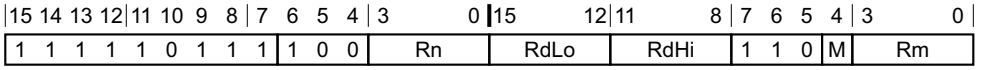
Applies when `M = 1`.

```
SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

### T1




#### *SMLALD variant*

Applies when `M = 0`.

```
SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *SMLALDX variant*

Applies when `M = 1`.

```
SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *SMLALD* on page K1-5457.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<RdLo>` Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

`<RdHi>` Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

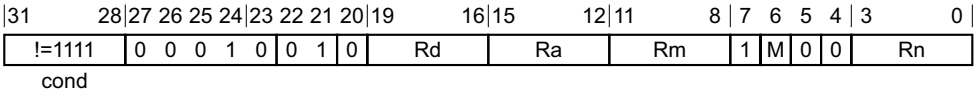

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.192 SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets PSTATE.Q to 1. No overflow can occur during the multiplication.

### A1




#### *SMLAWB variant*

Applies when `M = 0`.

```
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLAWT variant*

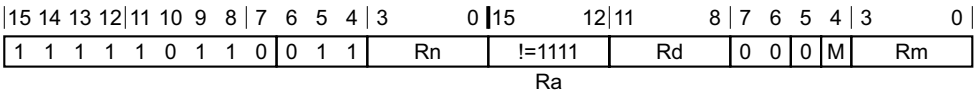
Applies when `M = 1`.

```
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

### T1




#### *SMLAWB variant*

Applies when `M = 0`.

```
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLAWT variant*

Applies when `M = 1`.

```
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
if Ra == '1111' then SEE SMULWB, SMULWT;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(R[n]) * SInt(operand2) + (SInt(R[a]) << 16);
    R[d] = result<47:16>;
    if (result >> 16) != SInt(R[d]) then  // Signed overflow
        PSTATE.Q = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

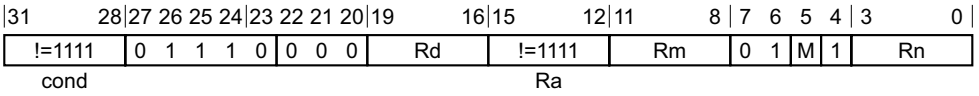
## F6.1.193 SMLSD, SMLSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

### A1




#### *SMLSD variant*

Applies when `M = 0`.

```
SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLSDX variant*

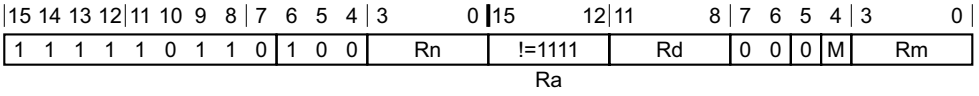
Applies when `M = 1`.

```
SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
if Ra == '1111' then SEE SMUSD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *SMLSD variant*

Applies when `M = 0`.

```
SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMLSDX variant*

Applies when `M = 1`.

```
SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
if Ra == '1111' then SEE SMUSD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

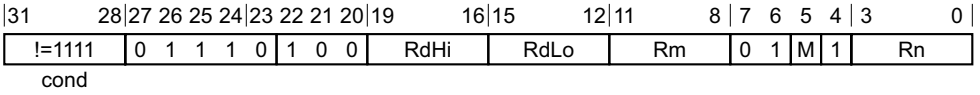
## F6.1.194 SMLSLD, SMLSLDX

Signed Multiply Subtract Long Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo $2^{64}$.

### A1




#### *SMLSLD variant*

Applies when `M = 0`.

```
SMLSLD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *SMLSLDX variant*

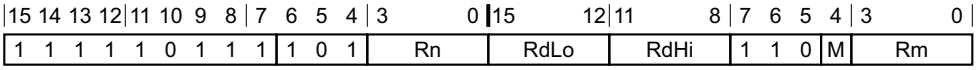
Applies when `M = 1`.

```
SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

### T1




#### *SMLSLD variant*

Applies when `M = 0`.

```
SMLSLD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *SMLSLDX variant*

Applies when `M = 1`.

```
SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *SMLSLD* on page K1-5457.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<RdLo>` Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

`<RdHi>` Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

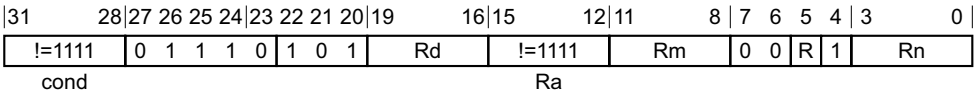

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.195 SMMLA, SMMLAR

Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

#### A1




##### *SMMLA variant*

Applies when `R = 0`.

```
SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

##### *SMMLAR variant*

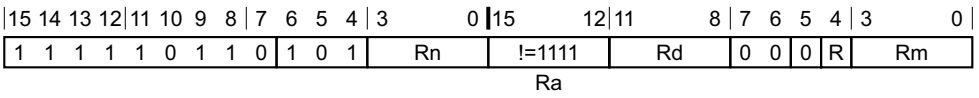
Applies when `R = 1`.

```
SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

##### *Decode for all variants of this encoding*

```
if Ra == '1111' then SEE SMMUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

#### T1




##### *SMMLA variant*

Applies when `R = 0`.

```
SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

##### *SMMLAR variant*

Applies when `R = 1`.

```
SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

##### *Decode for all variants of this encoding*

```
if Ra == '1111' then SEE SMMUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

### Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();
     result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R

[m]);
     if round then result = result + 0x80000000;
     R[d] = result<63:32>;
```

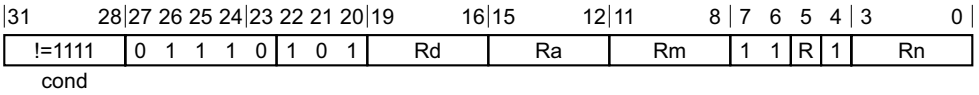

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.196 SMMLS, SMMLSR

Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.

Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the result of the subtraction before the high word is extracted.

### A1




#### *SMMLS variant*

Applies when `R = 0`.

```
SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMMLSR variant*

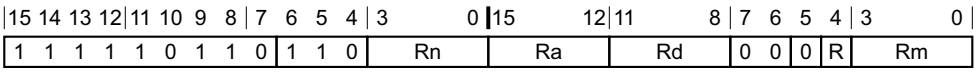
Applies when `R = 1`.

```
SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

### T1




#### *SMMLS variant*

Applies when `R = 0`.

```
SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *SMMLSR variant*

Applies when `R = 1`.

```
SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

### Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();
     result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R

[m]);
     if round then result = result + 0x80000000;
     R[d] = result<63:32>;
```

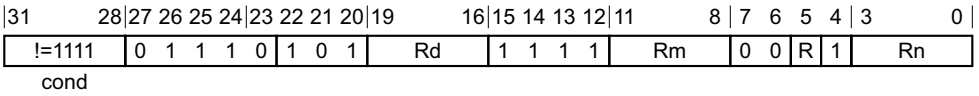

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.197 SMMUL, SMMULR

Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

### A1




#### *SMMUL variant*

Applies when `R = 0`.

```
SMMUL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMMULR variant*

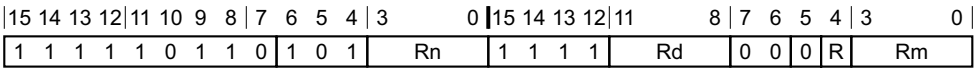
Applies when `R = 1`.

```
SMMULR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *SMMUL variant*

Applies when `R = 0`.

```
SMMUL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMMULR variant*

Applies when `R = 1`.

```
SMMULR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

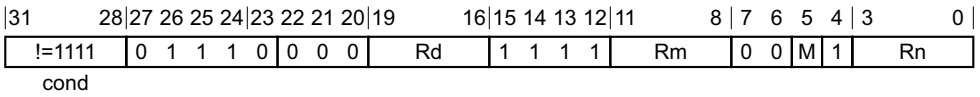
## F6.1.198 SMUAD, SMUADX

Signed Dual Multiply Add performs two signed 16 x 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the addition overflows. The multiplications cannot overflow.

### A1




#### *SMUAD variant*

Applies when `M = 0`.

```
SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMUADX variant*

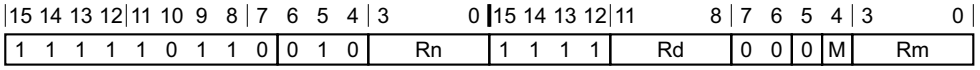
Applies when `M = 1`.

```
SMUADX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *SMUAD variant*

Applies when `M = 0`.

```
SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMUADX variant*

Applies when `M = 1`.

```
SMUADX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

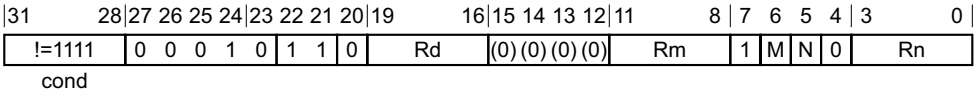
```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2;
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.199 SMULBB, SMULBT, SMULTB, SMULTT

Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

### A1




#### *SMULBB variant*

Applies when `M = 0 && N = 0`.

```
SMULBB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMULBT variant*

Applies when `M = 1 && N = 0`.

```
SMULBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMULTB variant*

Applies when `M = 0 && N = 1`.

```
SMULTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMULTT variant*

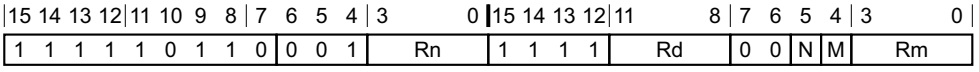
Applies when `M = 1 && N = 1`.

```
SMULTT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *SMULBB variant*

Applies when `N = 0 && M = 0`.

```
SMULBB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMULBT variant*

Applies when `N = 0 && M = 1`.

```
SMULBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *SMULTB variant*

Applies when `N = 1 && M = 0`.

```
SMULTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMULTT variant*

Applies when `N = 1 && M = 1`.

```
SMULTT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2);
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

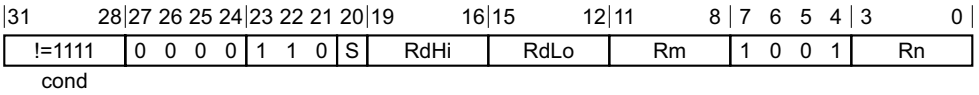

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.200 SMULL, SMULLS

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
SMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Not flag setting variant*

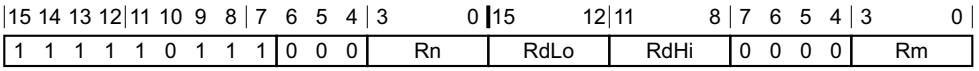
Applies when `S = 0`.

```
SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *SMULL* on page K1-5457.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<RdLo> Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation for all encodings

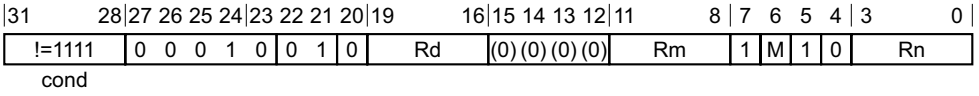
```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.201 SMULWB, SMULWT

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

### A1




#### *SMULWB variant*

Applies when `M = 0`.

```
SMULWB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMULWT variant*

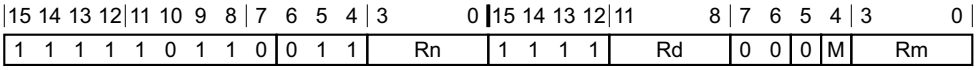
Applies when `M = 1`.

```
SMULWT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *SMULWB variant*

Applies when `M = 0`.

```
SMULWB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMULWT variant*

Applies when `M = 1`.

```
SMULWT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    product = SInt(R[n]) * SInt(operand2);
    R[d] = product<47:16>;
    // Signed overflow cannot occur
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

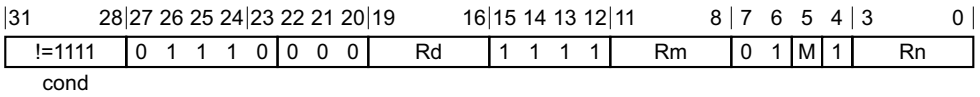
## F6.1.202 SMUSD, SMUSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow cannot occur.

### A1




#### *SMUSD variant*

Applies when `M = 0`.

```
SMUSD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMUSDX variant*

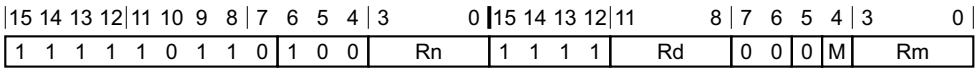
Applies when `M = 1`.

```
SMUSDX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *SMUSD variant*

Applies when `M = 0`.

```
SMUSD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *SMUSDX variant*

Applies when `M = 1`.

```
SMUSDX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2;
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.203 SRS, SRSDA, SRSDB, SRSIA, SRSIB

Store Return State stores the LR_<current_mode> and SPSR_<current_mode> to the stack of a specified mode. For information about memory accesses see *Memory accesses* on page F2-2452.

SRS is UNDEFINED in Hyp mode.

SRS is UNPREDICTABLE if it is executed in User or System mode, or if the specified mode is any of the following:

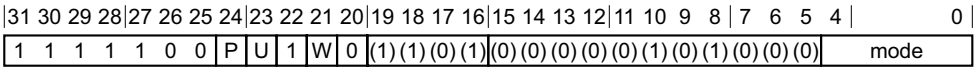
- Not implemented.
- A mode that  does not show.
- Not accessible from the context that the SRS instruction is executed in, as follows:
  - A mode that is at a higher Exception level.
  - When executing in Non-secure state, Monitor mode.

SRS is trapped to EL3 if all of the following are true:

- It is executed at Secure PL1.
- The specified mode is monitor mode.
- EL3 is using AArch64.

See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-1602.

### A1




#### *Decrement After variant*

Applies when `P = 0 && U = 0`.

```
SRSDA{<c>}{<q>} SP{!}, #<mode>
```

#### *Decrement Before variant*

Applies when `P = 1 && U = 0`.

```
SRSDB{<c>}{<q>} SP{!}, #<mode>
```

#### *Increment After variant*

Applies when `P = 0 && U = 1`.

```
SRS{IA}{<c>}{<q>} SP{!}, #<mode>
```

#### *Increment Before variant*

Applies when `P = 1 && U = 1`.

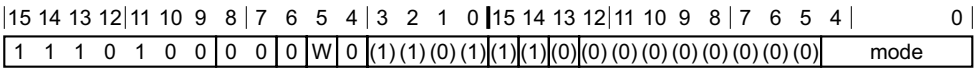
```
SRSIB{<c>}{<q>} SP{!}, #<mode>
```

#### *Decode for all variants of this encoding*

```
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



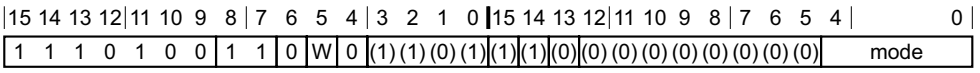

#### *T1 variant*

```
SRSDB{<c>}{<q>} SP{!}, #<mode>
```

#### *Decode for this encoding*

```
wback = (W == '1');  increment = FALSE;  wordhigher = FALSE;
```

### T2




#### *T2 variant*

```
SRS{IA}{<c>}{<q>} SP{!}, #<mode>
```

#### *Decode for this encoding*

```
wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *SRS (T32)* on page K1-5482 and .

## Assembler symbols

| | |
|---|---|
| IA | For encoding A1: is an optional suffix to indicate the Increment After variant.<br>For encoding T2: is an optional suffix for the Increment After form. |
| <c> | For encoding A1: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted.<br>For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| ! | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. |
| <mode> | Is the number of the mode whose Banked SP is used as the base register, encoded in the "mode" field. For details of PE modes and their numbers see . |

SRSFA, SRSEA, SRSFD, and SRSED are pseudo-instructions for SRSIB, SRSIA, SRSDB, and SRSDA respectively, referring to their use for pushing data onto Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

## Operation for all encodings

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then          // UNDEFINED at EL2
            UNDEFINED;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
            UNPREDICTABLE;
        elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || !IsSecure()  then
                UNPREDICTABLE;
            elsif !ELUsingAArch32(EL3) then
                AArch64.MonitorModeTrap();

        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4]   = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13,mode] = if increment then base+8 else base-8;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then          // UNDEFINED at EL2
            UNDEFINED;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
            UNPREDICTABLE;
        elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || !IsSecure()  then
                UNPREDICTABLE;
            elsif !ELUsingAArch32(EL3) then
                AArch64.MonitorModeTrap();

        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4]   = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13,mode] = if increment then base+8 else base-8;
```

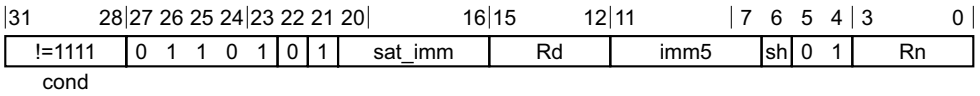

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.204 SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

### A1




#### *Arithmetic shift right variant*

Applies when `sh = 1`.

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

#### *Logical shift left variant*

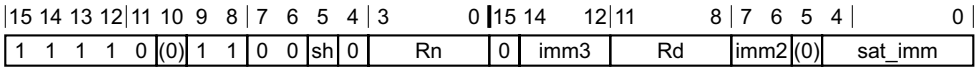
Applies when `sh = 0`.

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *Arithmetic shift right variant*

Applies when `sh = 1 && !(imm3 == 000 && imm2 == 00)`.

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

#### *Logical shift left variant*

Applies when `sh = 0`.

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

#### *Decode for all variants of this encoding*

```
if sh == '1' && (imm3:imm2) == '00000' then SEE SSAT16;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <imm> | Is the bit position for saturation, in the range 1 to 32, encoded in the "sat_imm" field as <imm>-1. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <amount> | For encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.<br>For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32.<br>For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.<br>For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    (result, sat) = SignedSatQ(SInt(operand), saturate_to);
    R[d] = SignExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
```

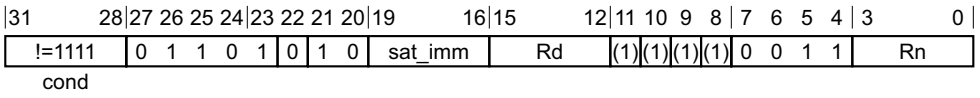

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.205 SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

### A1



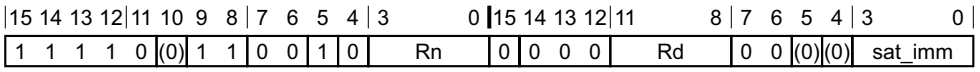

#### *A1 variant*

```
SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <imm> | Is the bit position for saturation, in the range 1 to 16, encoded in the "sat_imm" field as <imm>-1. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = SignedSatQ(SInt(R[n]<31:16>), saturate_to);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

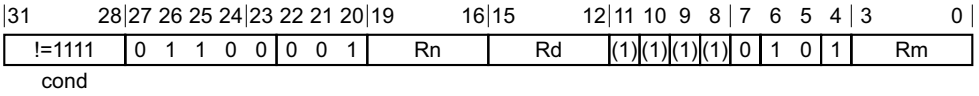
```
R[d]<15:0> = SignExtend(result1, 16);
R[d]<31:16> = SignExtend(result2, 16);
if sat1 || sat2 then
    PSTATE.Q = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.206 SSAX

Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets PSTATE.GE according to the results.

### A1



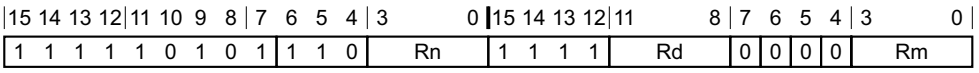

#### *A1 variant*

```
SSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

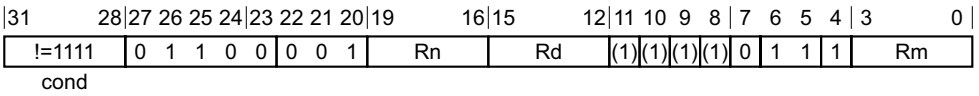
```
R[d]<15:0>  = sum<15:0>;
R[d]<31:16> = diff<15:0>;
PSTATE.GE<1:0> = if sum  >= 0 then '11' else '00';
PSTATE.GE<3:2> = if diff >= 0 then '11' else '00';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.207 SSUB16

Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

### A1



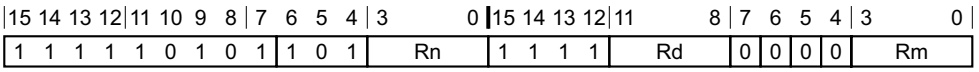

#### *A1 variant*

```
SSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<15:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

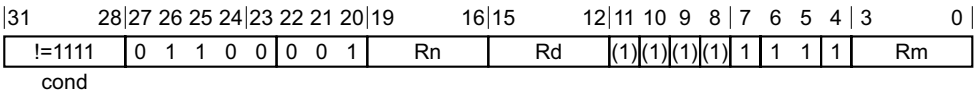
```
R[d]<31:16> = diff2<15:0>;
PSTATE.GE<1:0> = if diff1 >= 0 then '11' else '00';
PSTATE.GE<3:2> = if diff2 >= 0 then '11' else '00';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.208 SSUB8

Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

### A1



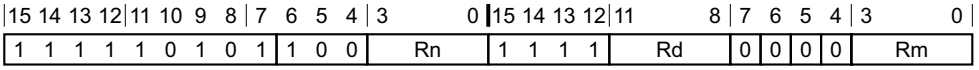

#### *A1 variant*

```
SSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
R[d]<7:0>   = diff1<7:0>;
R[d]<15:8>  = diff2<7:0>;
R[d]<23:16> = diff3<7:0>;
R[d]<31:24> = diff4<7:0>;
PSTATE.GE<0>  = if diff1 >= 0 then '1' else '0';
PSTATE.GE<1>  = if diff2 >= 0 then '1' else '0';
PSTATE.GE<2>  = if diff3 >= 0 then '1' else '0';
PSTATE.GE<3>  = if diff4 >= 0 then '1' else '0';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.209 STC, STC2

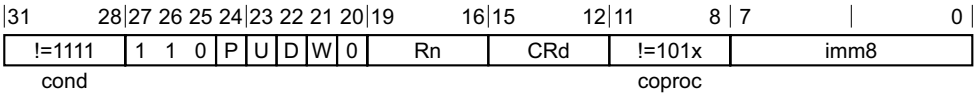
Store Coprocessor stores data from a conceptual coprocessor to a sequence of consecutive memory addresses.

This is a generic coprocessor instruction. The coproc field identifies the target conceptual coprocessor. This must be one of CP10, CP11, CP14, or CP15, and for these CP values, this manual defines the imm8, CRd, and D field values that are valid STC and STC2 instructions. Other encodings are UNDEFINED. For more information see *Conceptual coprocessor support* on page E1-2353 and *General behavior of System registers* on page G4-4118.

In an implementation that includes EL2, the permitted STC access to a System register can be trapped to Hyp mode, meaning that an attempt to execute an STC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *Trapping general Non-secure System register accesses to debug registers* on page G1-3871.

For simplicity, the STC pseudocode does not show this possible trap to Hyp mode.

### A1




#### *Offset variant*

Applies when P = 1 && W = 0.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when P = 0 && W = 1.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when P = 1 && W = 1.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

#### *Unindexed variant*

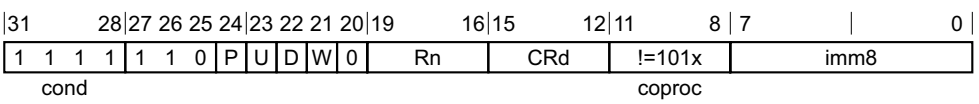
Applies when P = 0 && U = 1 && W = 0.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

#### *Decode for all variants of this encoding*

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
n = UInt(Rn);  cp = UInt(coproc);
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

### A2





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when P = 1 && W = 0.

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when P = 0 && W = 1.

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when P = 1 && W = 1.

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

#### *Unindexed variant*

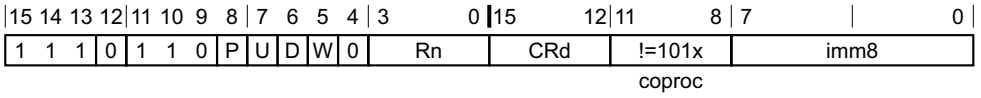
Applies when P = 0 && U = 1 && W = 0.

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

#### *Decode for all variants of this encoding*

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
if coproc == '101x' then UNDEFINED;
n = UInt(Rn);  cp = UInt(coproc);
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

### T1




#### *Offset variant*

Applies when P = 1 && W = 0.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when P = 0 && W = 1.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when P = 1 && W = 1.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

#### *Unindexed variant*

Applies when P = 0 && U = 1 && W = 0.

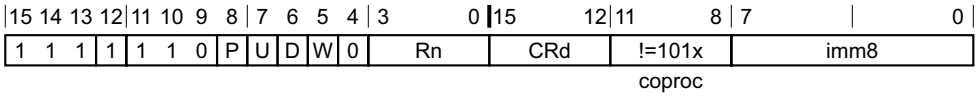
```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### Decode for all variants of this encoding

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
if coproc == '101x' then SEE "Advanced SIMD and Floating-point";
n = UInt(Rn);  cp = UInt(coproc);
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

### T2




#### Offset variant

Applies when `P = 1 && W = 0`.

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

#### Post-indexed variant

Applies when `P = 0 && W = 1`.

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

Applies when `P = 1 && W = 1`.

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

#### Unindexed variant

Applies when `P = 0 && U = 1 && W = 0`.

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

#### Decode for all variants of this encoding

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
if coproc == '101x' then UNDEFINED;
n = UInt(Rn);  cp = UInt(coproc);
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STC, STC2* on page K1-5458.

Advanced SIMD and floating-point: Extension register load/store instructions.

## Assembler symbols

| | |
|---|---|
| `L` | If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form. |
| `<c>` | For encoding A1, T1 and T2: see *Standard assembler syntax fields* on page F2-2446.<br>For encoding A2: see *Standard assembler syntax fields* on page F2-2446. `<c>` must be `AL` or omitted. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<coproc>` Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15.

`<CRd>` Is the coprocessor register to be transferred, encoded in the "CRd" field.

`<Rn>` For encoding A1, A2: is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding A1, A2, T1, T2: is the general-purpose base register, encoded in the "Rn" field.

`<option>` Is a coprocessor option, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Coproc_CheckInstr(cp, ThisInstr());
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    repeat
        MemA[address,4] = Coproc_GetWordToStore(cp, ThisInstr());
        address = address + 4;
    until Coproc_DoneStoring(cp, ThisInstr());
    if wback then R[n] = offset_addr;
```

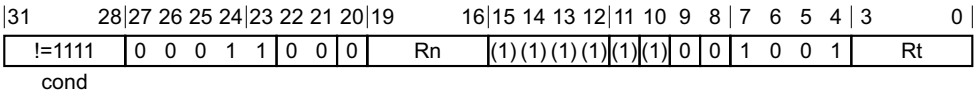

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.210 STL

Store-Release Word stores a word from a register to memory. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



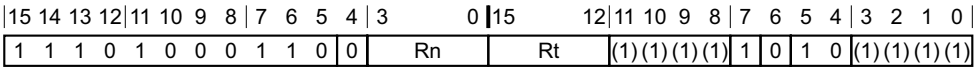

#### *A1 variant*

```
STL{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STL{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 4] = R[t];
```

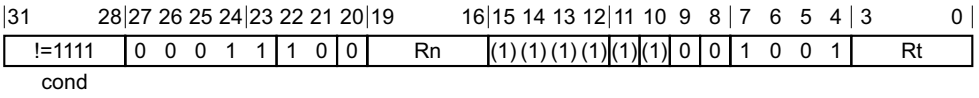

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.211 STLB

Store-Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



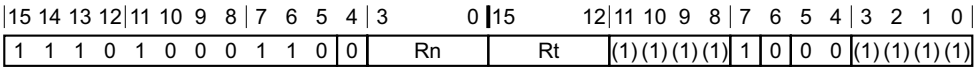

#### *A1 variant*

```
STLB{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STLB{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 1] = R[t]<7:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

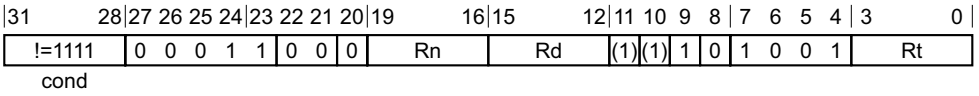
## F6.1.212 STLEX

Store-Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



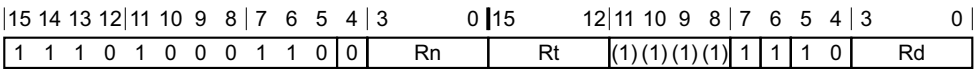

#### *A1 variant*

```
STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemO[address, 4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

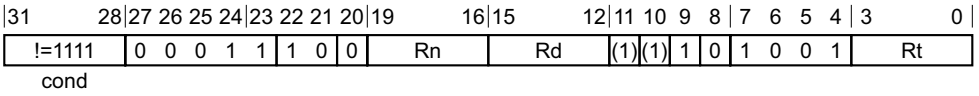
### F6.1.213 STLEXB

Store-Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

#### A1



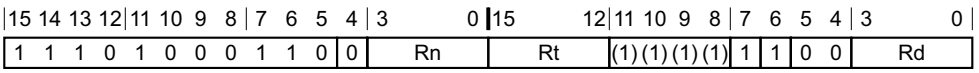

##### *A1 variant*

```
STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

##### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

#### T1




##### *T1 variant*

```
STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

##### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

## Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemO[address, 1] = R[t]<7:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

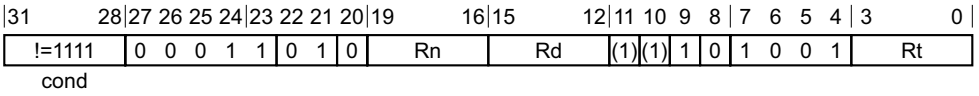
## F6.1.214 STLEXD

Store-Release Exclusive Doubleword stores a doubleword from two registers to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



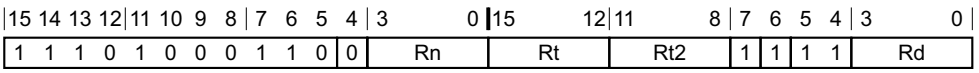

#### *A1 variant*

```
STLEXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);
if d == 15 || Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STLEXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STLEXD* on page K1-5459.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. `<Rt>` must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Rt2>` For encoding A1: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian() then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address, 8) then
        MemO[address, 8] = value;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

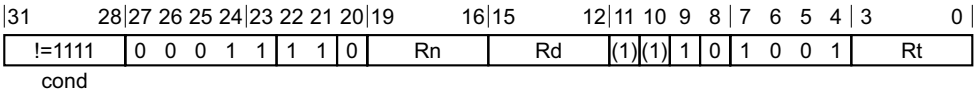
## F6.1.215 STLEXH

Store-Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



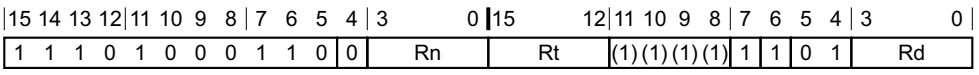

#### *A1 variant*

```
STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemO[address, 2] = R[t]<15:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

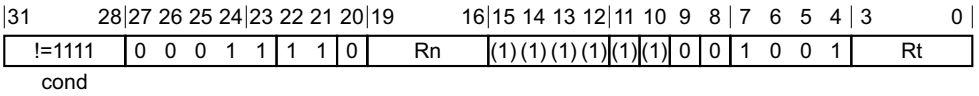

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.216 STLH

Store-Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-89.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



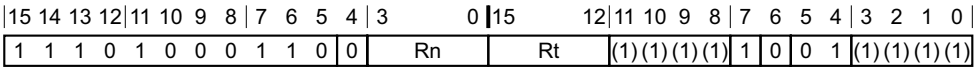

#### *A1 variant*

```
STLH{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STLH{<c>}{<q>} <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 2] = R[t]<15:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

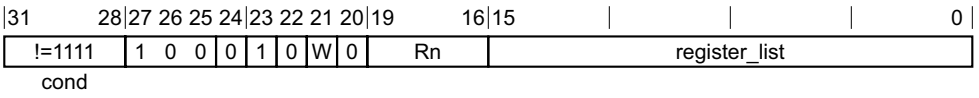
## F6.1.217 STM, STMIA, STMEA

Store Multiple (Increment After, Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

For details of related system instructions see STM (User registers).

### A1



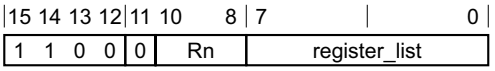

#### *A1 variant*

```
STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMEA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

### T1



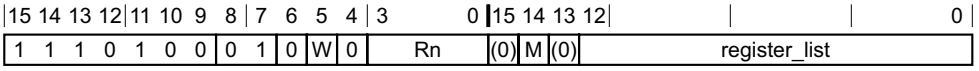

#### *T1 variant*

```
STM{IA}{<c>}{<q>} <Rn>!, <registers> // Preferred syntax
STMEA{<c>}{<q>} <Rn>!, <registers> // Alternate syntax, Empty Ascending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

### T2




#### *T2 variant*

```
STM{IA}{<c>}.W <Rn>{!}, <registers> // Preferred syntax, if <Rn>, '!' and <registers> can be represented
in T1
STMEA{<c>}.W <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack, if <Rn>, '!' and
<registers> can be represented in T1
STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMEA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for this encoding*

```
n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STM (STMIA, STMEA)* on page K1-5460.

## Assembler symbols

`IA` Is an optional suffix for the Increment After form.

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`!` The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

`<registers>` For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, ARM deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;  // Only possible for encodings T1 and A1
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);
```

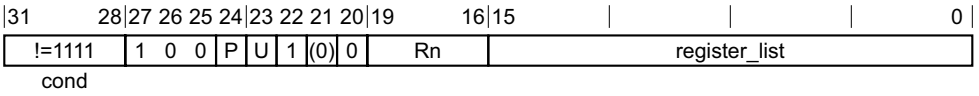

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.218 STM (User registers)

In an EL1 mode other than System mode, Store Multiple (User registers) stores multiple User mode registers to consecutive memory locations using an address from a base register. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Store Multiple (User registers) is UNDEFINED in Hyp mode, and CONSTRAINED UNPREDICTABLE in User or System modes.

#### A1




##### *A1 variant*

```
STM{<amode>}{<c>}{<q>} <Rn>, <registers>^
```

##### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

#### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STM (User registers)* on page K1-5483.

#### Assembler symbols

<amode> is one of:

| | |
|---|---|
| DA | Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0. |
| ED | Empty Descending. For this instruction, a synonym for DA. |
| DB | Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0. |
| FD | Full Descending. For this instruction, a synonym for DB. |
| IA | Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1. |
| EA | Empty Ascending. For this instruction, a synonym for IA. |
| IB | Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1. |
| FA | Full Ascending. For this instruction, a synonym for IB. |

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<registers> Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the STM instruction. The registers are stored with the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;
    else
        length = 4*BitCount(registers);
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;
        for i = 0 to 14
            if registers<i> == '1' then  // Store User mode register
                MemA[address,4] = Rmode[i, M32_User];
                address = address + 4;
        if registers<15> == '1' then
            MemA[address,4] = PCStoreValue();
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

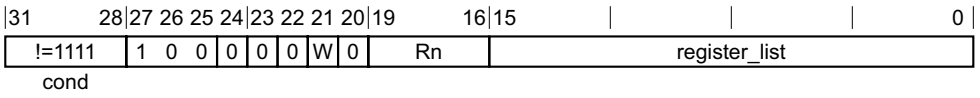
## F6.1.219 STMDA, STMED

Store Multiple Decrement After (Empty Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

For details of related system instructions see STM (User registers).

### A1




#### *A1 variant*

```
STMDA{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMED{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Descending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STMDA (STMED)* on page K1-5462.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| ! | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. |
| <registers> | Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, ARM deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. |

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
            address = address + 4;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if registers<15> == '1' then
    MemA[address,4] = PCStoreValue();
if wback then R[n] = R[n] - 4*BitCount(registers);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.220 STMDB, STMFD

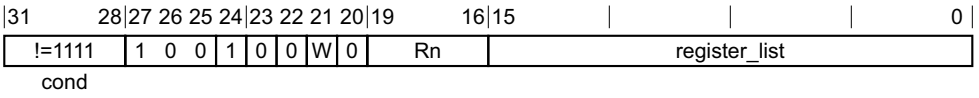
Store Multiple Decrement Before (Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

For details of related system instructions see STM (User registers).

This instruction is used by the alias PUSH (multiple registers). See the *Alias conditions* on page F6-3048 table for details of when each alias is preferred.

### A1



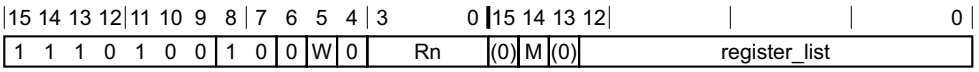

#### *A1 variant*

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STMDB (STMFD)* on page K1-5461.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Alias conditions

| Alias | is preferred when |
|---|---|
| PUSH (multiple registers) | `W == '1' && Rn == '1101' && BitCount(register_list) > 1` |

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`!` The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

`<registers>` For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, ARM deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;  // Only possible for encoding A1
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

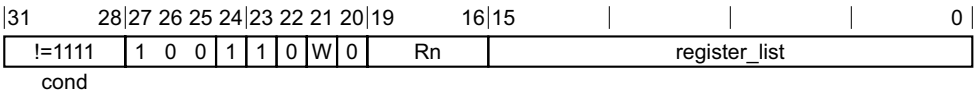
## F6.1.221 STMIB, STMFA

Store Multiple Increment Before (Full Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F2-2453.

For details of related system instructions see STM (User registers).

### A1




#### *A1 variant*

```
STMIB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMFA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Ascending stack
```

#### *Decode for this encoding*

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STMIB (STMFA)* on page K1-5463.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| ! | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. |
| <registers> | Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, ARM deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. |

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
            address = address + 4;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if registers<15> == '1' then
    MemA[address,4] = PCStoreValue();
if wback then R[n] = R[n] + 4*BitCount(registers);
```

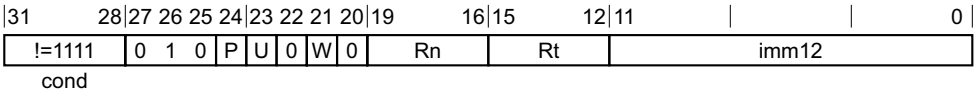

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.222 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

This instruction is used by the alias PUSH (single register). See the *Alias conditions* on page F6-3053 table for details of when each alias is preferred.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
STR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

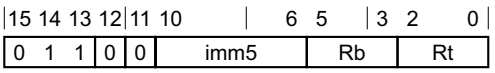
Applies when `P = 1 && W = 1`.

```
STR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE STRT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

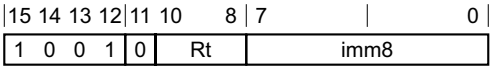
```
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2



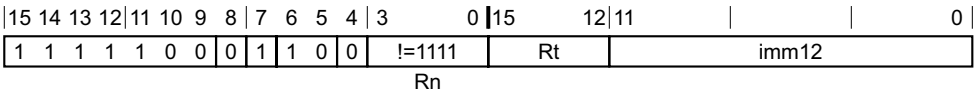

#### *T2 variant*

```
STR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T3



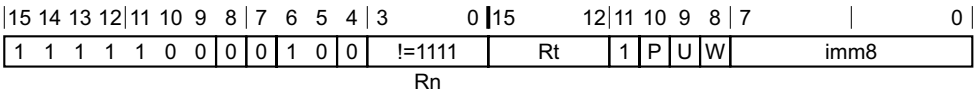

#### *T3 variant*

```
STR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1 or T2
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### *Decode for this encoding*

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE;
```

### T4




#### *Offset variant*

Applies when `P = 1 && U = 0 && W = 0`.

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 1`.

```
STR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.

```
STR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if P == '1' && U == '1' && W == '0' then SEE STRT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STR (immediate, T32)* on page K1-5464 and .

## Alias conditions

| Alias | of variant | is preferred when |
|---|---|---|
| PUSH (single register) | A1 (pre-indexed) | `P == '1' && U == '0' && W == '1' && Rn == '1101' && imm12 == '000000000100'` |
| PUSH (single register) | T4 (pre-indexed) | `Rn == '1101' && U == '0' && imm8 == '00000100'` |

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T1, T2, T3 and T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1 and T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.

For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T4: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

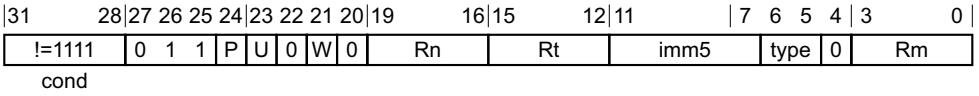
```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,4] = if t == 15 then PCStoreValue() else R[t];
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,4] = R[t];
        if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.223 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
STR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

#### *Pre-indexed variant*

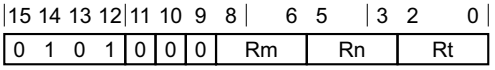
Applies when `P = 1 && W = 1`.

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE STRT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

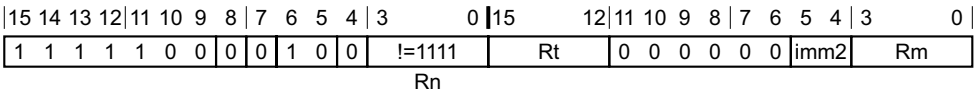
```
STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *T2 variant*

```
STR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

#### *Decode for this encoding*

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STR (register)* on page K1-5465.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the index register is added to the base register.

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see .

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as `0b00`.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    if t == 15 then  // Only possible for encoding A1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

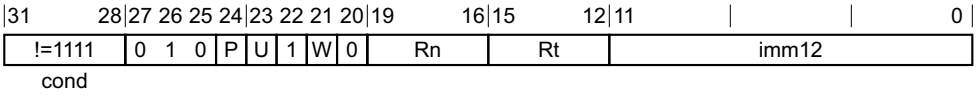
```
    data = PCStoreValue();
else
    data = R[t];
MemU[address,4] = data;
if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.224 STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

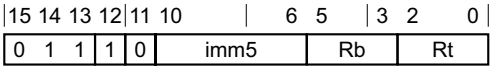
Applies when `P = 1 && W = 1`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE STRBT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1



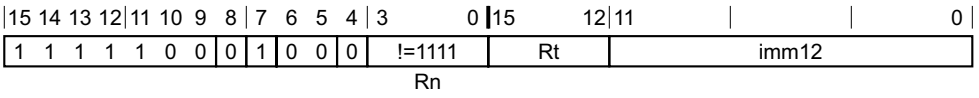

#### *T1 variant*

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T2





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

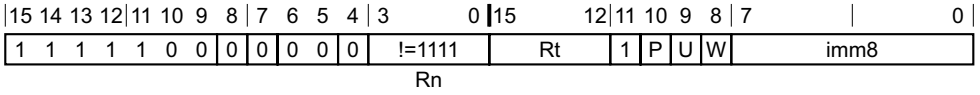
#### T2 variant

```
STRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T3




#### Offset variant

Applies when `P = 1 && U = 0 && W = 0`.

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### Post-indexed variant

Applies when `P = 0 && W = 1`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

Applies when `P = 1 && W = 1`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then SEE STRBT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRB (immediate), T32* on page K1-5466 and .

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.<br>For encoding T1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

## Operation for all encodings

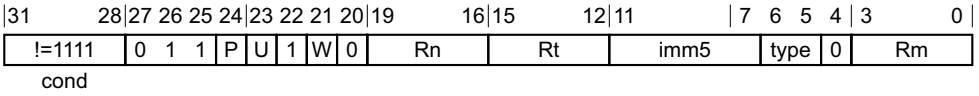
```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,1] = R[t]<7:0>;
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,1] = R[t]<7:0>;
        if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.225 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

#### *Pre-indexed variant*

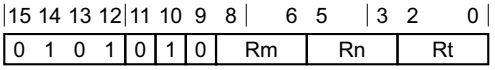
Applies when `P = 1 && W = 1`.

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE STRBT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

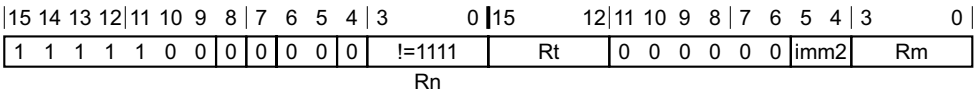
```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *T2 variant*

```
STRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

#### *Decode for this encoding*

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRB (register)* on page K1-5467.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.<br>For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field. |
| +/- | Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <shift> | The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see . |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.226 STRBT

Store Register Byte Unprivileged stores a byte from a register to memory. For information about memory accesses see *Memory accesses* on page F2-2452.

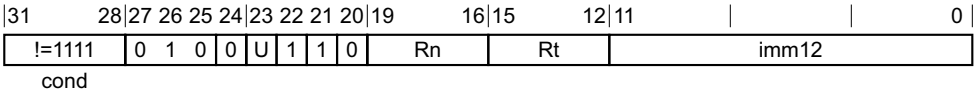
The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

### A1



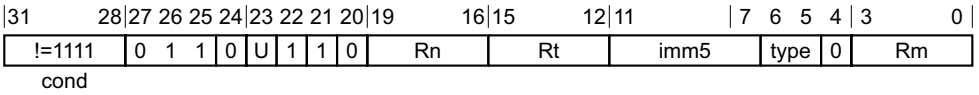

#### *A1 variant*

```
STRBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

### A2



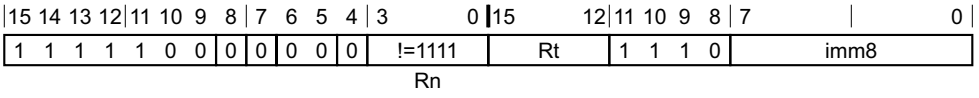

#### *A2 variant*

```
STRBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for this encoding*

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRBT* on page K1-5468.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.<br>For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| +/- | For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1<br>For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <shift> | The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see . |
| + | Specifies the offset is added to the base register. |
| <imm> | For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.<br>For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

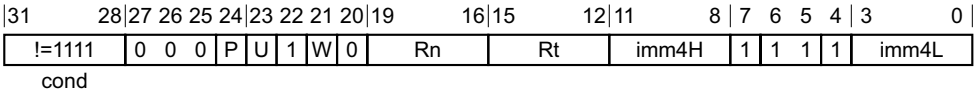
```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    MemU_unpriv[address,1] = R[t]<7:0>;
    if postindex then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.227 STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

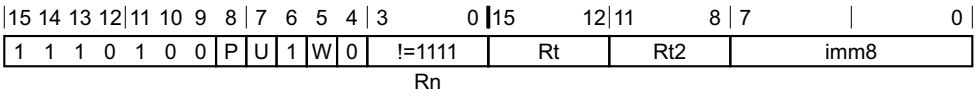
Applies when `P = 1 && W = 1`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

### T1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 1`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if n == 15 || t == 15 || t2 == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRD (immediate)* on page K1-5468.

Related encodings: *Load/store dual, load/store exclusive, table branch* on page F3-2480.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Rt2>` For encoding A1: is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as <imm>/4.

## Operation for all encodings

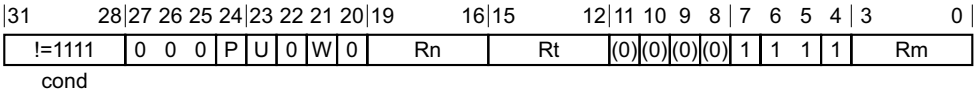
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        bits(64) data;
        if BigEndian() then
            data<63:32> = R[t];
            data<31:0> = R[t2];
        else
            data<31:0> = R[t];
            data<63:32> = R[t2];
        MemA[address,8] = data;
    else
        MemA[address,4] = R[t];
        MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.228 STRD (register)

Store Register Dual (register) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

#### *Pre-indexed variant*

Applies when `P = 1 && W = 1`.

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

#### *Decode for all variants of this encoding*

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRD (register)* on page K1-5470.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

`<Rt2>` Is the second general-purpose register to be transferred. This register must be `<R(t+1)>`.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

`+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when `U = 0`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        bits(64) data;
        if BigEndian() then
            data<63:32> = R[t];
            data<31:0> = R[t2];
        else
            data<31:0> = R[t];
            data<63:32> = R[t2];
        MemA[address,8] = data;
    else
        MemA[address,4] = R[t];
        MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

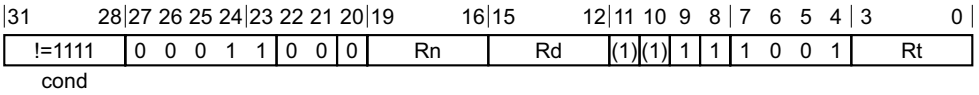

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.229 STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, stores a word from a register to the calculated address if the PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



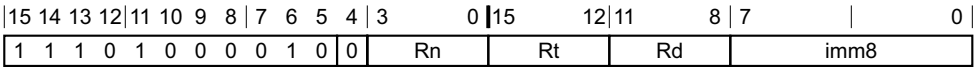

#### *A1 variant*

```
STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, {#}<imm>}]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, #<imm>}]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STREX* on page K1-5471.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<imm> For encoding A1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted.

For encoding T1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

## Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

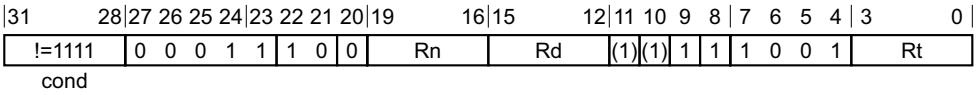

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.230 STREXB

Store Register Exclusive Byte derives an address from a base register value, stores a byte from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



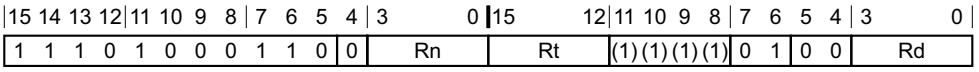

#### *A1 variant*

```
STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STREXB* on page K1-5471.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

## Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemA[address,1] = R[t]<7:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

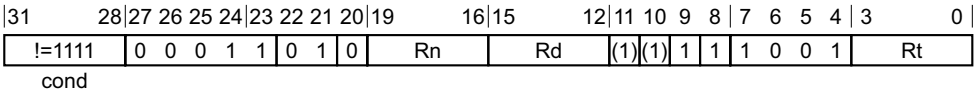

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.231 STREXD

Store Register Exclusive Doubleword derives an address from a base register value, stores a 64-bit doubleword from two registers to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



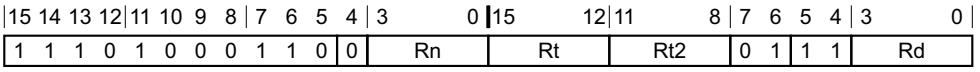

#### *A1 variant*

```
STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);
if d == 15 || Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STREXD* on page K1-5472.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

<Rd> must not be the same as <Rn>, <Rt>, or <Rt2>.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

<Rt2> For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

## Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non doubleword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian() then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address,8) then
        MemA[address,8] = value;  R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

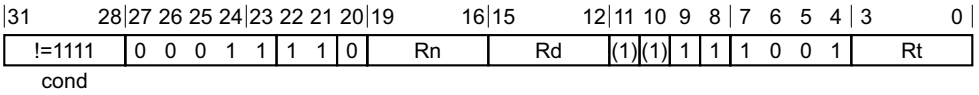

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.232 STREXH

Store Register Exclusive Halfword derives an address from a base register value, stores a halfword from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-2395. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1



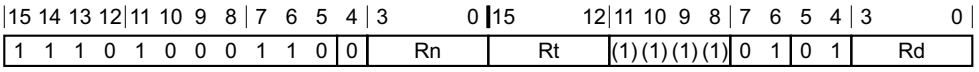

#### *A1 variant*

```
STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

#### *Decode for this encoding*

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STREXH* on page K1-5472.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

### Operation for all encodings

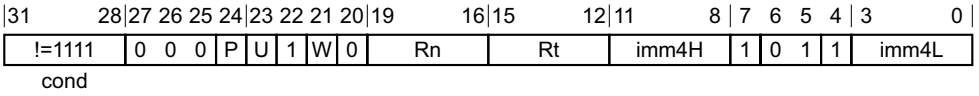
```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t]<15:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.233 STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### *Pre-indexed variant*

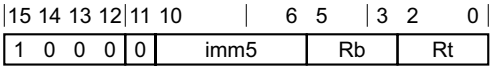
Applies when `P = 1 && W = 1`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE STRHT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1



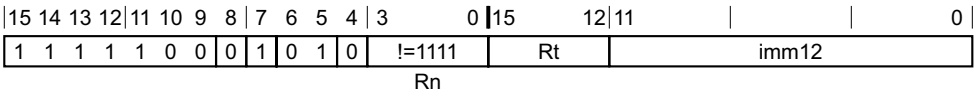

#### *T1 variant*

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T2





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

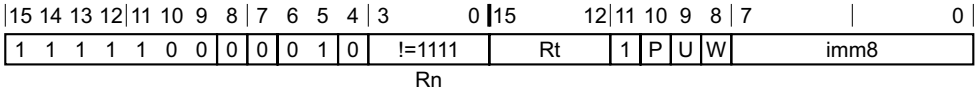
#### T2 variant

```
STRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

#### Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T3




#### Offset variant

Applies when `P = 1 && U = 0 && W = 0`.

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### Post-indexed variant

Applies when `P = 0 && W = 1`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

Applies when `P = 1 && W = 1`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

#### Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then SEE STRHT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRH (immediate, T32)* on page K1-5473 and .

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.<br>For encoding A1, T1, T2, T3: is the general-purpose base register, encoded in the "Rn" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when `U = 0`

+ when `U = 1`

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

## Operation for all encodings

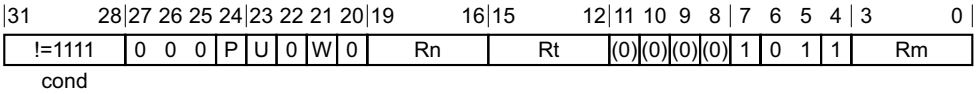
```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,2] = R[t]<15:0>;
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,2] = R[t]<15:0>;
        if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.234 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page F2-2452.

### A1




#### *Offset variant*

Applies when `P = 1 && W = 0`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

#### *Post-indexed variant*

Applies when `P = 0 && W = 0`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

#### *Pre-indexed variant*

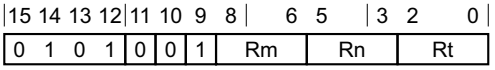
Applies when `P = 1 && W = 1`.

```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

#### *Decode for all variants of this encoding*

```
if P == '0' && W == '1' then SEE STRHT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### T1




#### *T1 variant*

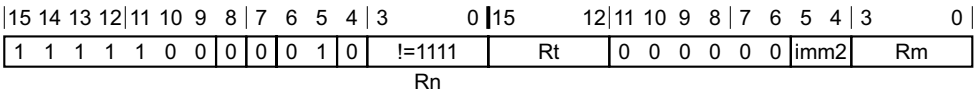
```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *T2 variant*

```
STRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

#### *Decode for this encoding*

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRH (register)* on page K1-5474.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.<br>For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field. |
| +/- | Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;
    if wback then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.235 STRHT

Store Register Halfword Unprivileged stores a halfword from a register to memory. For information about memory accesses see *Memory accesses* on page F2-2452.

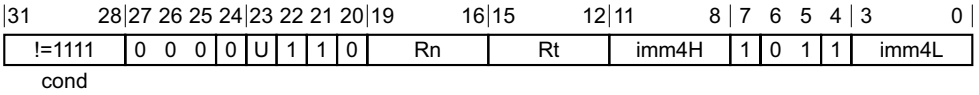
The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

### A1



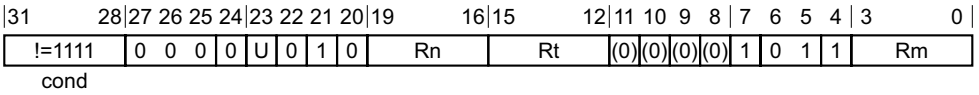

#### *A1 variant*

```
STRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

### A2



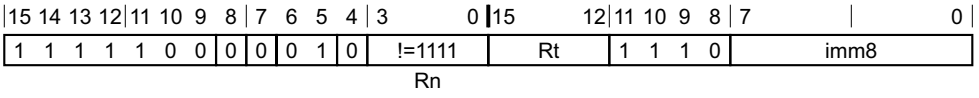

#### *A2 variant*

```
STRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STRHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for this encoding*

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRHT* on page K1-5475.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

`+` Specifies the offset is added to the base register.

`<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

## Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    MemU_unpriv[address,2] = R[t]<15:0>;
    if postindex then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.236 STRT

Store Register Unprivileged stores a word from a register to memory. For information about memory accesses see *Memory accesses* on page F2-2452.

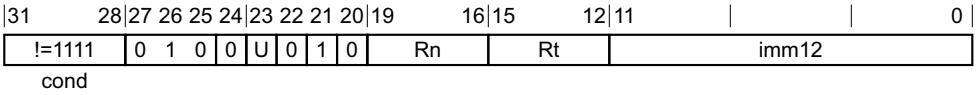
The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

### A1



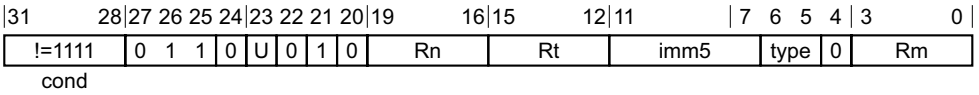

#### *A1 variant*

```
STRT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if n == 15 || n == t then UNPREDICTABLE;
```

### A2



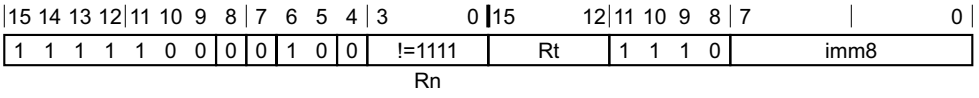

#### *A2 variant*

```
STRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

#### *Decode for this encoding*

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
STRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for this encoding*

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *STRT* on page K1-5476.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rt> | For encoding A1 and A2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.<br>For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| +/- | For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1<br>For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <shift> | The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see . |
| + | Specifies the offset is added to the base register. |
| <imm> | For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.<br>For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    if t == 15 then  // Only possible for encodings A1 and A2
        data = PCStoreValue();
    else
        data = R[t];
    MemU_unpriv[address,4] = data;
    if postindex then R[n] = offset_addr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

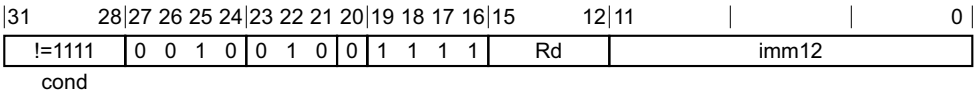
### F6.1.237 SUB (immediate, from PC)

Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. ARM recommends that, where possible, software avoids using this alias

This instruction is an alias of the ADR instruction. This means that:

- The encodings in this description are named to match the encodings of ADR.
- The description of ADR gives the operational pseudocode for this instruction.

#### A2




##### *A2 variant*

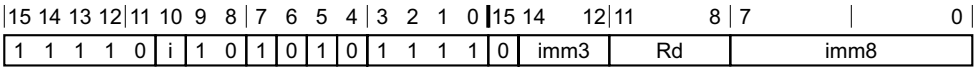
```
SUB{<c>}{<q>} <Rd>, PC, #<const>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is the preferred disassembly when `imm12==000000000000`.

#### T2




##### *T2 variant*

```
SUB{<c>}{<q>} <Rd>, PC, #<imm12>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is the preferred disassembly when `i:imm3:imm8 = '000000000000'`.

#### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` For encoding A2: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field.

`<label>` For encoding A1 and A2: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. If the offset is zero or positive, encoding A1 is used, with `imm32` equal to the offset. If the offset is negative, encoding A2 is used, with `imm32` equal to the size of the offset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

That is, the use of encoding A2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions* on page F2-2456.

For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset. If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are 0-4095.

`<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

`<const>` An immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

## Operation for all encodings

The description of ADR gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.238 SUB, SUBS (immediate)

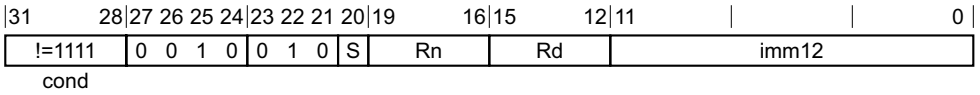
Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The SUBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode, except for encoding T5 with <imm8> set to zero, which is the encoding for the ERET instruction, see ERET.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1




#### *SUB variant*

Applies when S = 0 && Rn != 11x1.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *SUBS variant*

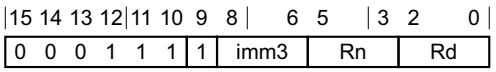
Applies when S = 1 && Rn != 1101.

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### *Decode for all variants of this encoding*

```
if Rn == '1111' && S == '0' then SEE ADR;
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

### T1




#### *T1 variant*

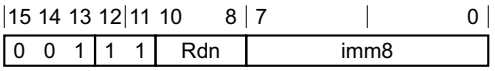
```
SUB<c>{<q>} <Rd>, <Rn>, #<imm3> // Inside IT block
SUBS{<q>} <Rd>, <Rn>, #<imm3> // Outside IT block
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

## T2



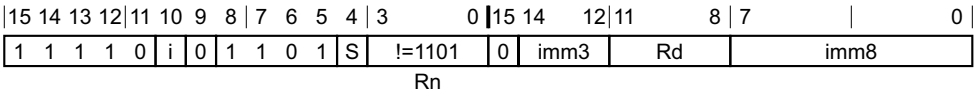

### T2 variant

```
SUB<c>{<q>} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> can be represented in T1
SUB<c>{<q>} {<Rdn>,} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
SUBS{<q>} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> can be represented in T1
SUBS{<q>} {<Rdn>,} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1
```

### Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

## T3




### SUB variant

Applies when `S = 0`.

```
SUB<c>.W {<Rd>,} <Rn>, #<const> // Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or
T2
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### SUBS variant

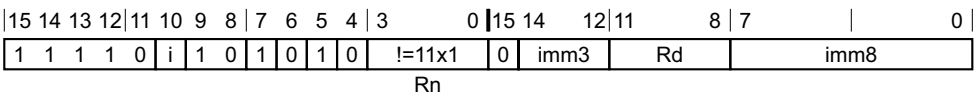
Applies when `S = 1 && Rd != 1111`.

```
SUBS.W {<Rd>,} <Rn>, #<const> // Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
SUBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE CMP (immediate);
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## T4




### T4 variant

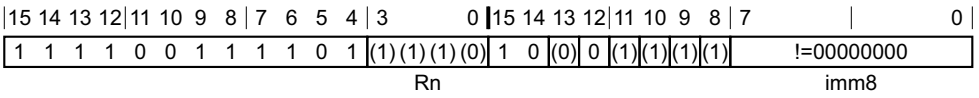
```
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
SUBW{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // <imm12> can be represented in T1, T2, or T3
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### Decode for this encoding

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T5




#### T5 variant

```
SUBS{<c>}{<q>} PC, LR, #<imm8>
```

#### Decode for this encoding

```
if Rn == '1110' && IsZero(imm8) then SEE ERET;
d = 15;  n = UInt(Rn);  setflags = TRUE;  imm32 = ZeroExtend(imm8, 32);
if n != 14 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *SUBS PC. LR and related instructions (A32)* on page K1-5484 and .

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rdn>` Is the general-purpose source and destination register, encoded in the "Rdn" field.

`<imm8>` For encoding T2: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

For encoding T5: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. If `<Rn>` is the LR, and zero is used, see ERET.

`<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. ARM deprecates use of this instruction unless `<Rn>` is the LR.

For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

`<Rn>` For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB, SUBS (SP minus immediate). If the PC is used, see ADR.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB, SUBS (SP minus immediate).


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<imm3>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.

`<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

`<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T3: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

In the T32 instruction set, `MOVS{<c>}{<q>} PC, LR` is a pseudo-instruction for `SUBS{<c>}{<q>} PC, LR, #0`.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), '1');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.239 SUB, SUBS (register)

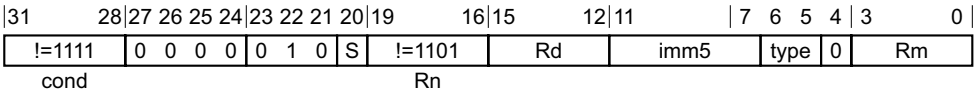
Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The SUBS variant of the instruction performs an exception return without the use of the stack. ARM deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *SUB, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *SUB, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *SUBS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

##### *SUBS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

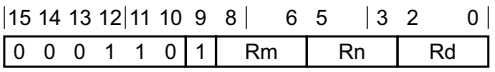
```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
if Rn == '1101' then SEE SUB (SP minus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



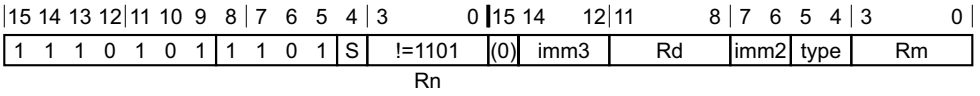

#### *T1 variant*

```
SUB<c>{<q>} <Rd>, <Rn>, <Rm> // Inside IT block
SUBS{<q>} {<Rd>,} <Rn>, <Rm> // Outside IT block
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *SUB, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *SUB, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
SUB<c>.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *SUBS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && Rd != 1111 && imm2 = 00 && type = 11`.

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### *SUBS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11) && Rd != 1111`.

```
SUBS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
if Rd == '1111' && S == '1' then SEE CMP (register);
if Rn == '1101' then SEE SUB (SP minus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used. If the SP is used, see SUB, SUBS (SP minus register).

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB, SUBS (SP minus register).

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| LSL | when type = 00 |
| LSR | when type = 01 |
| ASR | when type = 10 |
| ROR | when type = 11 |

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

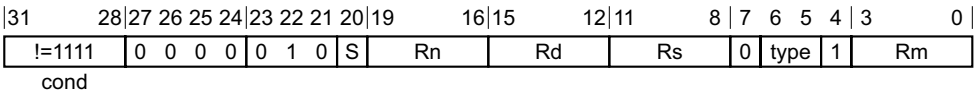
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.240 SUB, SUBS (register-shifted register)

Subtract (register-shifted register) subtracts a register-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

#### *Not flag setting variant*

Applies when `S = 0`.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<type>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.241 SUB, SUBS (SP minus immediate)

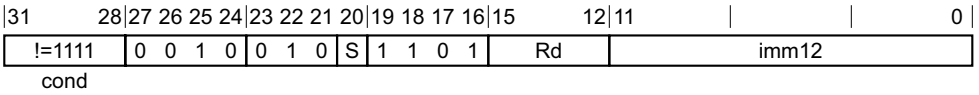
Subtract from SP (immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The SUBS variant of the instruction performs an exception return without the use of the stack. ARM deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *SUB variant*

Applies when S = 0.

```
SUB{<c>}{<q>} {<Rd>,} SP, #<const>
```

##### *SUBS variant*

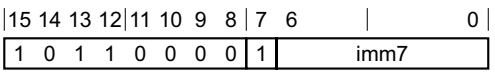
Applies when S = 1.

```
SUBS{<c>}{<q>} {<Rd>,} SP, #<const>
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

#### T1




##### *T1 variant*

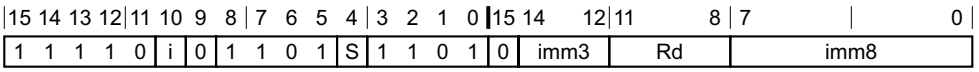
```
SUB{<c>}{<q>} {SP,} SP, #<imm7>
```

##### *Decode for this encoding*

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *SUB variant*

Applies when `S = 0`.

```
SUB{<c>}.W {<Rd>,} SP, #<const> // <Rd>, <const> can be represented in T1
SUB{<c>}{<q>} {<Rd>,} SP, #<const>
```

#### *SUBS variant*

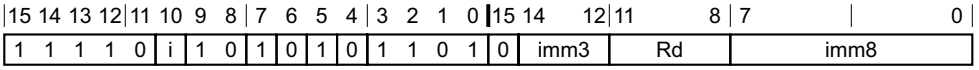
Applies when `S = 1 && Rd != 1111`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, #<const>
```

#### *Decode for all variants of this encoding*

```
if Rd == '1111' && S == '1' then SEE CMP (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && !setflags then UNPREDICTABLE;
```

### T3




#### *T3 variant*

```
SUB{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
SUBW{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> can be represented in T1, T2, or T3
```

#### *Decode for this encoding*

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<imm7> Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. ARM deprecates use of this instruction unless <Rn> is the LR.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(SP, NOT(imm32), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.242 SUB, SUBS (SP minus register)

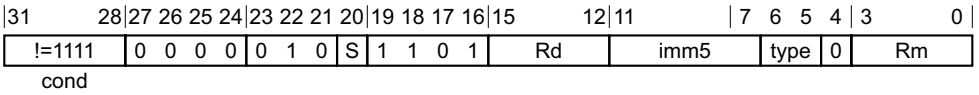
Subtract from SP (register) subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- The SUBS variant of the instruction performs an exception return without the use of the stack. ARM deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-3797.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

#### A1




##### *SUB, rotate right with extend variant*

Applies when `S = 0 && imm5 = 00000 && type = 11`.

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

##### *SUB, shift or rotate by value variant*

Applies when `S = 0 && !(imm5 == 00000 && type == 11)`.

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

##### *SUBS, rotate right with extend variant*

Applies when `S = 1 && imm5 = 00000 && type = 11`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

##### *SUBS, shift or rotate by value variant*

Applies when `S = 1 && !(imm5 == 00000 && type == 11)`.

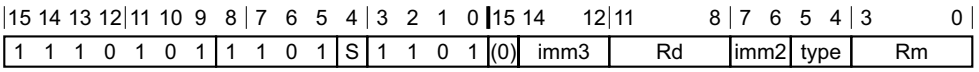
```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *SUB, rotate right with extend variant*

Applies when `S = 0 && imm3 = 000 && imm2 = 00 && type = 11`.

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

#### *SUB, shift or rotate by value variant*

Applies when `S = 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
SUB{<c>}.W {<Rd>,} SP, <Rm> // <Rd>, <Rm> can be represented in T1 or T2
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

#### *SUBS, rotate right with extend variant*

Applies when `S = 1 && imm3 = 000 && Rd != 1111 && imm2 = 00 && type = 11`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

#### *SUBS, shift or rotate by value variant*

Applies when `S = 1 && !(imm3 == 000 && imm2 == 00 && type == 11) && Rd != 1111`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
if Rd == '1111' && S == '1' then SEE CMP (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if (d == 15 && !setflags) || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. ARM deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

LSL when type = 00

LSR when type = 01

ASR when type = 10

ROR when type = 11

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(SP, NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.243 SVC

Supervisor Call causes a Supervisor Call exception. For more information, see .

**Note**

SVC was previously called SWI, Software Interrupt, and this name is still found in some documentation.

Software can use this instruction as a call to an operating system to provide a service.

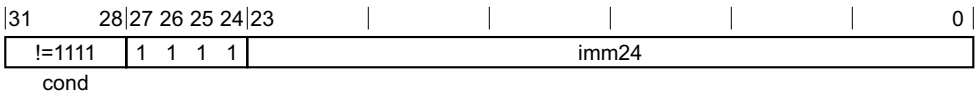
In the following cases, the Supervisor Call exception generated by the SVC instruction is taken to Hyp mode:

- If the SVC is executed in Hyp mode.
- If HCR.TGE is set to 1, and the SVC is executed in Non-secure User mode. For more information, see

In these cases, the HSR identifies that the exception entry was caused by a Supervisor Call exception, EC value 0x11, see *Use of the HSR* on page G4-4105. The immediate field in the HSR:

- If the SVC is unconditional:
  - For the T32 instruction, is the zero-extended value of the imm8 field.
  - For the A32 instruction, is the least-significant 16 bits the imm24 field.
- If the SVC is conditional, is UNKNOWN.

### A1



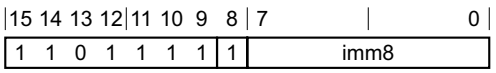

#### *A1 variant*

```
SVC{<c>}{<q>} {#}<imm>
```

#### *Decode for this encoding*

```
imm32 = ZeroExtend(imm24, 32);
// imm32 is for assembly/disassembly. SVC handlers in some
// systems interpret imm24 in software, for example to determine the required service.
```

### T1




#### *T1 variant*

```
SVC{<c>}{<q>} {#}<imm>
```

#### *Decode for this encoding*

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<imm> For encoding A1: is a 24-bit unsigned immediate, in the range 0 to 16777215, encoded in the "imm24" field.

For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

### Operation for all encodings

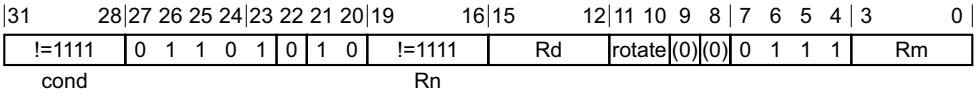
```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32.CallSupervisor(imm32<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.244 SXTAB

Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### A1



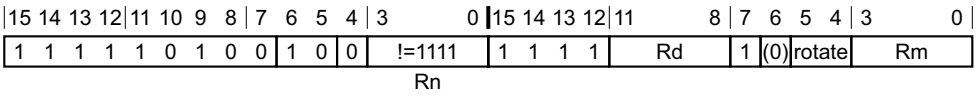

#### *A1 variant*

```
SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE SXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE SXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)` when `rotate = 00`

`8` when `rotate = 01`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| | |
|---|---|
| 16 | when rotate = 10 |
| 24 | when rotate = 11 |

### Operation for all encodings

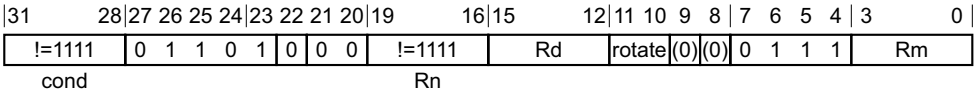
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<7:0>, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.245 SXTAB16

Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### A1



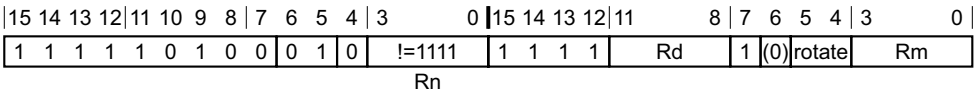

#### *A1 variant*

```
SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE SXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE SXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)` when `rotate = 00`

`8` when `rotate = 01`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16 when rotate = 10

24 when rotate = 11

## Operation for all encodings

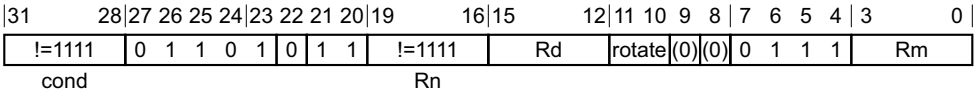
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = R[n]<15:0> + SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + SignExtend(rotated<23:16>, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.246 SXTAH

Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

### A1



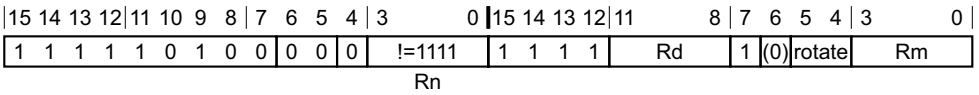

#### *A1 variant*

```
SXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE SXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE SXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`(omitted)` when `rotate = 00`

`8` when `rotate = 01`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| | |
|---|---|
| 16 | when rotate = 10 |
| 24 | when rotate = 11 |

### Operation for all encodings

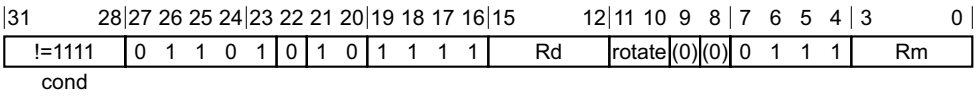
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<15:0>, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.247 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### A1



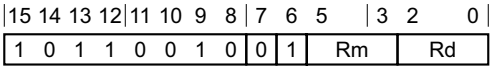

#### *A1 variant*

```
SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1



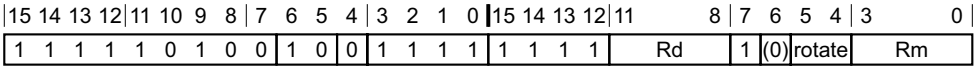

#### *T1 variant*

```
SXTB{<c>}{<q>} {<Rd>,} <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

### T2




#### *T2 variant*

```
SXTB{<c>}.W {<Rd>,} <Rm> // <Rd>, <Rm> can be represented in T1
SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

| | |
|---|---|
| `(omitted)` | when `rotate = 00` |
| `8` | when `rotate = 01` |
| `16` | when `rotate = 10` |
| `24` | when `rotate = 11` |

### Operation for all encodings

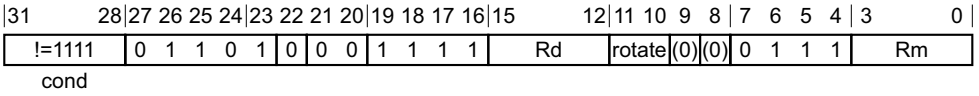
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.248 SXTB16

Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### A1



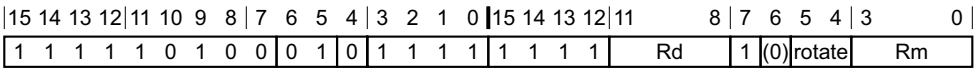

#### *A1 variant*

```
SXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
SXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

- `(omitted)` when `rotate = 00`
- `8` when `rotate = 01`
- `16` when `rotate = 10`
- `24` when `rotate = 11`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

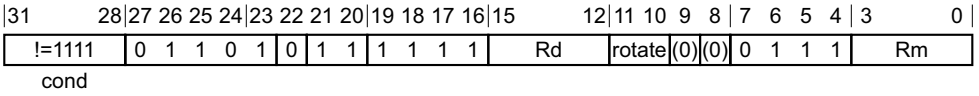
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = SignExtend(rotated<23:16>, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.249 SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

### A1



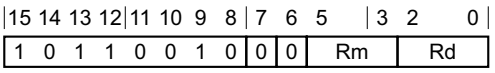

#### *A1 variant*

```
SXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1



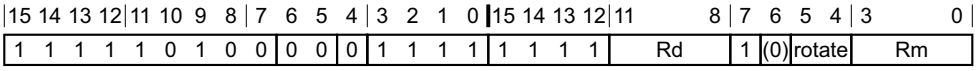

#### *T1 variant*

```
SXTH{<c>}{<q>} {<Rd>,} <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

### T2




#### *T2 variant*

```
SXTH{<c>}.W {<Rd>,} <Rm> // <Rd>, <Rm> can be represented in T1
SXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

| | |
|---|---|
| `(omitted)` | when `rotate = 00` |
| `8` | when `rotate = 01` |
| `16` | when `rotate = 10` |
| `24` | when `rotate = 11` |

### Operation for all encodings

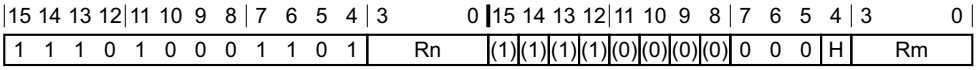
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.250 TBB, TBH

Table Branch Byte or Halfword causes a PC-relative forward branch using a table of single byte or halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value returned from the table.

### T1




#### *Byte variant*

Applies when `H = 0`.

```
TBB{<c>}{<q>} [<Rn>, <Rm>] // Outside or last in IT block
```

#### *Halfword variant*

Applies when `H = 1`.

```
TBH{<c>}{<q>} [<Rn>, <Rm>, LSL #1] // Outside or last in IT block
```

#### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);  is_tbh = (H == '1');
if m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` Is the general-purpose base register holding the address of the table of branch lengths, encoded in the "Rn" field. The PC can be used. If it is, the table immediately follows this instruction.

`<Rm>` For the byte variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.

For the halfword variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

### Operation

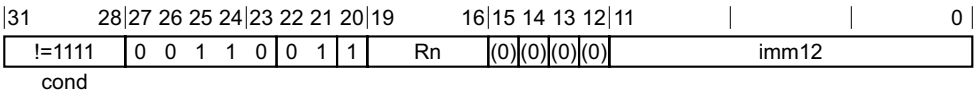
```
if ConditionPassed() then
    EncodingSpecificOperations();
    if is_tbh then
        halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
    else
        halfwords = UInt(MemU[R[n]+R[m], 1]);
    BranchWritePC(PC + 2*halfwords);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.251 TEQ (immediate)

Test Equivalence (immediate) performs a bitwise exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

### A1



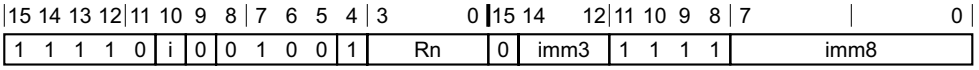

#### *A1 variant*

```
TEQ{<c>}{<q>} <Rn>, #<const>
```

#### *Decode for this encoding*

```
n = UInt(Rn);
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

### T1




#### *T1 variant*

```
TEQ{<c>}{<q>} <Rn>, #<const>
```

#### *Decode for this encoding*

```
n = UInt(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

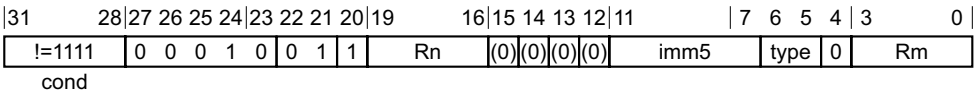
```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F6.1.252 TEQ (register)

Test Equivalence (register) performs a bitwise exclusive OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

#### A1




##### *Rotate right with extend variant*

Applies when `imm5 = 00000 && type = 11`.

```
TEQ{<c>}{<q>} <Rn>, <Rm>, RRX
```

##### *Shift or rotate by value variant*

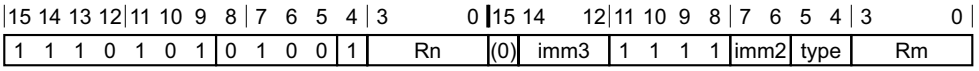
Applies when `!(imm5 == 00000 && type == 11)`.

```
TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

#### T1




##### *Rotate right with extend variant*

Applies when `imm3 = 000 && imm2 = 00 && type = 11`.

```
TEQ{<c>}{<q>} <Rn>, <Rm>, RRX
```

##### *Shift or rotate by value variant*

Applies when `!(imm3 == 000 && imm2 == 00 && type == 11)`.

```
TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

##### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

LSL when type = 00

LSR when type = 01

ASR when type = 10

ROR when type = 11

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

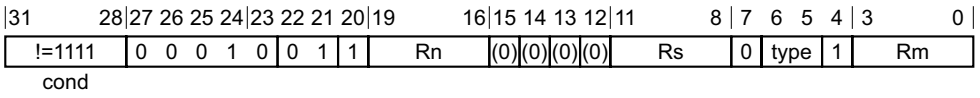
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.253 TEQ (register-shifted register)

Test Equivalence (register-shifted register) performs a bitwise exclusive OR operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

### A1




#### *A1 variant*

```
TEQ{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

#### *Decode for this encoding*

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<type> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| LSL | when type = 00 |
| LSR | when type = 01 |
| ASR | when type = 10 |
| ROR | when type = 11 |

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

### Operation

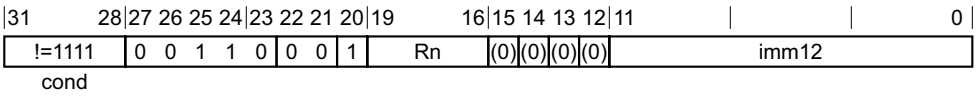
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.254 TST (immediate)

Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

### A1



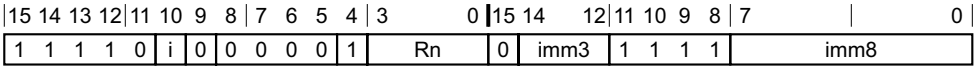

#### *A1 variant*

```
TST{<c>}{<q>} <Rn>, #<const>
```

#### *Decode for this encoding*

```
n = UInt(Rn);
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

### T1




#### *T1 variant*

```
TST{<c>}{<q>} <Rn>, #<const>
```

#### *Decode for this encoding*

```
n = UInt(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rn> | For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used.<br>For encoding T1: is the general-purpose source register, encoded in the "Rn" field. |
| <const> | For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* on page F2-2456 for the range of values.<br>For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* on page F2-2459 for the range of values. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

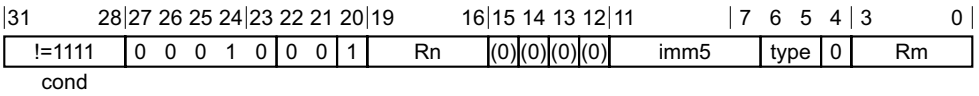
```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.255 TST (register)

Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

### A1




#### *Rotate right with extend variant*

Applies when `imm5 = 00000 && type = 11`.

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

#### *Shift or rotate by value variant*

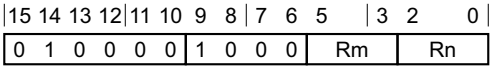
Applies when `!(imm5 == 00000 && type == 11)`.

```
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

#### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

### T1



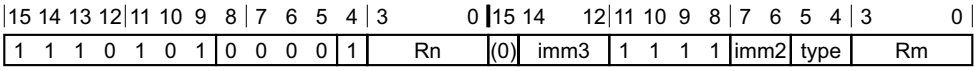

#### *T1 variant*

```
TST{<c>}{<q>} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2




#### *Rotate right with extend variant*

Applies when `imm3 = 000 && imm2 = 00 && type = 11`.

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

#### *Shift or rotate by value variant*

Applies when `!(imm3 == 000 && imm2 == 00 && type == 11)`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
TST{<c>}.W <Rn>, <Rm> // <Rn>, <Rm> can be represented in T1
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

### *Decode for all variants of this encoding*

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used.

For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.

`<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| `LSL` | when `type = 00` |
| `LSR` | when `type = 01` |
| `ASR` | when `type = 10` |
| `ROR` | when `type = 11` |

`<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

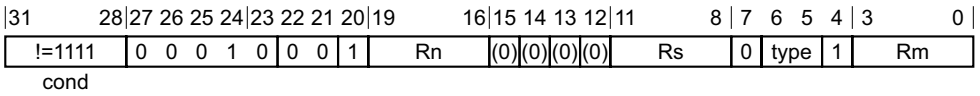
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.256 TST (register-shifted register)

Test (register-shifted register) performs a bitwise AND operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

### A1




#### *A1 variant*

```
TST{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

#### *Decode for this encoding*

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<type> Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

| | |
|---|---|
| LSL | when type = 00 |
| LSR | when type = 01 |
| ASR | when type = 10 |
| ROR | when type = 11 |

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

### Operation

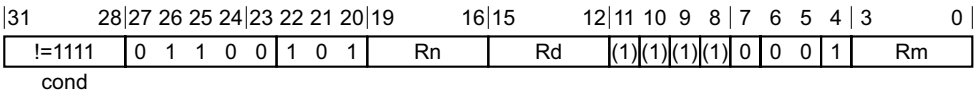
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.257 UADD16

Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

### A1



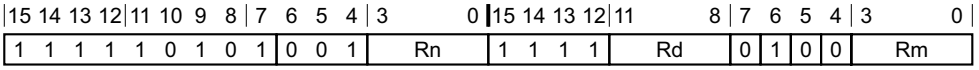

#### *A1 variant*

```
UADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

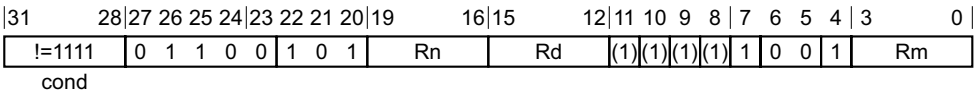
```
R[d]<31:16> = sum2<15:0>;
PSTATE.GE<1:0> = if sum1 >= 0x10000 then '11' else '00';
PSTATE.GE<3:2> = if sum2 >= 0x10000 then '11' else '00';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.258 UADD8

Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

### A1



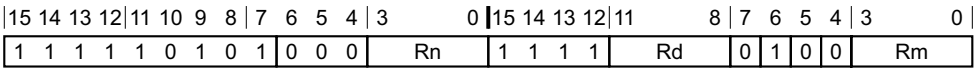

#### *A1 variant*

```
UADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m  == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

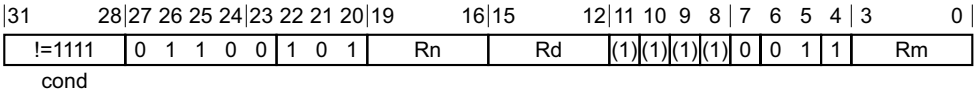
```
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
R[d]<7:0>   = sum1<7:0>;
R[d]<15:8>  = sum2<7:0>;
R[d]<23:16> = sum3<7:0>;
R[d]<31:24> = sum4<7:0>;
PSTATE.GE<0>  = if sum1 >= 0x100 then '1' else '0';
PSTATE.GE<1>  = if sum2 >= 0x100 then '1' else '0';
PSTATE.GE<2>  = if sum3 >= 0x100 then '1' else '0';
PSTATE.GE<3>  = if sum4 >= 0x100 then '1' else '0';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.259 UASX

Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets PSTATE.GE according to the results.

### A1



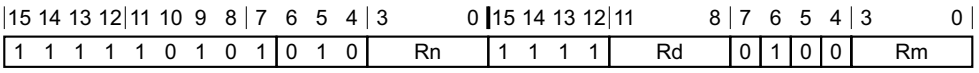

#### *A1 variant*

```
UASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

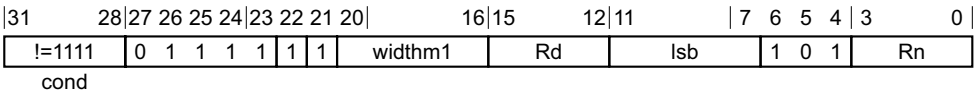
```
R[d]<15:0>  = diff<15:0>;
R[d]<31:16> = sum<15:0>;
PSTATE.GE<1:0> = if diff >= 0 then '11' else '00';
PSTATE.GE<3:2> = if sum  >= 0x10000 then '11' else '00';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.260 UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from a register, zero-extends them to 32 bits, and writes the result to the destination register.

### A1



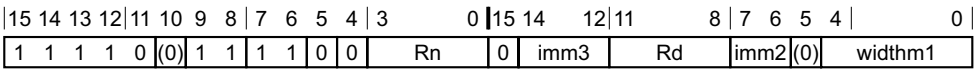

#### A1 variant

```
UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

#### Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### T1 variant

```
UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

#### Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *UBFX* on page K1-5476.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <lsb> | For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.<br>For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field. |
| <width> | Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

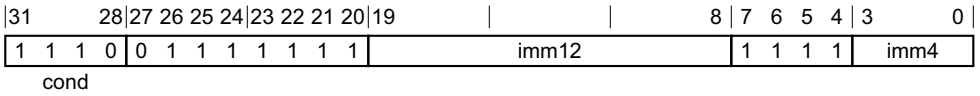
## F6.1.261 UDF

Permanently Undefined generates an Undefined Instruction exception.

The encodings for UDF used in this section are defined as permanently UNDEFINED in the ARMv8-A architecture. However:

- With the T32 instruction set, ARM deprecates using the UDF instruction in an IT block.
- In the A32 instruction set, UDF is not conditional.

### A1



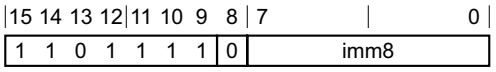

#### *A1 variant*

```
UDF{<c>}{<q>} {#}<imm>
```

#### *Decode for this encoding*

```
imm32 = ZeroExtend(imm12:imm4, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

### T1



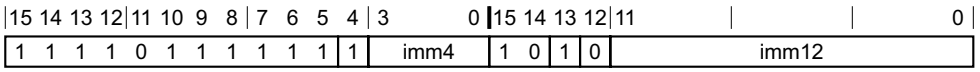

#### *T1 variant*

```
UDF{<c>}{<q>} {#}<imm>
```

#### *Decode for this encoding*

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

### T2




#### *T2 variant*

```
UDF{<c>}.W {#}<imm> // <imm> can be represented in T1
UDF{<c>}{<q>} {#}<imm>
```

#### *Decode for this encoding*

```
imm32 = ZeroExtend(imm4:imm12, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> For encoding A1: see *Standard assembler syntax fields* on page F2-2446. <c> must be AL or omitted.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F2-2446. ARM deprecates using any <c> value other than AL.

<q> See *Standard assembler syntax fields* on page F2-2446.

<imm> For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. The PE ignores the value of this constant.

For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores the value of this constant.

For encoding T2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. The PE ignores the value of this constant.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    UNDEFINED;
```

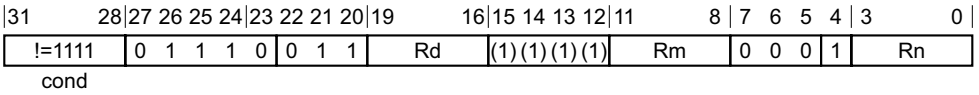

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.262 UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

See  for more information about this instruction.

### A1



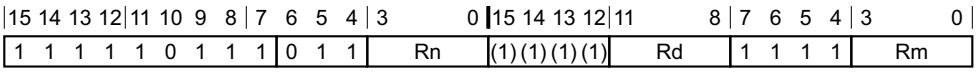

#### *A1 variant*

```
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *UDIV* on page K1-5477.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the dividend, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the divisor, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[m]) == 0 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

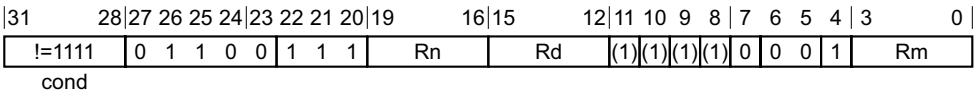
```
    if IntegerZeroDivideTrappingEnabled() then
        GenerateIntegerZeroDivide();
    else
        result = 0;
else
    result = RoundTowardsZero(Real(UInt(R[n])) / Real(UInt(R[m])));
R[d] = result<31:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.263 UHADD16

Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

### A1



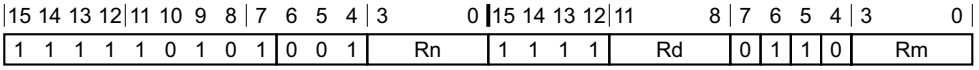

#### *A1 variant*

```
UHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

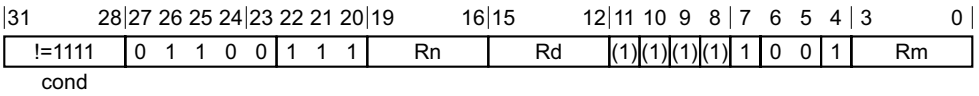
```
sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
R[d]<15:0>  = sum1<16:1>;
R[d]<31:16> = sum2<16:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.264 UHADD8

Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

### A1



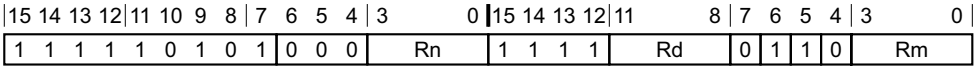

#### *A1 variant*

```
UHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

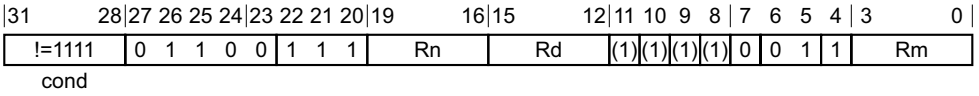
```
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
R[d]<7:0>   = sum1<8:1>;
R[d]<15:8>  = sum2<8:1>;
R[d]<23:16> = sum3<8:1>;
R[d]<31:24> = sum4<8:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.265 UHASX

Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

### A1



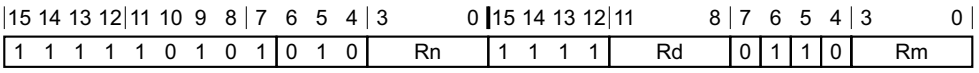

#### *A1 variant*

```
UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

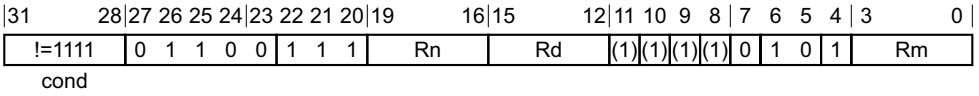
```
sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
R[d]<15:0>  = diff<16:1>;
R[d]<31:16> = sum<16:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.266 UHSAX

Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

### A1



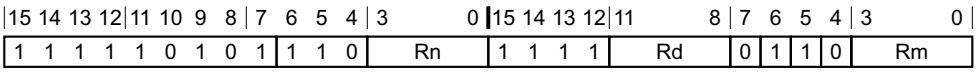

#### *A1 variant*

```
UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

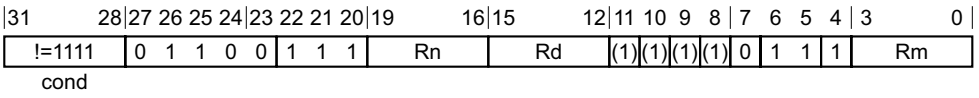
```
diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
R[d]<15:0>  = sum<16:1>;
R[d]<31:16> = diff<16:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.267 UHSUB16

Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

### A1



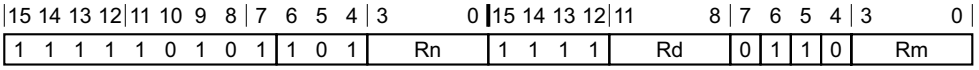

#### *A1 variant*

```
UHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

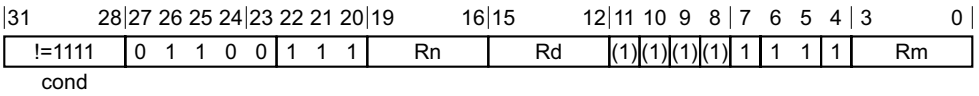
```
diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
R[d]<15:0>  = diff1<16:1>;
R[d]<31:16> = diff2<16:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.268 UHSUB8

Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

### A1



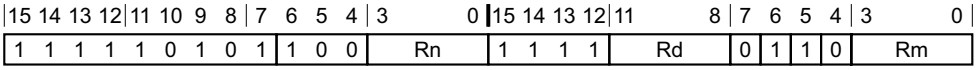

#### *A1 variant*

```
UHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

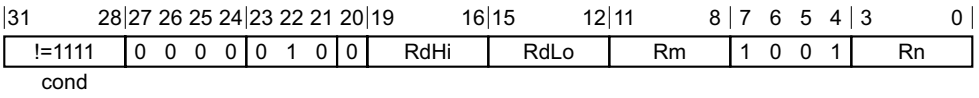
```
diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
R[d]<7:0>   = diff1<8:1>;
R[d]<15:8>  = diff2<8:1>;
R[d]<23:16> = diff3<8:1>;
R[d]<31:24> = diff4<8:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.269 UMAAL

Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

### A1



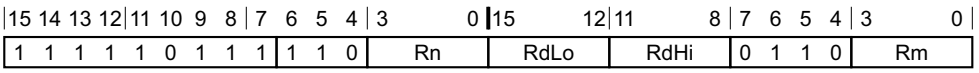

#### *A1 variant*

```
UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *UMAAL* on page K1-5477.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <RdLo> | Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R

[dHi]) + UInt(R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

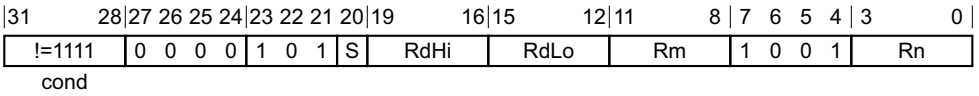

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.270 UMLAL, UMLALS

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
UMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Not flag setting variant*

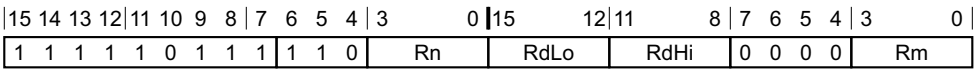
Applies when `S = 0`.

```
UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *UMLAL* on page K1-5477.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<RdLo> Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();
     result = UInt(R[n]) * UInt(R[m]) + UInt(R

[dHi]:R[dLo]);
     R[dHi] = result<63:32>;
     R[dLo] = result<31:0>;
     if setflags then
         PSTATE.N = result<63>;
         PSTATE.Z = IsZeroBit(result<63:0>);
         // PSTATE.C, PSTATE.V unchanged
```

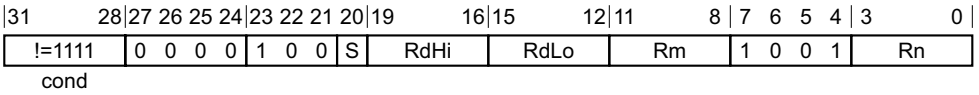

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.271 UMULL, UMULLS

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

### A1




#### *Flag setting variant*

Applies when `S = 1`.

```
UMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Not flag setting variant*

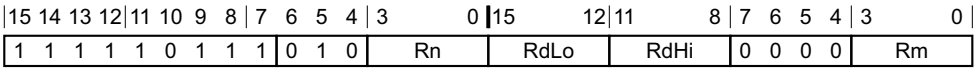
Applies when `S = 0`.

```
UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for all variants of this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

#### *Decode for this encoding*

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// ARMv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *UMULL* on page K1-5478.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<RdLo> Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation for all encodings

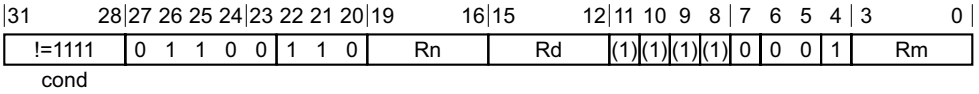
```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.272 UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range 0 <= x <= $2^{16}$ - 1, and writes the results to the destination register.

### A1



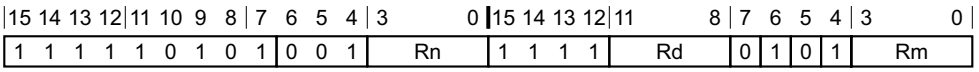

#### *A1 variant*

```
UQADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UQADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

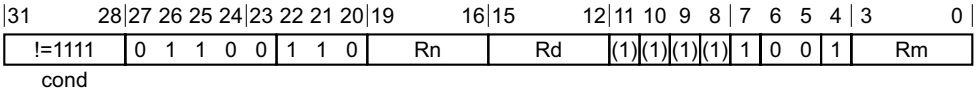
```
sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
R[d]<15:0>  = UnsignedSat(sum1, 16);
R[d]<31:16> = UnsignedSat(sum2, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.273 UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range 0 <= x <= $2^8$ - 1, and writes the results to the destination register.

### A1



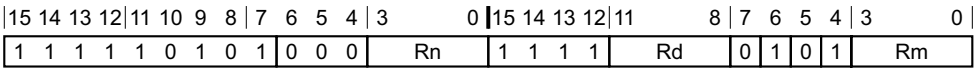

#### *A1 variant*

```
UQADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UQADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

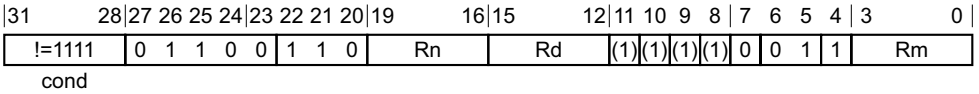
```
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
R[d]<7:0>   = UnsignedSat(sum1, 8);
R[d]<15:8>  = UnsignedSat(sum2, 8);
R[d]<23:16> = UnsignedSat(sum3, 8);
R[d]<31:24> = UnsignedSat(sum4, 8);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.274 UQASX

Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range $0 <= x <= 2^{16} - 1$, and writes the results to the destination register.

### A1



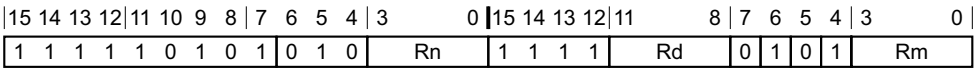

#### *A1 variant*

```
UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

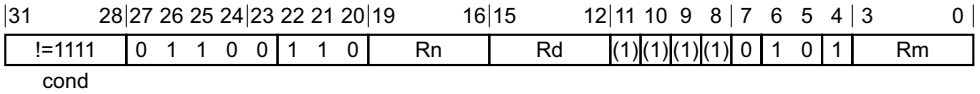
```
sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
R[d]<15:0>  = UnsignedSat(diff, 16);
R[d]<31:16> = UnsignedSat(sum, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.275 UQSAX

Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range $0 <= x <= 2^{16} - 1$, and writes the results to the destination register.

### A1



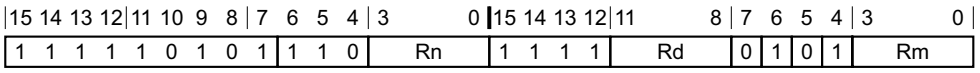

#### *A1 variant*

```
UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

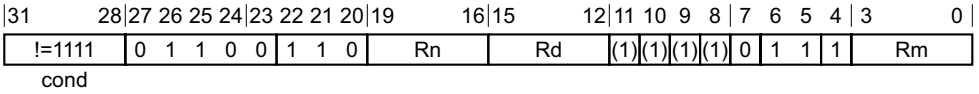
```
diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
R[d]<15:0>  = UnsignedSat(sum, 16);
R[d]<31:16> = UnsignedSat(diff, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.276 UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range 0 <= x <= $2^{16}$ - 1, and writes the results to the destination register.

### A1



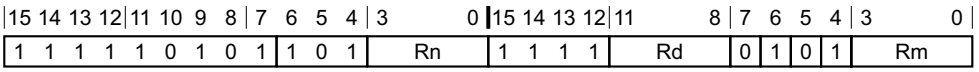

#### *A1 variant*

```
UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

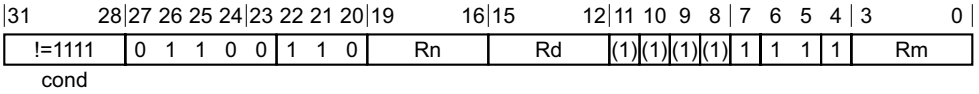
```
diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
R[d]<15:0>  = UnsignedSat(diff1, 16);
R[d]<31:16> = UnsignedSat(diff2, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.277 UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range 0 <= x <= $2^8$ - 1, and writes the results to the destination register.

### A1



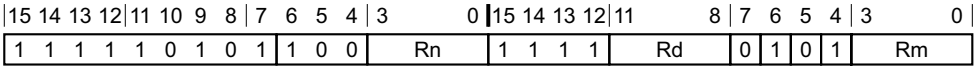

#### *A1 variant*

```
UQSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UQSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

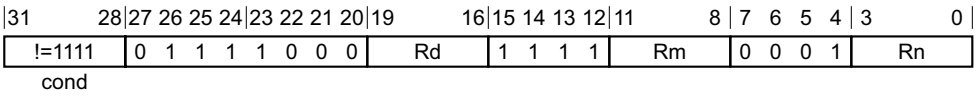
```
diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
R[d]<7:0>   = UnsignedSat(diff1, 8);
R[d]<15:8>  = UnsignedSat(diff2, 8);
R[d]<23:16> = UnsignedSat(diff3, 8);
R[d]<31:24> = UnsignedSat(diff4, 8);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.278 USAD8

Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

### A1



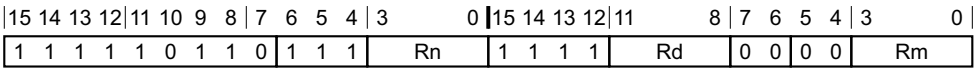

#### *A1 variant*

```
USAD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
USAD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

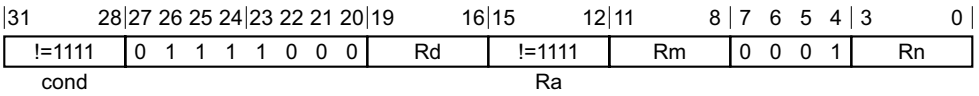
```
absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
R[d] = result<31:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.279 USADA8

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

### A1



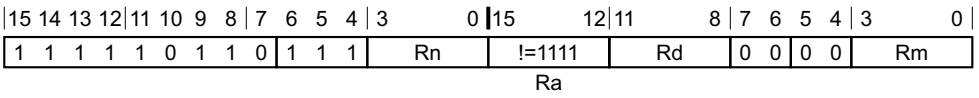

#### *A1 variant*

```
USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for this encoding*

```
if Ra == '1111' then SEE USAD8;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

#### *Decode for this encoding*

```
if Ra == '1111' then SEE USAD8;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

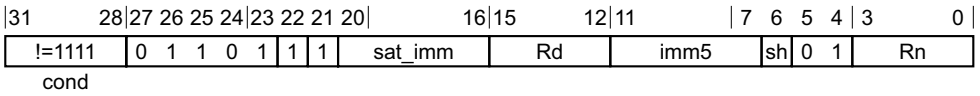

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.280 USAT

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

### A1




#### *Arithmetic shift right variant*

Applies when `sh = 1`.

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

#### *Logical shift left variant*

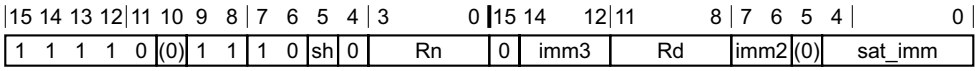
Applies when `sh = 0`.

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

#### *Decode for all variants of this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *Arithmetic shift right variant*

Applies when `sh = 1 && !(imm3 == 000 && imm2 == 00)`.

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

#### *Logical shift left variant*

Applies when `sh = 0`.

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

#### *Decode for all variants of this encoding*

```
if sh == '1' && (imm3:imm2) == '00000' then SEE USAT16;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<imm> Is the bit position for saturation, in the range 0 to 31, encoded in the "sat_imm" field.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

<amount> For encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.

For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
    R[d] = ZeroExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
```

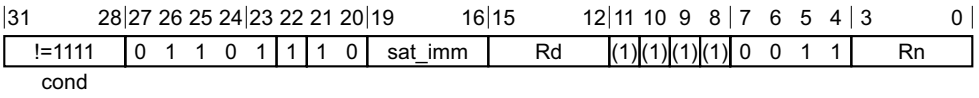

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.281 USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

### A1



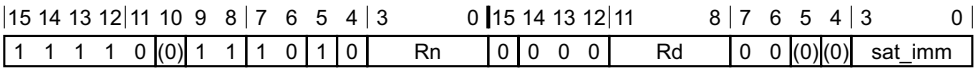

#### *A1 variant*

```
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <imm> | Is the bit position for saturation, in the range 0 to 15, encoded in the "sat_imm" field. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = UnsignedSatQ(SInt(R[n]<31:16>), saturate_to);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

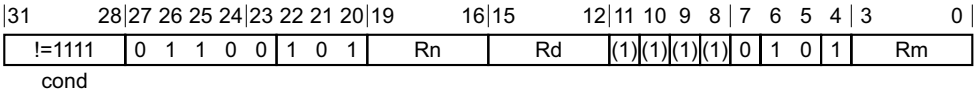
```
R[d]<15:0> = ZeroExtend(result1, 16);
R[d]<31:16> = ZeroExtend(result2, 16);
if sat1 || sat2 then
    PSTATE.Q = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.282 USAX

Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets PSTATE.GE according to the results.

### A1



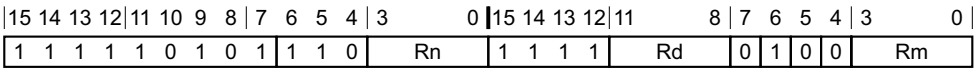

#### *A1 variant*

```
USAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
USAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

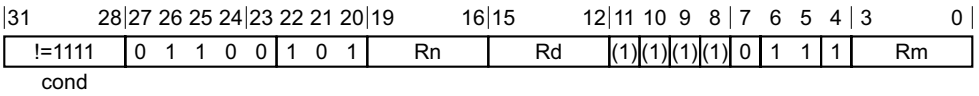
```
R[d]<15:0>  = sum<15:0>;
R[d]<31:16> = diff<15:0>;
PSTATE.GE<1:0> = if sum  >= 0x10000 then '11' else '00';
PSTATE.GE<3:2> = if diff >= 0 then '11' else '00';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.283 USUB16

Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

### A1



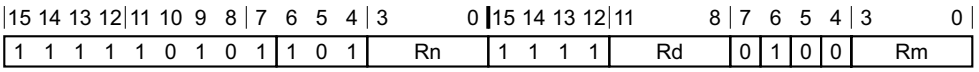

#### *A1 variant*

```
USUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
USUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<15:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

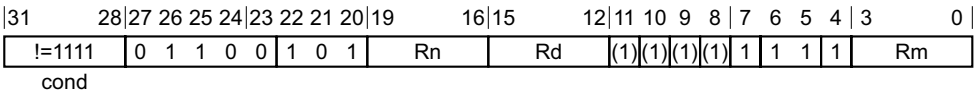
```
R[d]<31:16> = diff2<15:0>;
PSTATE.GE<1:0> = if diff1 >= 0 then '11' else '00';
PSTATE.GE<3:2> = if diff2 >= 0 then '11' else '00';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.284 USUB8

Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

### A1



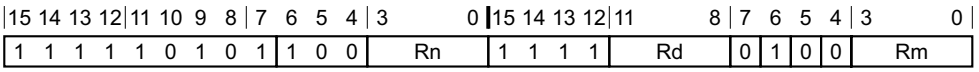

#### *A1 variant*

```
USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

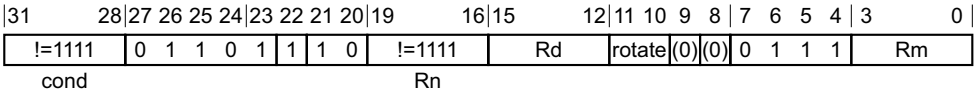
```
diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
R[d]<7:0>   = diff1<7:0>;
R[d]<15:8>  = diff2<7:0>;
R[d]<23:16> = diff3<7:0>;
R[d]<31:24> = diff4<7:0>;
PSTATE.GE<0>  = if diff1 >= 0 then '1' else '0';
PSTATE.GE<1>  = if diff2 >= 0 then '1' else '0';
PSTATE.GE<2>  = if diff3 >= 0 then '1' else '0';
PSTATE.GE<3>  = if diff4 >= 0 then '1' else '0';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.285 UXTAB

Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### A1



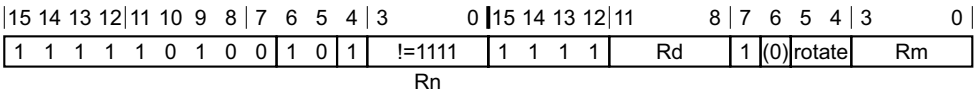

#### *A1 variant*

```
UXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE UXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE UXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>(omitted) when rotate = 00<br>8 when rotate = 01 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16 when rotate = 10

24 when rotate = 11

## Operation for all encodings

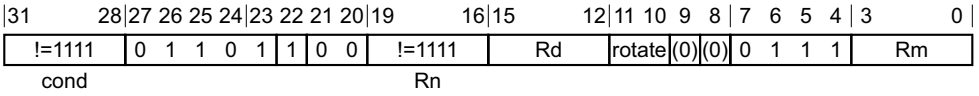
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<7:0>, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.286 UXTAB16

Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### A1



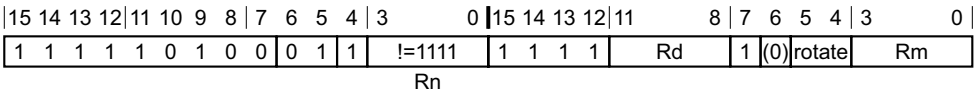

#### *A1 variant*

```
UXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE UXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE UXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>(omitted) when rotate = 00<br>8 when rotate = 01 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16 when rotate = 10

24 when rotate = 11

## Operation for all encodings

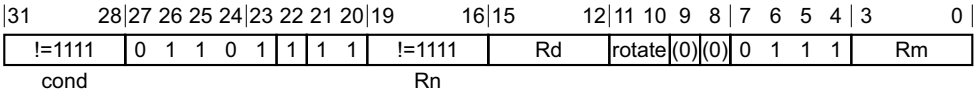
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = R[n]<15:0> + ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + ZeroExtend(rotated<23:16>, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.287 UXTAH

Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

### A1



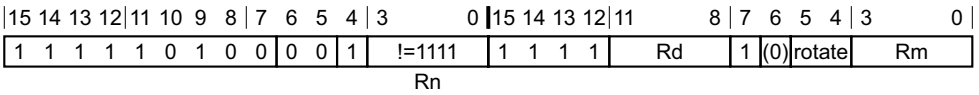

#### *A1 variant*

```
UXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE UXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
if Rn == '1111' then SEE UXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>(omitted) when rotate = 00<br>8 when rotate = 01 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

16 when rotate = 10

24 when rotate = 11

## Operation for all encodings

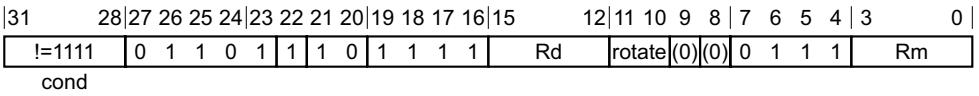
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<15:0>, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.288 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### A1



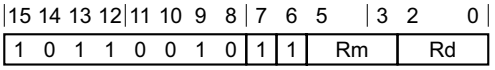

#### *A1 variant*

```
UXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1



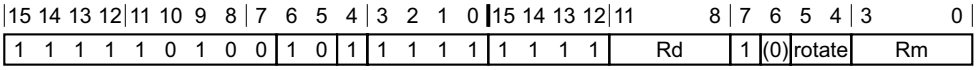

#### *T1 variant*

```
UXTB{<c>}{<q>} {<Rd>,} <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

### T2




#### *T2 variant*

```
UXTB{<c>}.W {<Rd>,} <Rm> // <Rd>, <Rm> can be represented in T1
UXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> Is the general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:

| | |
|---|---|
| (omitted) | when rotate = 00 |
| 8 | when rotate = 01 |
| 16 | when rotate = 10 |
| 24 | when rotate = 11 |

### Operation for all encodings

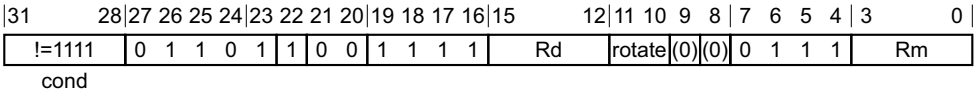
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.289 UXTB16

Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### A1



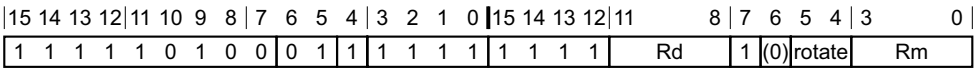

#### *A1 variant*

```
UXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

```
UXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | For encoding A1: is the general-purpose source register, encoded in the "Rm" field.<br>For encoding T1: is the second general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>(omitted) when rotate = 00<br>8 when rotate = 01<br>16 when rotate = 10<br>24 when rotate = 11 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

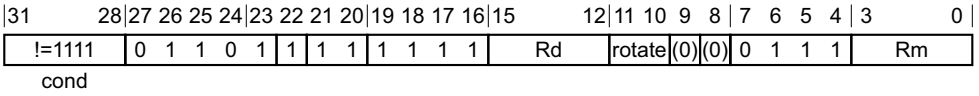
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = ZeroExtend(rotated<23:16>, 16);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.290 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

### A1



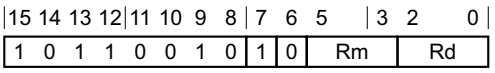

#### *A1 variant*

```
UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1



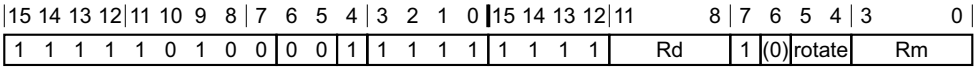

#### *T1 variant*

```
UXTH{<c>}{<q>} {<Rd>,} <Rm>
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

### T2




#### *T2 variant*

```
UXTH{<c>}.W {<Rd>,} <Rm> // <Rd>, <Rm> can be represented in T1
UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

#### *Decode for this encoding*

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>` Is the general-purpose source register, encoded in the "Rm" field.

`<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

| | |
|---|---|
| `(omitted)` | when `rotate = 00` |
| `8` | when `rotate = 01` |
| `16` | when `rotate = 10` |
| `24` | when `rotate = 11` |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

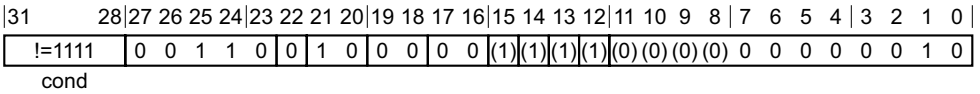
### F6.1.291 WFE

Wait For Event is a hint instruction that permits the PE to enter a low-power state until one of a number of events occurs, including events signaled by executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event and Send Event* on page G1-3834.

As described in *Wait For Event and Send Event* on page G1-3834, the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see:

- *Traps to Undefined mode of EL0 execution of WFE and WFI instructions* on page G1-3849.
- *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-3865.
- *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode* on page G1-3875.

#### A1



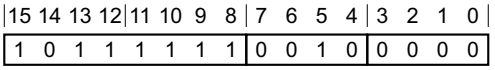

##### *A1 variant*

```
WFE{<c>}{<q>}
```

##### *Decode for this encoding*

```
// No additional decoding required
```

#### T1



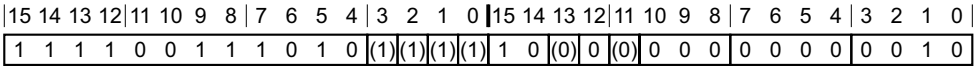

##### *T1 variant*

```
WFE{<c>}{<q>}
```

##### *Decode for this encoding*

```
// No additional decoding required
```

#### T2




##### *T2 variant*

```
WFE{<c>}.W
```

##### *Decode for this encoding*

```
// No additional decoding required
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if EventRegistered() then
        ClearEventRegister();
    else
        if PSTATE.EL == EL0 then
            AArch32.CheckForWFxTrap(EL1, TRUE);
        if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
            AArch32.CheckForWFxTrap(EL2, TRUE);
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            AArch32.CheckForWFxTrap(EL3, TRUE);
        WaitForEvent();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

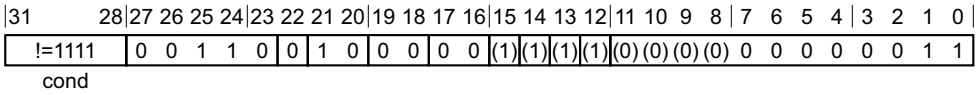
## F6.1.292 WFI

Wait For Interrupt is a hint instruction that permits the PE to enter a low-power state until one of a number of asynchronous events occurs. For more information, see *Wait For Interrupt* on page G1-3837.

As described in *Wait For Interrupt* on page G1-3837, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see:

- *Traps to Undefined mode of EL0 execution of WFE and WFI instructions* on page G1-3849.
- *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-3865.
- *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode* on page G1-3875.

### A1



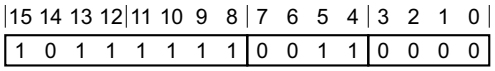

#### *A1 variant*

```
WFI{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T1



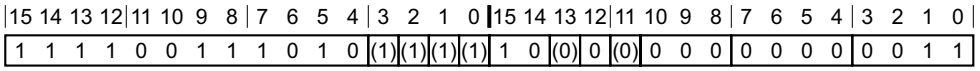

#### *T1 variant*

```
WFI{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T2




#### *T2 variant*

```
WFI{<c>}.W
```

#### *Decode for this encoding*

```
// No additional decoding required
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !InterruptPending() then
        if PSTATE.EL == EL0 then
            AArch32.CheckForWFxTrap(EL1, FALSE);
        if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
            AArch32.CheckForWFxTrap(EL2, FALSE);
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            AArch32.CheckForWFxTrap(EL3, FALSE);
        WaitForInterrupt();
```

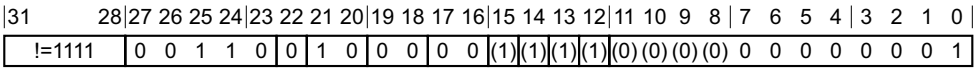

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.1.293 YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction see *The Yield instruction* on page F1-2425.

### A1



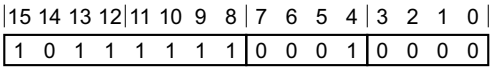

#### *A1 variant*

```
YIELD{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T1



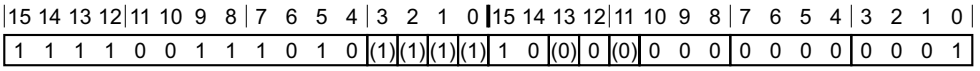

#### *T1 variant*

```
YIELD{<c>}{<q>}
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### T2




#### *T2 variant*

```
YIELD{<c>}.W
```

#### *Decode for this encoding*

```
// No additional decoding required
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F6.2 Encoding and use of Banked register transfer instructions

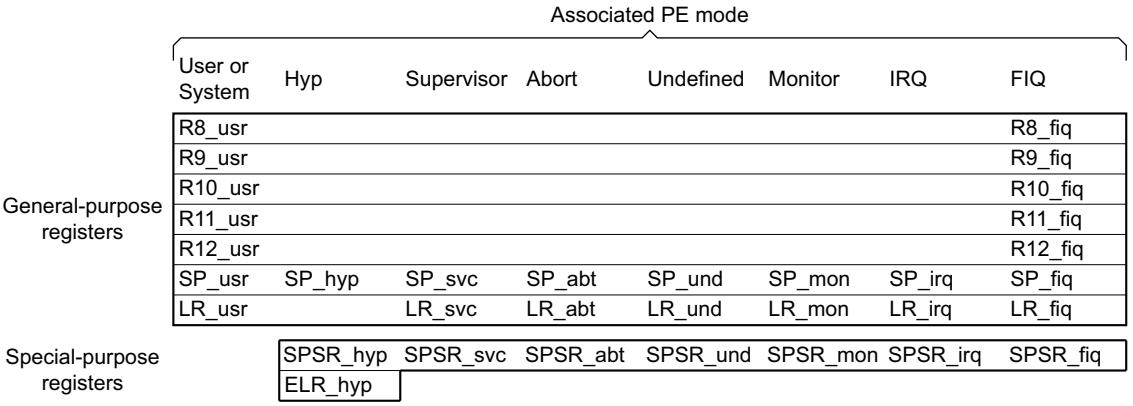
Software executing at EL1 or higher can use the MRS (Banked register) and MSR (Banked register) instructions to transfer values between the general-purpose registers and Special-purpose registers. One particular use of these instructions is for a hypervisor to save or restore the register values of a Guest OS. The following sections give more information about these instructions:

- *Register arguments in the Banked register transfer instructions*.
- *Usage restrictions on the Banked register transfer instructions* on page F6-3203.
- *Encoding the register argument in the Banked register transfer instructions* on page F6-3204.
- *Pseudocode support for the Banked register transfer instructions* on page F6-3205.

For descriptions of the instructions see *MRS (Banked register)* on page F6-2832 and *MSR (Banked register)* on page F6-2835.

### F6.2.1 Register arguments in the Banked register transfer instructions

Figure F6-1 shows the Banked general-purpose registers and Special-purpose registers:




For the general-purpose registers, if no other register is shown, the *current mode register* is the _usr register. So, for example, the full set of current mode registers, including the registers that are not banked:

- For Hyp mode, is {R0_usr - R12_usr, SP_hyp, LR_usr, SPSR_hyp, ELR_hyp}.
- For Abort mode, is {R0_usr - R12_usr, SP_abt, LR_abt, SPSR_abt}.

**Figure F6-1 Banking of general-purpose and Special-purpose registers**

Figure F6-1 is based on Figure G1-2 on page G1-3761, that shows the complete set of general-purpose registers and Special-purpose registers accessible in each mode.

**Note**

- System mode uses the same set of registers as User mode. Neither of these modes can access an SPSR, except that System mode can use the MRS (Banked register) and MSR (Banked register) instructions to access some SPSRs, as described in *Usage restrictions on the Banked register transfer instructions* on page F6-3203.
- General-purpose registers R0-R7, that are not Banked, cannot be accessed using the MRS (Banked register) and MSR (Banked register) instructions.
- In addition to the registers shown in Figure F6-1, the DLR and DSPSR are AArch32 System registers that map onto the AArch64 Special-purpose registers DLR_EL0 and DSPSR_EL0. However, DLR and DSPSR are not accessible using the MRS (Banked register) and MSR (Banked register) instructions.

Software using an MRS (Banked register) or MSR (Banked register) instruction specifies one of these registers using a name shown in Figure F6-1, or an alternative name for SP or LR. These registers can be grouped as follows:

**R8-R12** Each of these registers has two Banked copies, _usr and _fiq, for example R8_usr and R8_fiq.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SP** There is a Banked copy of SP for every mode except System mode. For example, SP_svc is the SP for Supervisor mode.

**LR** There is a Banked copy of LR for every mode except System mode and Hyp mode. For example, LR_svc is the LR for Supervisor mode.

**SPSR** There is a Banked copy of SPSR for every mode except System mode and User mode.

**ELR_hyp** Except for the operations provided by MRS (Banked register) and MSR (Banked register), ELR_hyp is accessible only from Hyp mode. It is not Banked.

### F6.2.2 Usage restrictions on the Banked register transfer instructions

MRS (Banked register) and MSR (Banked register) instructions are UNPREDICTABLE if any of the following applies:

- The instruction is executed in User mode.
- The instruction accesses a Banked register that is not implemented, or that either:
  - Is not accessible from the current Privilege level and Security state.
  - Can be accessed from the current mode by using a different instruction.

An MRS (Banked register) or an MSR (Banked register) executed:

- At Non-secure EL1 cannot access any Hyp mode Banked registers.
- At Non-secure EL1 or EL2 cannot access any Monitor mode Banked registers.
- In a Secure mode other than Monitor mode cannot access any Hyp Banked registers.

This means that the Banked registers that MRS (Banked register) and MSR (Banked register) instructions cannot access are:

**From Monitor mode**

- The current mode registers R8_usr-R12_usr, SP_mon, LR_mon, and SPSR_mon.

**From Hyp mode**

- The Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The current mode registers R8_usr-R12_usr, SP_hyp, LR_usr, and SPSR_hyp.

**Note**

MRS (Banked register) and MSR (Banked register) instructions can access the current mode register ELR_hyp.

**From FIQ mode**

- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_fiq-R12_fiq, SP_fiq, LR_fiq, and SPSR_fiq.

**From System mode**

- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_usr-R12_usr, SP_usr, and LR_usr.

**From Supervisor mode, Abort mode, Undefined mode, and IRQ mode**

- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_usr-R12_usr, SP_<current_mode>, LR_<current_mode>, and SPSR_<current_mode>.

If EL3 is using AArch64, all MRS (Banked register) and MSR (Banked register) accesses to the Monitor mode registers from Secure EL1 modes are trapped to EL3. See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-1602.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

For more information, see:

- *Encoding the register argument in the Banked register transfer instructions*.
- *Pseudocode support for the Banked register transfer instructions* on page F6-3205.
- *MRS (Banked register)* on page F6-2832.
- *MSR (Banked register)* on page F6-2835.

**Note**

UNPREDICTABLE behavior must not give access to registers that are not accessible from the current Privilege level and Security state.

### F6.2.3 Encoding the register argument in the Banked register transfer instructions

The MRS (Banked register) and MSR (Banked register) instructions include a 5-bit field, SYSm, and an R bit, that together encode the register argument for the instruction.

When the R bit is set to 0, the argument is a register other than a Banked copy of the SPSR, and Table F6-1 shows how the SYSm field defines the required register argument.

**Table F6-1 Banked register encodings when R==0**

| | SYSm<4:3> | | | |
|---|---|---|---|---|
| SYSm<2:0> | 0b00 | 0b01 | 0b10 | 0b11 |
| 0b000 | R8_usr | R8_fiq | LR_irq | UNPREDICTABLE |
| 0b001 | R9_usr | R9_fiq | SP_irq | UNPREDICTABLE |
| 0b010 | R10_usr | R10_fiq | LR_svc | UNPREDICTABLE |
| 0b011 | R11_usr | R11_fiq | SP_svc | UNPREDICTABLE |
| 0b100 | R12_usr | R12_fiq | LR_abt | LR_mon |
| 0b101 | SP_usr | SP_fiq | SP_abt | SP_mon |
| 0b110 | LR_usr | LR_fiq | LR_und | ELR_hyp |
| 0b111 | UNPREDICTABLE | UNPREDICTABLE | SP_und | SP_hyp |

When the R bit is set to 1, the argument is a Banked copy of the SPSR, and Table F6-2 shows how the SYSm field defines the required register argument.

**Table F6-2 Banked register encodings when R==1**

| | SYSm<4:3> | | | |
|---|---|---|---|---|
| SYSm<2:0> | 0b00 | 0b01 | 0b10 | 0b11 |
| 0b000 | UNPREDICTABLE | UNPREDICTABLE | SPSR_irq | UNPREDICTABLE |
| 0b001 | UNPREDICTABLE | UNPREDICTABLE | UNPREDICTABLE | UNPREDICTABLE |
| 0b010 | UNPREDICTABLE | UNPREDICTABLE | SPSR_svc | UNPREDICTABLE |
| 0b011 | UNPREDICTABLE | UNPREDICTABLE | UNPREDICTABLE | UNPREDICTABLE |
| 0b100 | UNPREDICTABLE | UNPREDICTABLE | SPSR_abt | SPSR_mon |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table F6-2 Banked register encodings when R==1 (continued)**

| | SYSm<4:3> | | | |
|---|---|---|---|---|
| **SYSm<2:0>** | 0b00 | 0b01 | 0b10 | 0b11 |
| 0b101 | UNPREDICTABLE | UNPREDICTABLE | UNPREDICTABLE | UNPREDICTABLE |
| 0b110 | UNPREDICTABLE | SPSR_fiq | SPSR_und | SPSR_hyp |
| 0b111 | UNPREDICTABLE | UNPREDICTABLE | UNPREDICTABLE | UNPREDICTABLE |

### F6.2.4 Pseudocode support for the Banked register transfer instructions

The pseudocode functions `BankedRegisterAccessValid()` and `SPSRaccessValid()` check the validity of `MRS` (Banked register) and `MSR` (Banked register) accesses. That is, they filter the accesses that are UNPREDICTABLE either because:

- They attempt to access a register that *Usage restrictions on the Banked register transfer instructions* on page F6-3203 shows is not accessible.
- They use an `SYSm<4:0>` encoding that *Encoding the register argument in the Banked register transfer instructions* on page F6-3204 shows as UNPREDICTABLE.

`BankedRegisterAccessValid()` applies to accesses to the banked general-purpose registers, or to ELR_hyp, and `SPSRaccessValid()` applies to accesses to the SPSRs.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter F7
# T32 and A32 Advanced SIMD and floating-point Instruction Descriptions

This chapter describes each instruction. It contains the following sections:


- *Alphabetical list of floating-point and Advanced SIMD instructions* on page F7-3208.


**Note**

Some headings in this chapter use the term *floating-point register*. This is an abbreviated description, and means a register in the Advanced SIMD and floating-point register file.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F7.1 Alphabetical list of floating-point and Advanced SIMD instructions

This section lists every floating-point and Advanced SIMD instruction in the T32 and A32 instruction sets. For details of the format used see *Format of instruction descriptions* on page F2-2442.

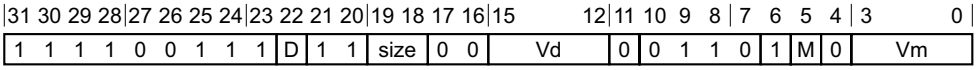
This section is formatted so that each instruction description starts on a new page.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F7.1.1 AESD

AES single round decryption.

### A1



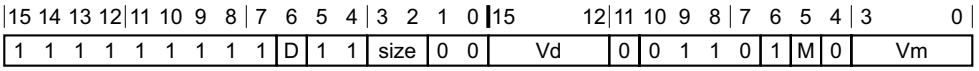

#### *A1 variant*

```
AESD.<dt> <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
AESD.<dt> <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<dt> Is the data type, encoded in the "size" field. It can have the following values:

8 when `size = 00`

The following encodings are reserved:

- `size = 01`.
- `size = 1x`.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

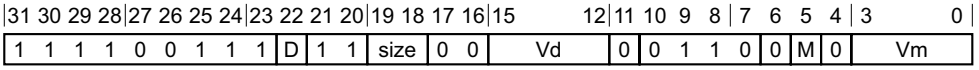
```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[m>>1];
    Q[d>>1] = AESInvSubBytes(AESInvShiftRows(op1 EOR op2));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.2 AESE

AES single round encryption.

### A1



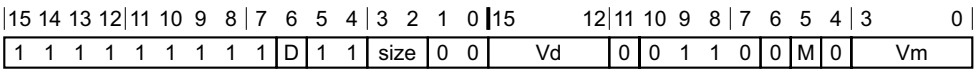

#### *A1 variant*

```
AESE.<dt> <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
AESE.<dt> <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<dt> Is the data type, encoded in the "size" field. It can have the following values:

8 when `size = 00`

The following encodings are reserved:

- `size = 01`.
- `size = 1x`.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

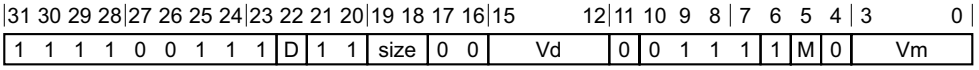
```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[m>>1];
    Q[d>>1] = AESSubBytes(AESShiftRows(op1 EOR op2));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.3 AESIMC

AES inverse mix columns.

### A1



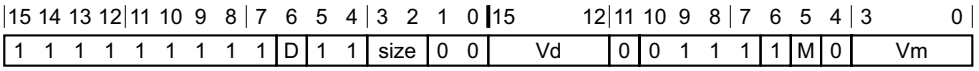

#### *A1 variant*

```
AESIMC.<dt> <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
AESIMC.<dt> <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Assembler symbols

<dt> Is the data type, encoded in the "size" field. It can have the following values:

8 when `size = 00`

The following encodings are reserved:

- `size = 01`.
- `size = 1x`.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

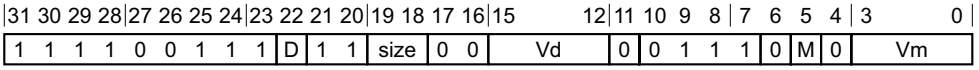
```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = AESInvMixColumns(Q[m>>1]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F7.1.4 AESMC

AES mix columns.

#### A1



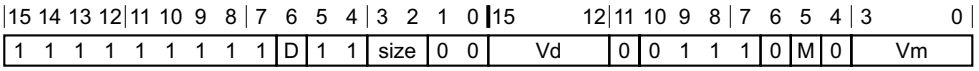

##### *A1 variant*

```
AESMC.<dt> <Qd>, <Qm>
```

##### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

#### T1




##### *T1 variant*

```
AESMC.<dt> <Qd>, <Qm>
```

##### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<dt> Is the data type, encoded in the "size" field. It can have the following values:

8 when size = 00

The following encodings are reserved:

- size = 01.
- size = 1x.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = AESMixColumns(Q[m>>1]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

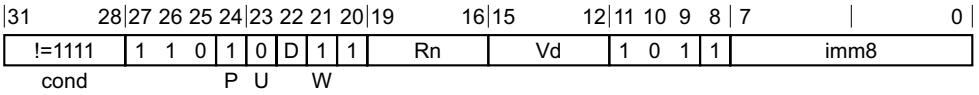
### F7.1.5 FLDMDBX

FLDMDBX loads multiple registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register. ARM deprecates use of FLDMDBX, except for disassembly purposes, and reassembly of disassembled code.

This instruction is an alias of the VLDM, VLDMDB, VLDMIA instruction. This means that:

- The encodings in this description are named to match the encodings of VLDM, VLDMDB, VLDMIA.
- The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.

#### A1




##### *Decrement Before variant*

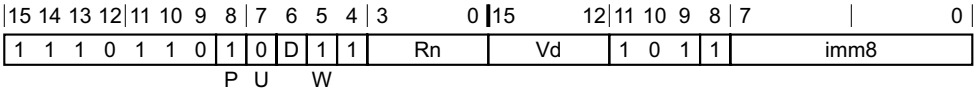
```
FLDMDBX{<c>}{<q>} <Rn>{!}, <dreglist>
```

is equivalent to

```
VLDMDB{<c>}{<q>} <Rn>{!}, <dreglist>
```

and is the preferred disassembly when `imm8<0> == '1'`.

#### T1




##### *Decrement Before variant*

```
FLDMDBX{<c>}{<q>} <Rn>{!}, <dreglist>
```

is equivalent to

```
VLDMDB{<c>}{<q>} <Rn>{!}, <dreglist>
```

and is the preferred disassembly when `imm8<0> == '1'`.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Rn>` | For encoding A1: is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.<br>For encoding T1: is the general-purpose base register, encoded in the "Rn" field. |
| `!` | Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0. |
| `<dreglist>` | Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

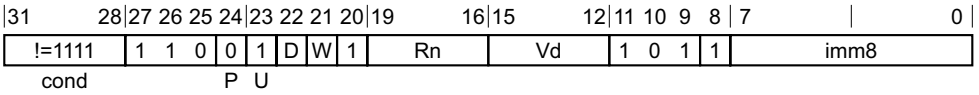
### F7.1.6 FLDMIAX

FLDMIAX loads multiple registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register. ARM deprecates use of FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

This instruction is an alias of the VLDM, VLDMDB, VLDMIA instruction. This means that:

- The encodings in this description are named to match the encodings of VLDM, VLDMDB, VLDMIA.
- The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.

#### A1




##### *Increment After variant*

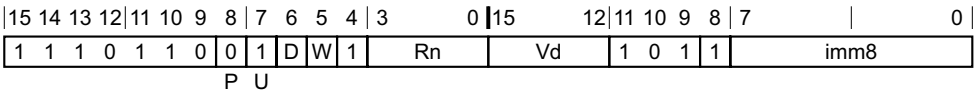
`FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>`

is equivalent to

`VLDM{<c>}{<q>} <Rn>{!}, <dreglist>`

and is the preferred disassembly when `imm8<0> == '1'`.

#### T1




##### *Increment After variant*

`FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>`

is equivalent to

`VLDM{<c>}{<q>} <Rn>{!}, <dreglist>`

and is the preferred disassembly when `imm8<0> == '1'`.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

`!` Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.

`<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

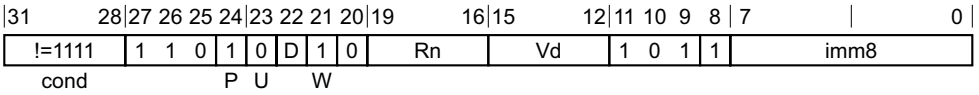
## F7.1.7 FSTMDBX

FSTMDBX stores multiple registers from the Advanced SIMD and floating-point register file to consecutive memory locations using an address from a general-purpose register. ARM deprecates use of FSTMDBX, except for disassembly purposes, and reassembly of disassembled code.

This instruction is an alias of the VSTM, VSTMDB, VSTMIA instruction. This means that:

- The encodings in this description are named to match the encodings of VSTM, VSTMDB, VSTMIA.
- The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

### A1




#### *Decrement Before variant*

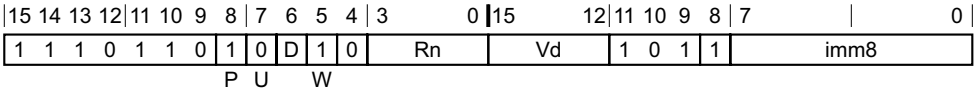
`FSTMDBX{<c>}{<q>} <Rn>{!}, <dreglist>`

is equivalent to

`VSTMDB{<c>}{<q>} <Rn>{!}, <dreglist>`

and is the preferred disassembly when `imm8<0> == '1'`.

### T1




#### *Decrement Before variant*

`FSTMDBX{<c>}{<q>} <Rn>{!}, <dreglist>`

is equivalent to

`VSTMDB{<c>}{<q>} <Rn>{!}, <dreglist>`

and is the preferred disassembly when `imm8<0> == '1'`.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

`!` Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.

`<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

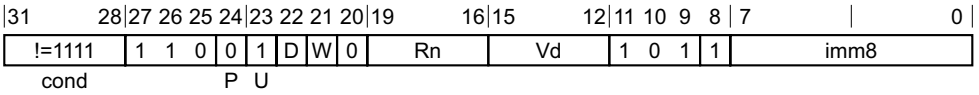
### F7.1.8 FSTMIAX

FSTMIAX stores multiple registers from the Advanced SIMD and floating-point register file to consecutive memory locations using an address from a general-purpose register. ARM deprecates use of FSTMIAX, except for disassembly purposes, and reassembly of disassembled code.

This instruction is an alias of the VSTM, VSTMDB, VSTMIA instruction. This means that:

- The encodings in this description are named to match the encodings of VSTM, VSTMDB, VSTMIA.
- The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

#### A1




##### *Increment After variant*

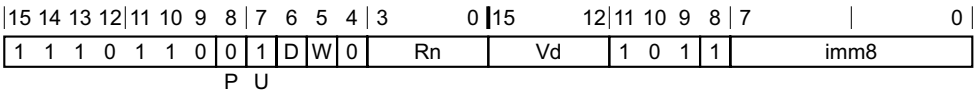
`FSTMIAX{<c>}{<q>} <Rn>{!}, <dreglist>`

is equivalent to

`VSTM{<c>}{<q>} <Rn>{!}, <dreglist>`

and is the preferred disassembly when `imm8<0> == '1'`.

#### T1




##### *Increment After variant*

`FSTMIAX{<c>}{<q>} <Rn>{!}, <dreglist>`

is equivalent to

`VSTM{<c>}{<q>} <Rn>{!}, <dreglist>`

and is the preferred disassembly when `imm8<0> == '1'`.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

`!` Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.

`<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

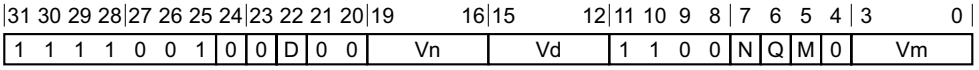
The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.9 SHA1C

SHA1 hash update (choose).

### A1



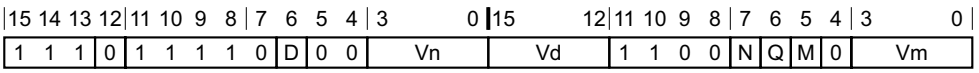

#### *A1 variant*

```
SHA1C.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
SHA1C.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1];
    Y = Q[n>>1]<31:0>; // Note: 32 bits wide
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

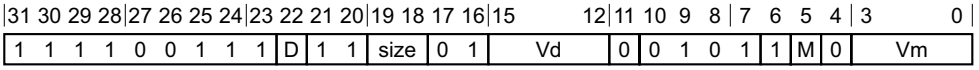
```
W = Q[m>>1];
for e = 0 to 3
    t = SHAchoose(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
Q[d>>1] = X;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.10 SHA1H

SHA1 fixed rotate.

### A1



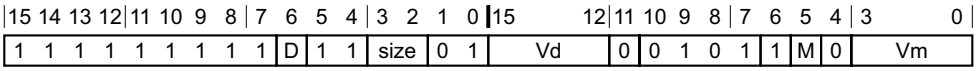

#### *A1 variant*

```
SHA1H.32 <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
SHA1H.32 <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

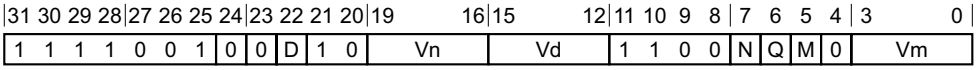
```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = ZeroExtend(ROL(Q[m>>1]<31:0>, 30), 128);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.11 SHA1M

SHA1 hash update (majority).

### A1



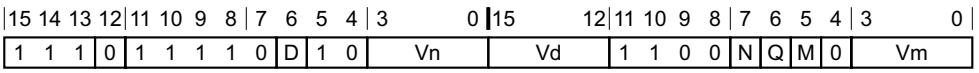

#### *A1 variant*

```
SHA1M.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
SHA1M.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1];
    Y = Q[n>>1]<31:0>; // Note: 32 bits wide
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

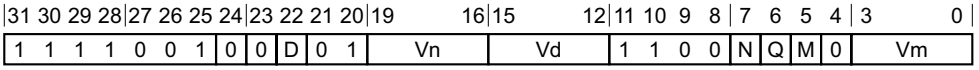
```
W = Q[m>>1];
for e = 0 to 3
    t = SHAmajority(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
Q[d>>1] = X;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.12 SHA1P

SHA1 hash update (parity).

### A1



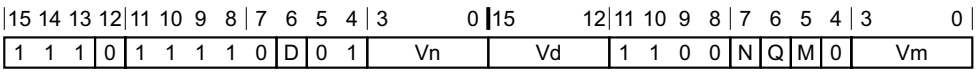

#### *A1 variant*

```
SHA1P.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
SHA1P.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1];
    Y = Q[n>>1]<31:0>; // Note: 32 bits wide
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

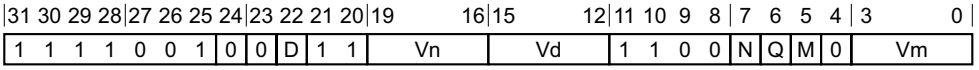
```
W = Q[m>>1];
for e = 0 to 3
    t = SHAparity(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
Q[d>>1] = X;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.13 SHA1SU0

SHA1 schedule update 0.

### A1



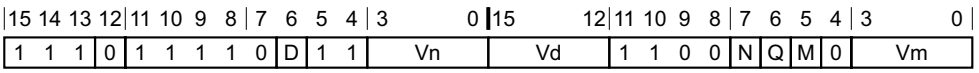

#### *A1 variant*

```
SHA1SU0.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
SHA1SU0.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

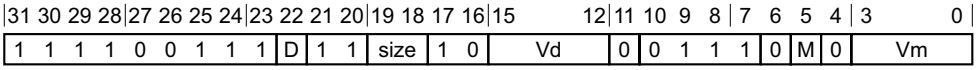
```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[n>>1]; op3 = Q[m>>1];
    op2 = op2<63:0> : op1<127:64>;
    Q[d>>1] = op1 EOR op2 EOR op3;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.14 SHA1SU1

SHA1 schedule update 1.

### A1



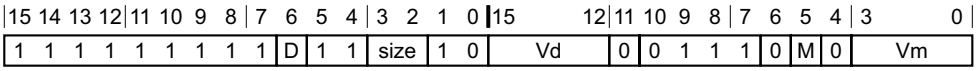

#### *A1 variant*

```
SHA1SU1.32 <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
SHA1SU1.32 <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[m>>1];
    T = X EOR LSR(Y, 32);
    W0 = ROL(T<31:0>, 1);
    W1 = ROL(T<63:32>, 1);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

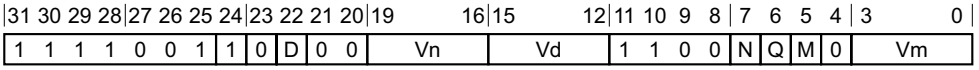
```
W2 = ROL(T<95:64>, 1);
W3 = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
Q[d>>1] = W3:W2:W1:W0;
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## F7.1.15 SHA256H

SHA256 hash update part 1.

### A1



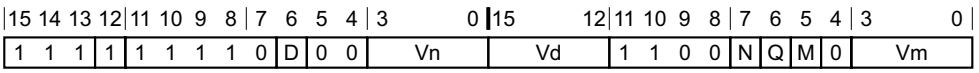

#### *A1 variant*

```
SHA256H.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
SHA256H.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |

### Operation for all encodings

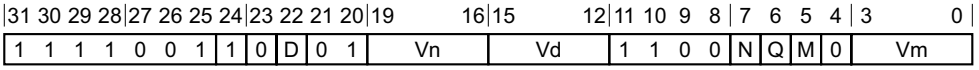
```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[n>>1]; W = Q[m>>1]; part1 = TRUE;
    Q[d>>1] = SHA256hash(X, Y, W, part1);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.16 SHA256H2

SHA256 hash update part 2.

### A1



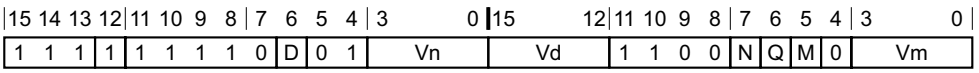

#### *A1 variant*

```
SHA256H2.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
SHA256H2.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |

### Operation for all encodings

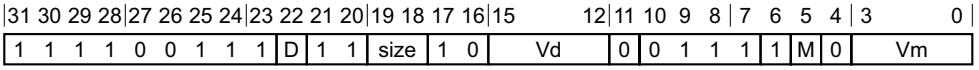
```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[n>>1]; Y = Q[d>>1]; W = Q[m>>1]; part1 = FALSE;
    Q[d>>1] = SHA256hash(X, Y, W, part1);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.17 SHA256SU0

SHA256 schedule update 0.

### A1



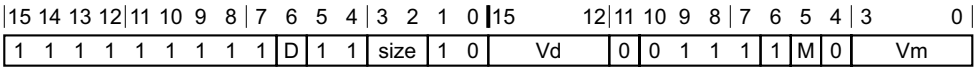

#### *A1 variant*

```
SHA256SU0.32 <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
SHA256SU0.32 <Qd>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

```
if ConditionPassed() then
    bits(128) result;
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[m>>1];
    T = Y<31:0> : X<127:32>;
    for e = 0 to 3
        elt = Elem[T, e, 32];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

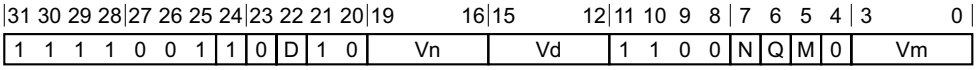
```
        elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
        Elem[result, e, 32] = elt + Elem[X, e, 32];
    Q[d>>1] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.18 SHA256SU1

SHA256 schedule update 1.

### A1



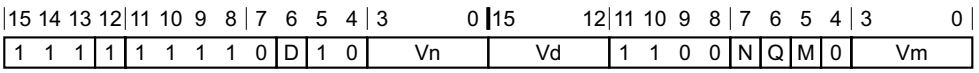

#### *A1 variant*

```
SHA256SU1.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
SHA256SU1.32 <Qd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if ! HaveCryptoExt() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

```
if ConditionPassed() then
    bits(128) result;
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[n>>1]; Z = Q[m>>1];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
T0 = Z<31:0> : Y<127:32>;

T1 = Z<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[X, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e - 2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[X, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

Q[d>>1] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.19 VABA

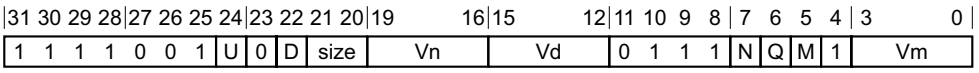
Vector Absolute Difference and Accumulate subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction that is not also available as a floating-point instruction, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

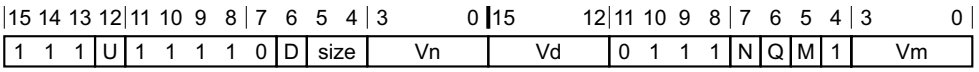
Applies when `Q = 1`.

```
VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VABA or VABAL instruction must be unconditional. ARM strongly recommends that a T32 VABA or VABAL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8 when U = 0, size = 00

S16 when U = 0, size = 01

S32 when U = 0, size = 10

U8 when U = 1, size = 00

U16 when U = 1, size = 01

U32 when U = 1, size = 10

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + absdiff;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.20 VABAL

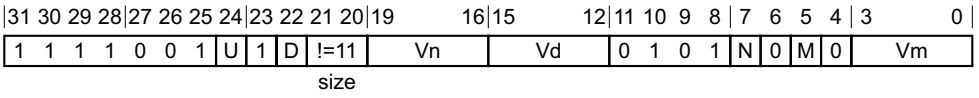
Vector Absolute Difference and Accumulate Long subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction that is not also available as a floating-point instruction, see *Conditional execution* on page F2-2447.

### A1



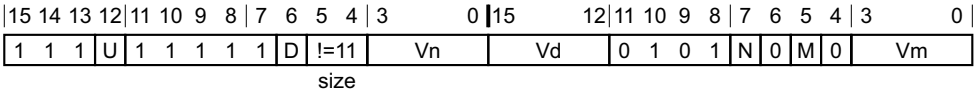

#### *A1 variant*

```
VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

### T1




#### *T1 variant*

```
VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VABA or VABAL instruction must be unconditional. ARM strongly recommends that a T32 VABA or VABAL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + absdiff;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.21 VABD (floating-point)

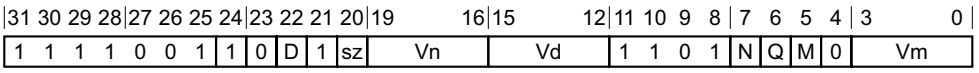
Vector Absolute Difference (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are all single-precision floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VABD{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

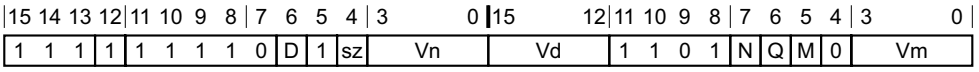
Applies when `Q = 1`.

```
VABD{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VABD{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VABD{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VABD instruction must be unconditional. ARM strongly recommends that a T32 VABD instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            Elem[D[d+r],e,esize] = FPAbs(FPSub(op1,op2,StandardFPSCRValue()));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.22 VABD (integer)

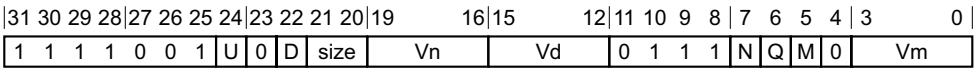
Vector Absolute Difference (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction that is not also available as a floating-point instruction, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

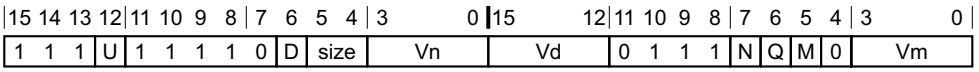
Applies when `Q = 1`.

```
VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VABD or VABDL instruction must be unconditional. ARM strongly recommends that a T32 VABD or VABDL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8 when U = 0, size = 00

S16 when U = 0, size = 01

S32 when U = 0, size = 10

U8 when U = 1, size = 00

U16 when U = 1, size = 01

U32 when U = 1, size = 10

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.23 VABDL (integer)

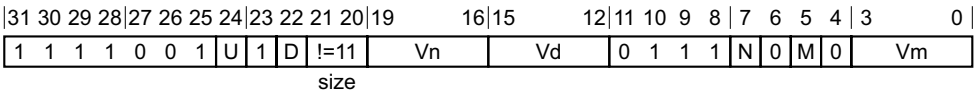
Vector Absolute Difference Long (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction that is not also available as a floating-point instruction, see *Conditional execution* on page F2-2447.

### A1



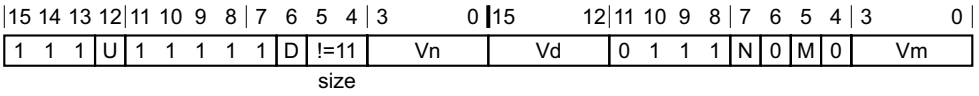

#### *A1 variant*

```
VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

### T1




#### *T1 variant*

```
VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VABD or VABDL instruction must be unconditional. ARM strongly recommends that a T32 VABD or VABDL instruction is unconditional, see *Conditional execution* on page F2-2447.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

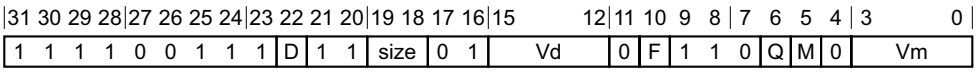
## F7.1.24 VABS

Vector Absolute takes the absolute value of each element in a vector, and places the results in a second vector. The floating-point version only clears the sign bit.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

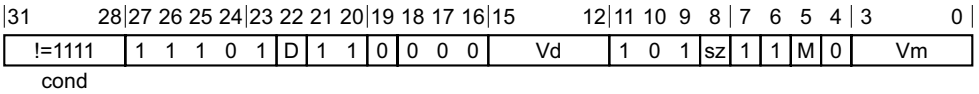
Applies when `Q = 1`.

```
VABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### A2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VABS{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

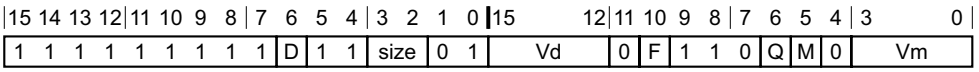
```
VABS{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

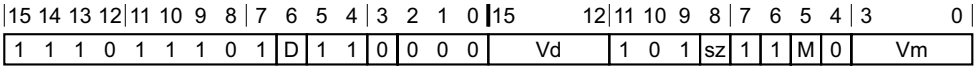
Applies when `Q = 1`.

```
VABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VABS{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VABS{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VABS` instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VABS` instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<dt> Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:

S8 when F = 0, size = 00

S16 when F = 0, size = 01

S32 when F = 0, size = 10

F32 when F = 1, size = 10

The following encodings are reserved:

- F = 0, size = 11.
- F = 1, size = 0x.
- F = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPAbs(Elem[D[m+r],e,esize]);
                else
                    result = Abs(SInt(Elem[D[m+r],e,esize]));
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else             // VFP instruction
        if dp_operation then
            D[d] = FPAbs(D[m]);
        else
            S[d] = FPAbs(S[m]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.25 VACGE

Vector Absolute Compare Greater Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements must be 32-bit floating-point numbers.

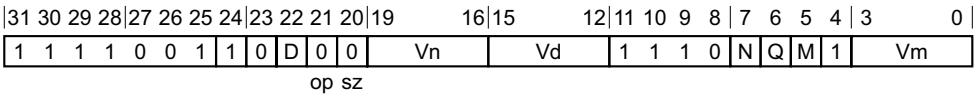
The result vector elements are 32-bit fields.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction that is not also available as a floating-point instruction, see *Conditional execution* on page F2-2447.

This instruction is used by the alias VACLE. See the *Alias conditions* on page F7-3255 table for details of when each alias is preferred.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VACGE{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

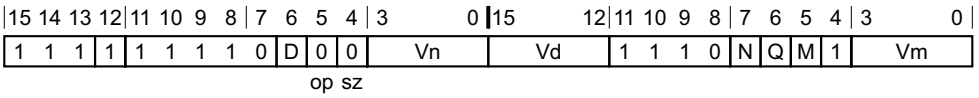
Applies when `Q = 1`.

```
VACGE{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
or_equal = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VACGE{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VACGE{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
or_equal = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| VACLE | Never |

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VACGE, VACGT, VACLE, or VACLT instruction must be unconditional. ARM strongly recommends that a T32 VACGE, VACGT, VACLE, or VACLT instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = FPAbs(Elem[D[n+r],e,esize]);  op2 = FPAbs(Elem[D[m+r],e,esize]);
            if or_equal then
                test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
            else
                test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

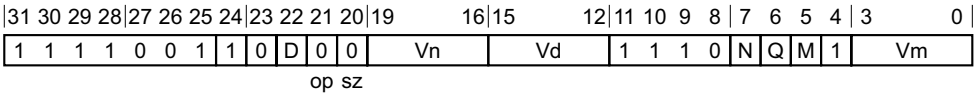
## F7.1.26 VACLE

Vector Absolute Compare Less Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros

This instruction is an alias of the VACGE instruction. This means that:

- The encodings in this description are named to match the encodings of VACGE.
- The description of VACGE gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VACLE{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGE{<c>}{<q>}.F32 <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

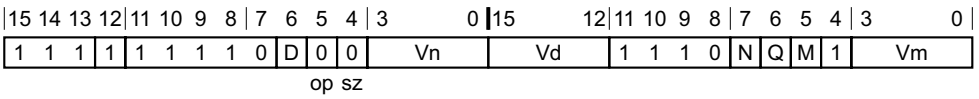
```
VACLE{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGE{<c>}{<q>}.F32 <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VACLE{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGE{<c>}{<q>}.F32 <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`VACLE{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>`

is equivalent to

`VACGE{<c>}{<q>}.F32 <Qd>, <Qm>, <Qn>`

and is never the preferred disassembly.

### Assembler symbols

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VACGE`, `VACGT`, `VACLE`, or `VACLT` instruction must be unconditional. ARM strongly recommends that a T32 `VACGE`, `VACGT`, `VACLE`, or `VACLT` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

### Operation for all encodings

The description of VACGE gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.27 VACGT

Vector Absolute Compare Greater Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements must be 32-bit floating-point numbers.

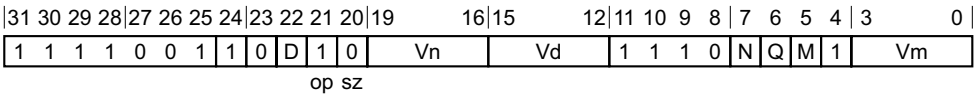
The result vector elements are 32-bit fields.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction that is not also available as a floating-point instruction, see *Conditional execution* on page F2-2447.

This instruction is used by the alias VACLT. See the *Alias conditions* on page F7-3259 table for details of when each alias is preferred.

### A1




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VACGT{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### 128-bit SIMD vector variant

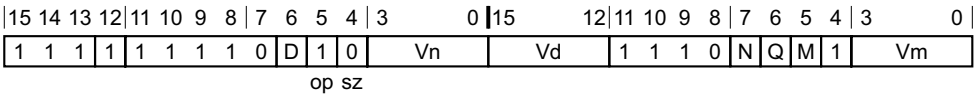
Applies when `Q = 1`.

```
VACGT{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
or_equal = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VACGT{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### 128-bit SIMD vector variant

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VACGT{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
or_equal = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| VACLT | Never |

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VACGE, VACGT, VACLE, or VACLT instruction must be unconditional. ARM strongly recommends that a T32 VACGE, VACGT, VACLE, or VACLT instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = FPAbs(Elem[D[n+r],e,esize]);  op2 = FPAbs(Elem[D[m+r],e,esize]);
            if or_equal then
                test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
            else
                test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

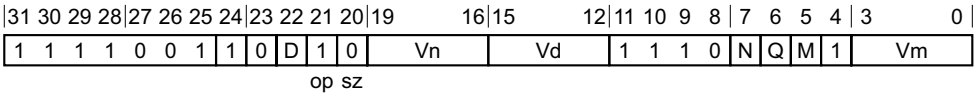
## F7.1.28 VACLT

Vector Absolute Compare Less Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros

This instruction is an alias of the VACGT instruction. This means that:

- The encodings in this description are named to match the encodings of VACGT.
- The description of VACGT gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VACLT{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGT{<c>}{<q>}.F32 <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

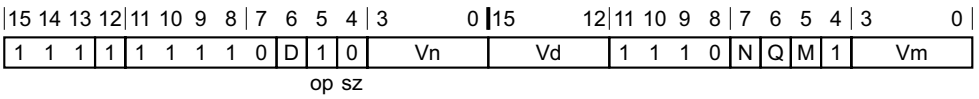
```
VACLT{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGT{<c>}{<q>}.F32 <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VACLT{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGT{<c>}{<q>}.F32 <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`VACLT{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>`

is equivalent to

`VACGT{<c>}{<q>}.F32 <Qd>, <Qm>, <Qn>`

and is never the preferred disassembly.

### Assembler symbols

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VACGE`, `VACGT`, `VACLE`, or `VACLT` instruction must be unconditional. ARM strongly recommends that a T32 `VACGE`, `VACGT`, `VACLE`, or `VACLT` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

### Operation for all encodings

The description of VACGT gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

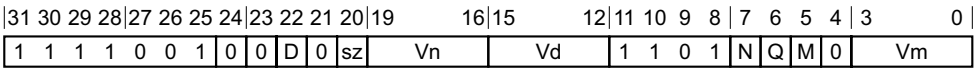
## F7.1.29 VADD (floating-point)

Vector Add (floating-point) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VADD{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

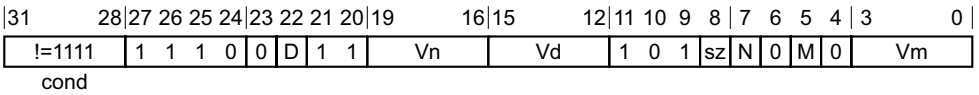
Applies when `Q = 1`.

```
VADD{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### A2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VADD{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

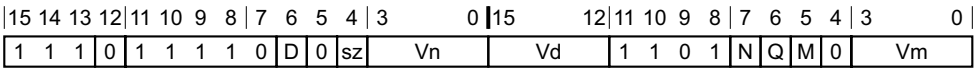
```
VADD{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VADD{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

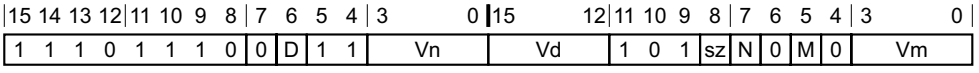
Applies when `Q = 1`.

```
VADD{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VADD{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VADD{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## Assembler symbols

<c> For encoding A1, A2 and T1: see *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VADD` instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VADD` instruction is unconditional, see *Conditional execution* on page F2-2447

For encoding T2: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize],
                                             StandardFPSCRValue());
    else             // VFP instruction
        if dp_operation then
            D[d] = FPAdd(D[n], D[m], FPSCR);
        else
            S[d] = FPAdd(S[n], S[m], FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

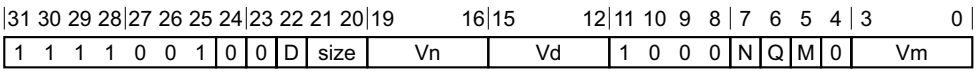
## F7.1.30 VADD (integer)

Vector Add (integer) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

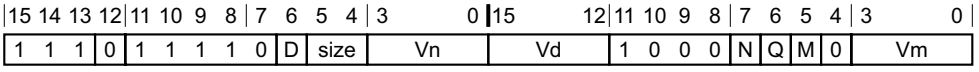
Applies when `Q = 1`.

```
VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VADD instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VADD instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| I8 | when size = 00 |
| I16 | when size = 01 |
| I32 | when size = 10 |
| I64 | when size = 11 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] + Elem[D[m+r],e,esize];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.31 VADDHN

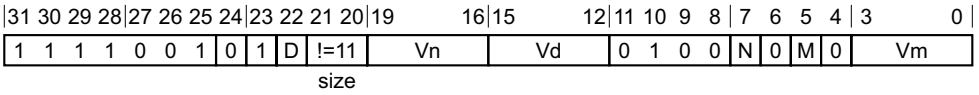
Vector Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are truncated. For rounded results, see VRADDHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



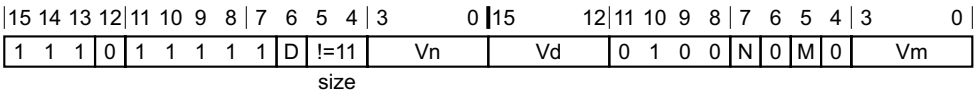

#### *A1 variant*

```
VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VADDHN instruction must be unconditional.ARM strongly recommends that a T32 VADDHN instruction is unconditional, see *Conditional execution* on page F2-2447.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

I16 when size = 00

I32 when size = 01

I64 when size = 10

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

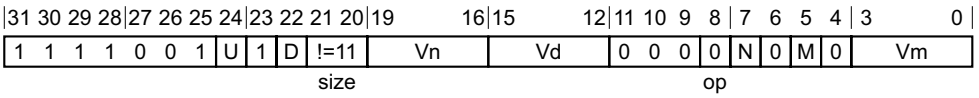
## F7.1.32 VADDL

Vector Add Long adds corresponding elements in two doubleword vectors, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



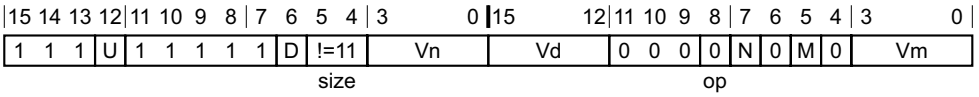

#### *A1 variant*

```
VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VADDL or VADDW instruction must be unconditional. ARM strongly recommends that a T32 VADDL or VADDW instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<dt> Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

S8 when U = 0, size = 00

S16 when U = 0, size = 01

S32 when U = 0, size = 10

U8 when U = 1, size = 00

U16 when U = 1, size = 01

U32 when U = 1, size = 10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vaddw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 + Int(Elem[Din[m],e,esize],unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

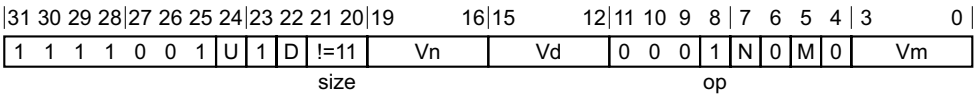
## F7.1.33 VADDW

Vector Add Wide adds corresponding elements in one quadword and one doubleword vector, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



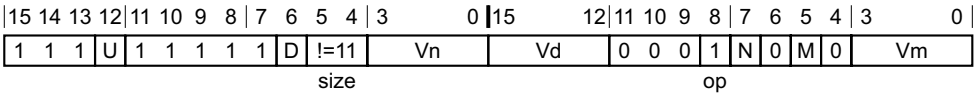

#### *A1 variant*

```
VADDW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VADDW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VADDL or VADDW instruction must be unconditional. ARM strongly recommends that a T32 VADDL or VADDW instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<dt> Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vaddw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 + Int(Elem[Din[m],e,esize],unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

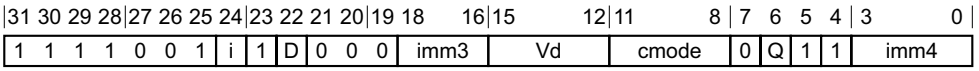
## F7.1.34 VAND (immediate)

Vector Bitwise AND (immediate) performs a bitwise AND between a register value and an immediate value, and returns the result into the destination vector

This instruction is an alias of the VBIC (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of VBIC (immediate).
- The description of VBIC (immediate) gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VAND{<c>}{<q>}.<dt> {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.<dt> <Dd>, #~<imm>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

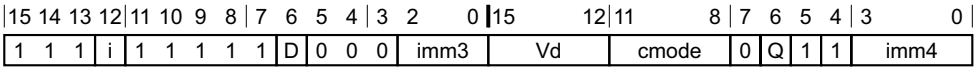
```
VAND{<c>}{<q>}.<dt> {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.<dt> <Qd>, #~<imm>
```

and is never the preferred disassembly.

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VAND{<c>}{<q>}.<dt> {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.<dt> <Dd>, #~<imm>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VAND{<c>}{<q>}.<dt> {<Qd>,} <Qd>, #<imm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
VBIC{<c>}{<q>}.<dt> <Qd>, #~<imm>
```

and is never the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. An A32 `VORR` instruction must be unconditional. ARM strongly recommends that a T32 `VORR` instruction is unconditional, see *Conditional execution* on page F2-2447. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<dt>` | The data type used for `<imm>`. It can be either `I16` or `I32`. `I8`, `I64`, and `F32` are also permitted, but the resulting syntax is a pseudo-instruction. |
| `<Qd>` | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| `<imm>` | Is a constant of the type specified by `<dt>` that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<dt>` and `<imm>`, see *One register and a modified immediate value* on page F5-2562. |

### Operation for all encodings

The description of VBIC (immediate) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

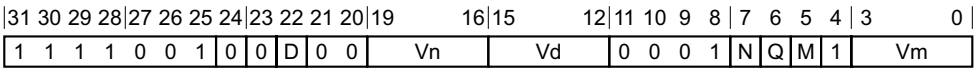
## F7.1.35 VAND (register)

Vector Bitwise AND (register) performs a bitwise AND operation between two registers, and places the result in the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VAND{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

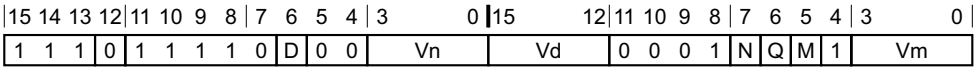
Applies when `Q = 1`.

```
VAND{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VAND{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VAND{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. An A32 VAND instruction must be unconditional. ARM strongly recommends that a T32 VAND instruction is unconditional, see *Conditional execution* on page F2-2447. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <dt> | An optional data type. It is ignored by assemblers, and does not affect the encoding. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND D[m+r];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.36 VBIC (immediate)

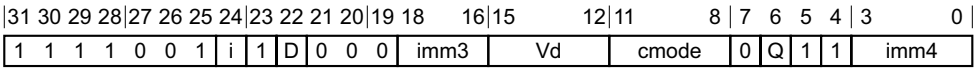
Vector Bitwise Bit Clear (immediate) performs a bitwise AND between a register value and the complement of an immediate value, and returns the result into the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

This instruction is used by the alias VAND (immediate). See the *Alias conditions* on page F7-3278 table for details of when each alias is preferred.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VBIC{<c>}{<q>}.<dt> {<Dd>,} <Dd>, #<imm>
```

#### *128-bit SIMD vector variant*

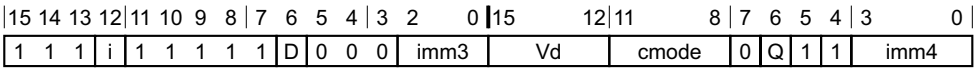
Applies when `Q = 1`.

```
VBIC{<c>}{<q>}.<dt> {<Qd>,} <Qd>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VBIC{<c>}{<q>}.<dt> {<Dd>,} <Dd>, #<imm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VBIC{<c>}{<q>}.<dt> {<Qd>,} <Qd>, #<imm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Decode for all variants of this encoding*

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

### Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562

### Alias conditions

| Alias | is preferred when |
|---|---|
| VAND (immediate) | Never |

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VBIC instruction must be unconditional. ARM strongly recommends that a T32 VBIC instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> The data type used for <imm>. It can be either I16 or I32. I8, I64, and F32 are also permitted, but the resulting syntax is a pseudo-instruction.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<imm> Is a constant of the type specified by <dt> that is replicated to fill the destination register. For details of the range of constants available and the encoding of <dt> and <imm>, see *One register and a modified immediate value* on page F5-2562.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] AND NOT(imm64);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

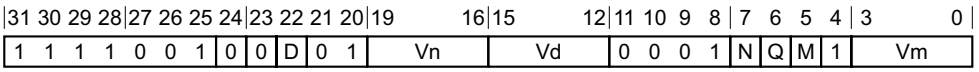
## F7.1.37 VBIC (register)

Vector Bitwise Bit Clear (register) performs a bitwise AND between a register value and the complement of a register value, and places the result in the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VBIC{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

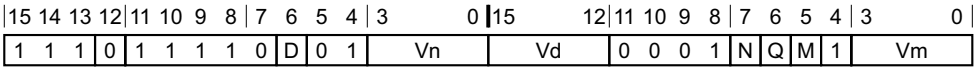
Applies when `Q = 1`.

```
VBIC{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VBIC{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VBIC{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VBIC` instruction must be unconditional. ARM strongly recommends that a T32 `VBIC` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND NOT(D[m+r]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

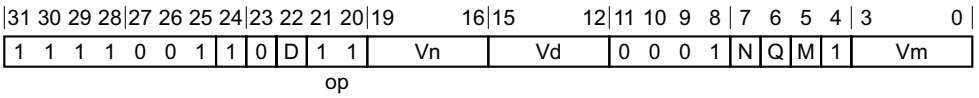
## F7.1.38 VBIF

Vector Bitwise Insert if False inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VBIF{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

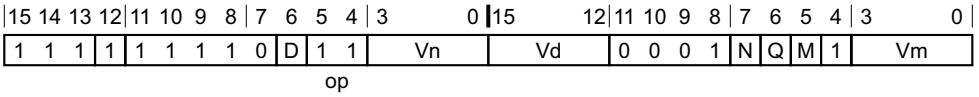
Applies when `Q = 1`.

```
VBIF{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE VEOR;
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VBIF{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VBIF{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE VEOR;
if op == '01' then operation = VBitOps_VBSL;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VBIF, VBIT, or VBSL instruction must be unconditional. ARM strongly recommends that a T32 VBIF, VBIT, or VBSL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

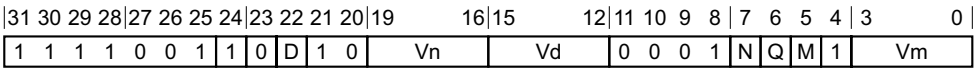
## F7.1.39 VBIT

Vector Bitwise Insert if True inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VBIT{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

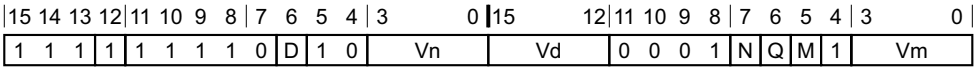
Applies when `Q = 1`.

```
VBIT{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE VEOR;
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VBIT{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VBIT{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE VEOR;
if op == '01' then operation = VBitOps_VBSL;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VBIF, VBIT, or VBSL instruction must be unconditional. ARM strongly recommends that a T32 VBIF, VBIT, or VBSL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

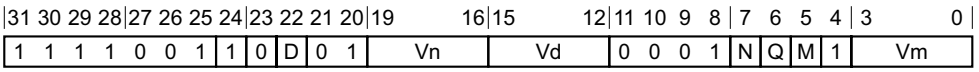
## F7.1.40 VBSL

Vector Bitwise Select sets each bit in the destination to the corresponding bit from the first source operand when the original destination bit was 1, otherwise from the second source operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VBSL{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

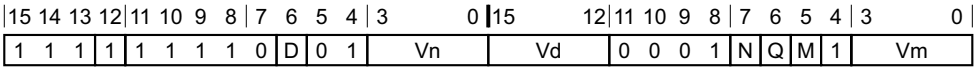
Applies when `Q = 1`.

```
VBSL{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE VEOR;
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VBSL{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VBSL{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE VEOR;
if op == '01' then operation = VBitOps_VBSL;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VBIF, VBIT, or VBSL instruction must be unconditional. ARM strongly recommends that a T32 VBIF, VBIT, or VBSL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.41 VCEQ (immediate #0)

Vector Compare Equal to Zero takes each element in a vector, and compares it with zero. If it is equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

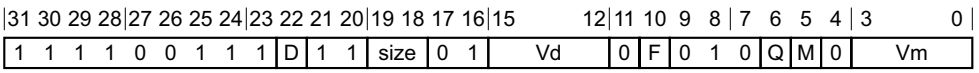
- 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.
- 32-bit floating-point numbers.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

#### *128-bit SIMD vector variant*

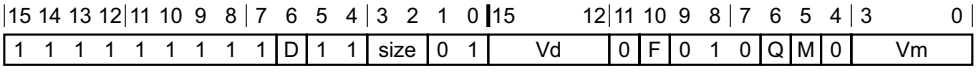
Applies when `Q = 1`.

```
VCEQ{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCEQ{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

### Decode for all variants of this encoding

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCEQ instruction must be unconditional. ARM strongly recommends that a T32 VCEQ instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

I8 when F = 0, size = 00

I16 when F = 0, size = 01

I32 when F = 0, size = 10

F32 when F = 1, size = 10

The following encodings are reserved:

- F = 0, size = 11.
- F = 1, size = 0x.
- F = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareEQ(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            else
                test_passed = (Elem[D[m+r],e,esize] == Zeros(esize));
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.42 VCEQ (register)

Vector Compare Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If they are equal, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

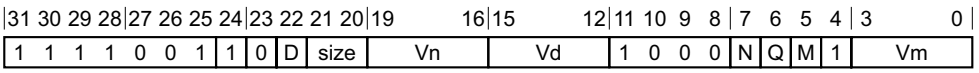
- 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.
- 32-bit floating-point numbers.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

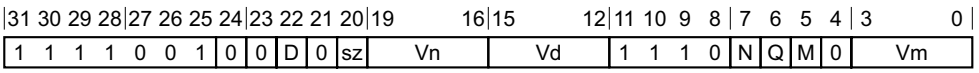
Applies when `Q = 1`.

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### A2




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

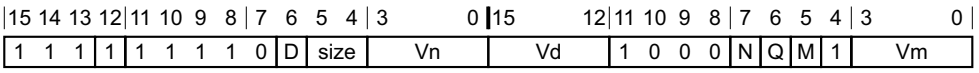

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
int_operation = FALSE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## T1




### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

### 128-bit SIMD vector variant

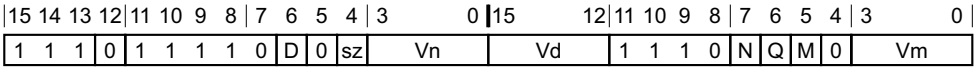
Applies when `Q = 1`.

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## T2




### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

### 128-bit SIMD vector variant

Applies when `Q = 1`.

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
int_operation = FALSE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCEQ instruction must be unconditional. ARM strongly recommends that a T32 VCEQ instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> The data types for the elements of the operands. It must be one of:

- I8 Encoding T1/A1, size = 0b00.
- I16 Encoding T1/A1, size = 0b01.
- I32 Encoding T1/A1, size = 0b10.
- F32 Encoding T2/A2, sz = 0.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if int_operation then
                test_passed = (op1 == op2);
            else
                test_passed = FPCompareEQ(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F7.1.43 VCGE (immediate #0)

Vector Compare Greater Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is greater than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

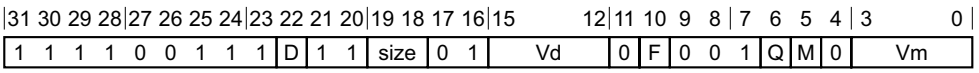
- 8-bit, 16-bit, or 32-bit signed integers.
- 32-bit floating-point numbers.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

#### A1




##### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCGE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

##### *128-bit SIMD vector variant*

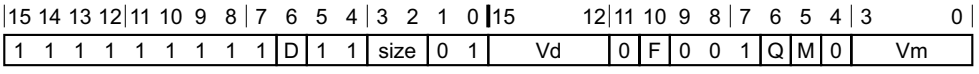
Applies when `Q = 1`.

```
VCGE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

##### *Decode for all variants of this encoding*

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

#### T1




##### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCGE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

##### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCGE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

### Decode for all variants of this encoding

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCGE instruction must be unconditional. ARM strongly recommends that a T32 VCGE instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

S8 when F = 0, size = 00

S16 when F = 0, size = 01

S32 when F = 0, size = 10

F32 when F = 1, size = 10

The following encodings are reserved:

- F = 0, size = 11.
- F = 1, size = 0x.
- F = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGE(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) >= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F7.1.44 VCGE (register)

Vector Compare Greater Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 32-bit floating-point numbers.

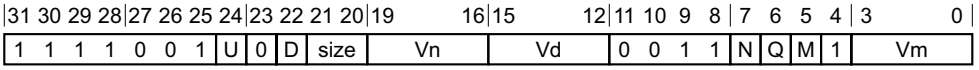
The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

This instruction is used by the alias VCLE (register). See the *Alias conditions* on page F7-3296 table for details of when each alias is preferred.

#### A1




##### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

##### *128-bit SIMD vector variant*

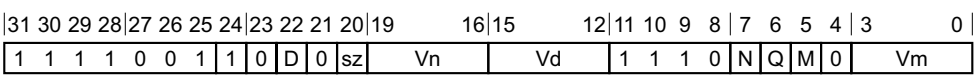
Applies when `Q = 1`.

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

##### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
type = if U == '1' then VCGEtype_unsigned else VCGEtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

#### A2




##### *64-bit SIMD vector variant*

Applies when `Q = 0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCGE{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### 128-bit SIMD vector variant

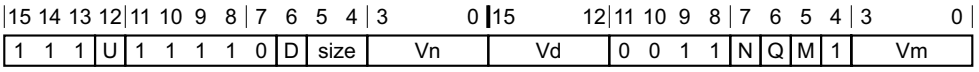
Applies when `Q = 1`.

```
VCGE{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
type = VCGEtype_fp;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### 128-bit SIMD vector variant

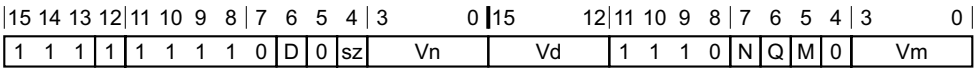
Applies when `Q = 1`.

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
type = if U == '1' then VCGEtype_unsigned else VCGEtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T2




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VCGE{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### 128-bit SIMD vector variant

Applies when `Q = 1`.

```
VCGE{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
type = VCGEtype_fp;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| VCLE (register) | Never |

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCGE instruction must be unconditional. ARM strongly recommends that a T32 VCGE instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8 when U = 0, size = 00

S16 when U = 0, size = 01

S32 when U = 0, size = 10

U8 when U = 1, size = 00

U16 when U = 1, size = 01

U32 when U = 1, size = 10

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
enumeration VCGEtype {VCGEtype_signed, VCGEtype_unsigned, VCGEtype_fp};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            case type of
                when VCGEtype_signed    test_passed = (SInt(op1) >= SInt(op2));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when VCGEtype_unsigned  test_passed = (UInt(op1) >= UInt(op2));
        when VCGEtype_fp        test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
    Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.45 VCGT (immediate #0)

Vector Compare Greater Than Zero takes each element in a vector, and compares it with zero. If it is greater than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

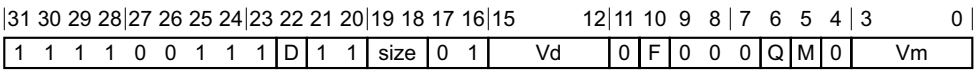
- 8-bit, 16-bit, or 32-bit signed integers.
- 32-bit floating-point numbers.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCGT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

#### *128-bit SIMD vector variant*

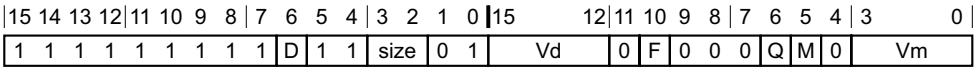
Applies when `Q = 1`.

```
VCGT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCGT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCGT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

### *Decode for all variants of this encoding*

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCGT instruction must be unconditional. ARM strongly recommends that a T32 VCGT instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

S8 when F = 0, size = 00

S16 when F = 0, size = 01

S32 when F = 0, size = 10

F32 when F = 1, size = 10

The following encodings are reserved:

- F = 0, size = 11.
- F = 1, size = 0x.
- F = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGT(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) > 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.46 VCGT (register)

Vector Compare Greater Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 32-bit floating-point numbers.

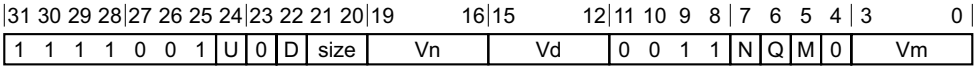
The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

This instruction is used by the alias VCLT (register). See the *Alias conditions* on page F7-3302 table for details of when each alias is preferred.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

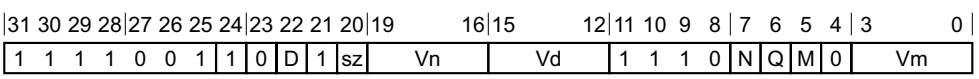
Applies when `Q = 1`.

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
type = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### A2




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCGT{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### 128-bit SIMD vector variant

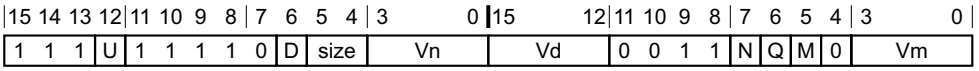
Applies when `Q = 1`.

```
VCGT{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
type = VCGTtype_fp;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### 128-bit SIMD vector variant

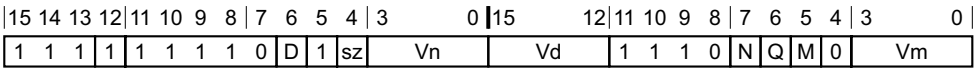
Applies when `Q = 1`.

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
type = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T2




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VCGT{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### 128-bit SIMD vector variant

Applies when `Q = 1`.

```
VCGT{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
type = VCGTtype_fp;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Alias conditions

| Alias | is preferred when |
|---|---|
| VCLT (register) | Never |

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCGT instruction must be unconditional. ARM strongly recommends that a T32 VCGT instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8 when U = 0, size = 00

S16 when U = 0, size = 01

S32 when U = 0, size = 10

U8 when U = 1, size = 00

U16 when U = 1, size = 01

U32 when U = 1, size = 10

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            case type of
                when VCGTtype_signed    test_passed = (SInt(op1) > SInt(op2));
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when VCGTtype_unsigned  test_passed = (UInt(op1) > UInt(op2));
        when VCGTtype_fp        test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
    Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.47 VCLE (immediate #0)

Vector Compare Less Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is less than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

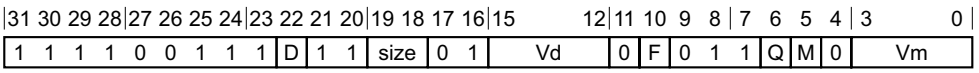
- 8-bit, 16-bit, or 32-bit signed integers.
- 32-bit floating-point numbers.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

#### *128-bit SIMD vector variant*

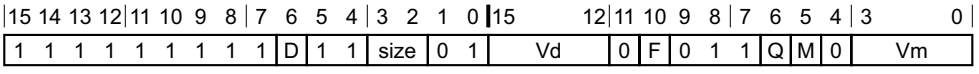
Applies when `Q = 1`.

```
VCLE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCLE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

### Decode for all variants of this encoding

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCLE instruction must be unconditional. ARM strongly recommends that a T32 VCLE instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

S8 when F = 0, size = 00

S16 when F = 0, size = 01

S32 when F = 0, size = 10

F32 when F = 1, size = 10

The following encodings are reserved:

- F = 0, size = 11.
- F = 1, size = 0x.
- F = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGE(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) <= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

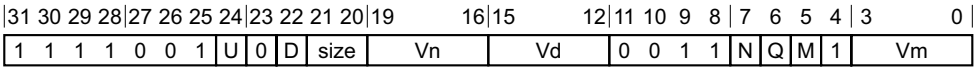
## F7.1.48 VCLE (register)

Vector Compare Less Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros

This instruction is an alias of the VCGE (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VCGE (register).
- The description of VCGE (register) gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

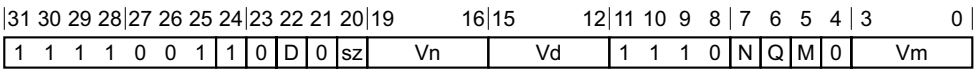
```
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

### A2




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLE{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.F32 <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

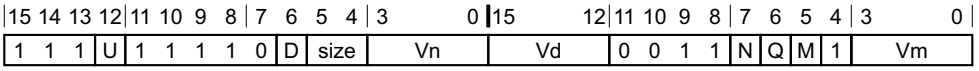
`VCLE{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>`

is equivalent to

`VCGE{<c>}{<q>}.F32 <Qd>, <Qm>, <Qn>`

and is never the preferred disassembly.

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

`VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

is equivalent to

`VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>`

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

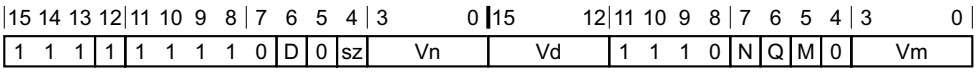
`VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>`

is equivalent to

`VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>`

and is never the preferred disassembly.

### T2




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

`VCLE{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>`

is equivalent to

`VCGE{<c>}{<q>}.F32 <Dd>, <Dm>, <Dn>`

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

`VCLE{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>`

is equivalent to


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCGE{<c>}{<q>}.F32 <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

### Assembler symbols

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCGE instruction must be unconditional. ARM strongly recommends that a T32 VCGE instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8 when U = 0, size = 00

S16 when U = 0, size = 01

S32 when U = 0, size = 10

U8 when U = 1, size = 00

U16 when U = 1, size = 01

U32 when U = 1, size = 10

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

### Operation for all encodings

The description of VCGE (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.49 VCLS

Vector Count Leading Sign Bits counts the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector. The count does not include the topmost bit itself.

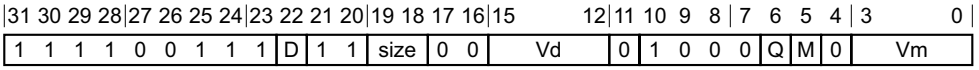
The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit signed integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

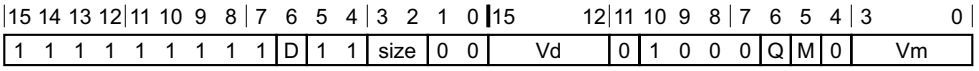
Applies when `Q = 1`.

```
VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCLS instruction must be unconditional. ARM strongly recommends that a T32 VCLS instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

S8 when size = 00

S16 when size = 01

S32 when size = 10

The encoding size = 11 is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingSignBits(Elem[D[m+r],e,esize])<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.50 VCLT (immediate #0)

Vector Compare Less Than Zero takes each element in a vector, and compares it with zero. If it is less than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

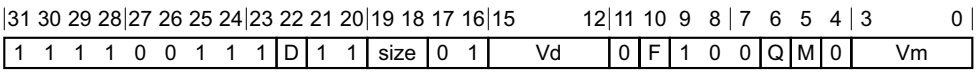
- 8-bit, 16-bit, or 32-bit signed integers.
- 32-bit floating-point numbers.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

#### *128-bit SIMD vector variant*

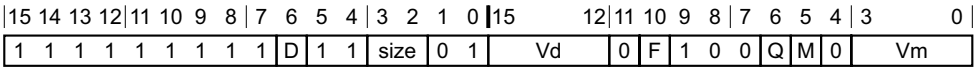
Applies when `Q = 1`.

```
VCLT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCLT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

### Decode for all variants of this encoding

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCLT instruction must be unconditional. ARM strongly recommends that a T32 VCLT instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

S8 when F = 0, size = 00

S16 when F = 0, size = 01

S32 when F = 0, size = 10

F32 when F = 1, size = 10

The following encodings are reserved:

- F = 0, size = 11.
- F = 1, size = 0x.
- F = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGT(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) < 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

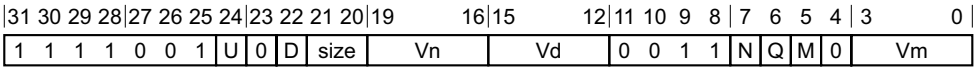
## F7.1.51 VCLT (register)

Vector Compare Less Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros

This instruction is an alias of the VCGT (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VCGT (register).
- The description of VCGT (register) gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

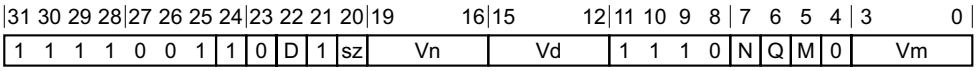
```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

### A2




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLT{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.F32 <Dd>, <Dm>, <Dn>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

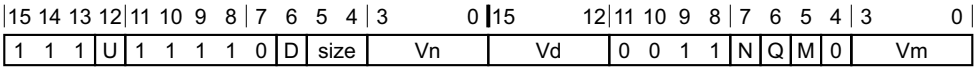
`VCLT{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>`

is equivalent to

`VCGT{<c>}{<q>}.F32 <Qd>, <Qm>, <Qn>`

and is never the preferred disassembly.

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

`VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

is equivalent to

`VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>`

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

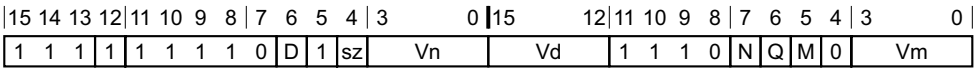
`VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>`

is equivalent to

`VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>`

and is never the preferred disassembly.

### T2




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

`VCLT{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>`

is equivalent to

`VCGT{<c>}{<q>}.F32 <Dd>, <Dm>, <Dn>`

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

`VCLT{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>`

is equivalent to


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VCGT{<c>}{<q>}.F32 <Qd>, <Qm>, <Qn>
```

and is never the preferred disassembly.

### Assembler symbols

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCGT instruction must be unconditional. ARM strongly recommends that a T32 VCGT instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8 when U = 0, size = 00

S16 when U = 0, size = 01

S32 when U = 0, size = 10

U8 when U = 1, size = 00

U16 when U = 1, size = 01

U32 when U = 1, size = 10

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

### Operation for all encodings

The description of VCGT (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.52 VCLZ

Vector Count Leading Zeros counts the number of consecutive zeros, starting from the most significant bit, in each element in a vector, and places the results in a second vector.

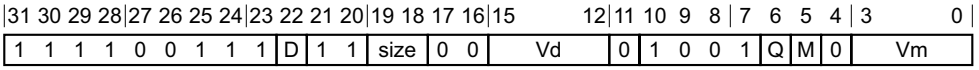
The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

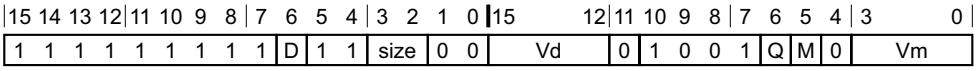
Applies when `Q = 1`.

```
VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCLZ instruction must be unconditional. ARM strongly recommends that a T32 VCLZ instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

- I8 when size = 00
- I16 when size = 01
- I32 when size = 10

The encoding size = 11 is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingZeroBits(Elem[D[m+r],e,esize])<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

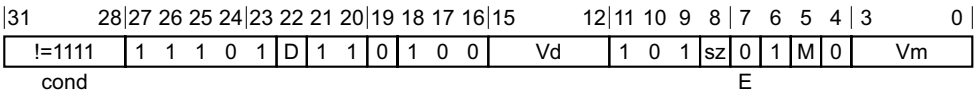
## F7.1.53 VCMP

Vector Compare compares two floating-point registers, or one floating-point register and zero. It writes the result to the FPSCR flags. These are normally transferred to the PSTATE.{N, Z, C, V} Condition flags by a subsequent VMRS instruction.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when sz = 0.

```
VCMP{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

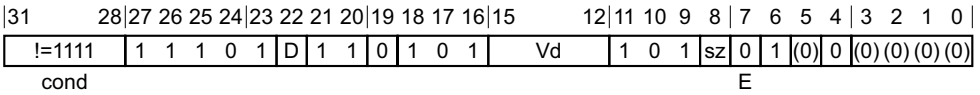
Applies when sz = 1.

```
VCMP{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### A2




#### *Single-precision scalar variant*

Applies when sz = 0.

```
VCMP{<c>}{<q>}.F32 <Sd>, #0.0
```

#### *Double-precision scalar variant*

Applies when sz = 1.

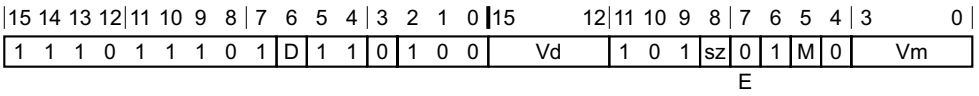
```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

#### *Decode for all variants of this encoding*

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCMP{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

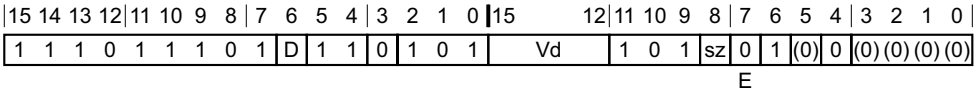
Applies when `sz = 1`.

```
VCMP{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCMP{<c>}{<q>}.F32 <Sd>, #0.0
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

#### *Decode for all variants of this encoding*

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| `<Sm>` | Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

VCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        bits(64) op64 = if with_zero then FPZero('0') else D[m];
        FPSCR.<N,Z,C,V> = FPCompare(D[d], op64, quiet_nan_exc, FPSCR);
    else
        bits(32) op32 = if with_zero then FPZero('0') else S[m];
        FPSCR.<N,Z,C,V> = FPCompare(S[d], op32, quiet_nan_exc, FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

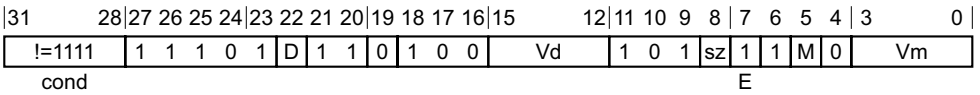
## F7.1.54 VCMPE

Vector Compare, raising Invalid Operation on NaN compares two floating-point registers, or one floating-point register and zero. It writes the result to the FPSCR flags. These are normally transferred to the PSTATE.{N, Z, C, V} Condition flags by a subsequent VMRS instruction.

It raises an Invalid Operation exception if either operand is any type of NaN, or if either operand is a signaling NaN.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when sz = 0.

```
VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

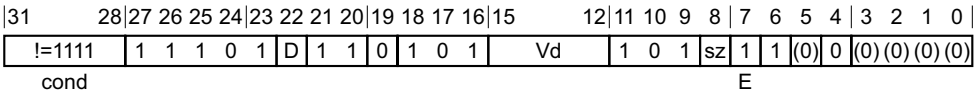
Applies when sz = 1.

```
VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### A2




#### *Single-precision scalar variant*

Applies when sz = 0.

```
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0
```

#### *Double-precision scalar variant*

Applies when sz = 1.

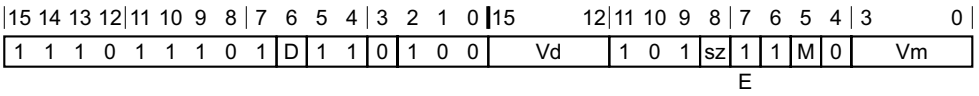
```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

#### *Decode for all variants of this encoding*

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

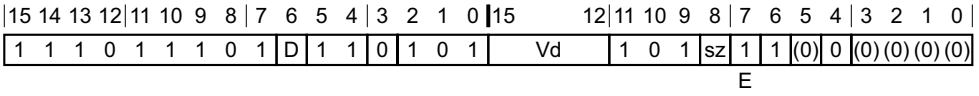
Applies when `sz = 1`.

```
VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

#### *Decode for all variants of this encoding*

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| `<Sm>` | Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

VCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        bits(64) op64 = if with_zero then FPZero('0') else D[m];
        FPSCR.<N,Z,C,V> = FPCompare(D[d], op64, quiet_nan_exc, FPSCR);
    else
        bits(32) op32 = if with_zero then FPZero('0') else S[m];
        FPSCR.<N,Z,C,V> = FPCompare(S[d], op32, quiet_nan_exc, FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.55 VCNT

Vector Count Set Bits counts the number of bits that are one in each element in a vector, and places the results in a second vector.

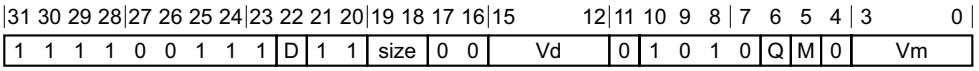
The operand vector elements must be 8-bit fields.

The result vector elements are 8-bit integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCNT{<c>}{<q>}.8 <Dd>, <Dm> // Encoded as Q = 0
```

#### *128-bit SIMD vector variant*

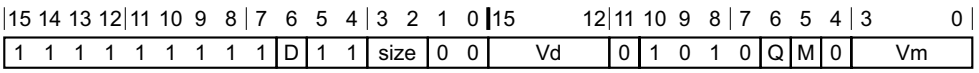
Applies when `Q = 1`.

```
VCNT{<c>}{<q>}.8 <Qd>, <Qm> // Encoded as Q = 1
```

#### *Decode for all variants of this encoding*

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8;  elements = 8;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCNT{<c>}{<q>}.8 <Dd>, <Dm> // Encoded as Q = 0
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VCNT{<c>}{<q>}.8 <Qd>, <Qm> // Encoded as Q = 1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8;  elements = 8;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. An A32 VCNT instruction must be unconditional. ARM strongly recommends that a T32 VCNT instruction is unconditional, see *Conditional execution* on page F2-2447. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qm> | Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = BitCount(Elem[D[m+r],e,esize])<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

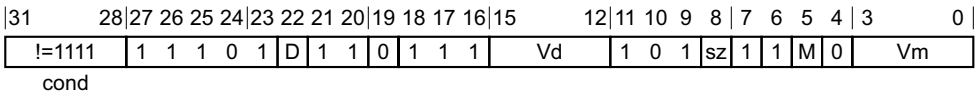
## F7.1.56 VCVT (between double-precision and single-precision)

Convert between double-precision and single-precision does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Encoding*

Applies when `sz = 0`.

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

#### *Encoding*

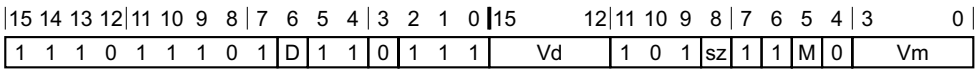
Applies when `sz = 1`.

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
double_to_single = (sz == '1');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Encoding*

Applies when `sz = 0`.

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

#### *Encoding*

Applies when `sz = 1`.

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
double_to_single = (sz == '1');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if double_to_single then
        S[d] = FPConvert(D[m], FPSCR);
    else
        D[d] = FPConvert(S[m], FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.57 VCVT (between half-precision and single-precision, Advanced SIMD)

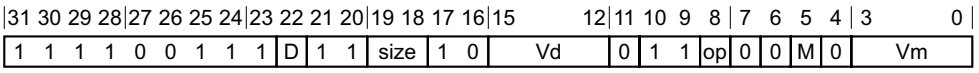
Vector Convert between half-precision and single-precision converts each element in a vector from single-precision to half-precision floating-point, or from half-precision to single-precision, and places the results in a second vector.

The vector elements must be 32-bit floating-point numbers, or 16-bit floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Encoding*

Applies when op = 0.

```
VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm> // Encoded as op = 0
```

#### *Encoding*

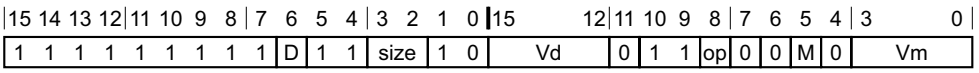
Applies when op = 1.

```
VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm> // Encoded as op = 1
```

#### *Decode for all variants of this encoding*

```
if size != '01' then UNDEFINED;
half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16;  elements = 4;
m = UInt(M:Vm);  d = UInt(D:Vd);
```

### T1




#### *Encoding*

Applies when op = 0.

```
VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm> // Encoded as op = 0
```

#### *Encoding*

Applies when op = 1.

```
VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm> // Encoded as op = 1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size != '01' then UNDEFINED;
half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16;  elements = 4;
m = UInt(M:Vm);  d = UInt(D:Vd);
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VCVT instruction must be unconditional. ARM strongly recommends that a T32 VCVT instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if half_to_single then
            Elem[Q[d>>1],e,32] = FPConvert(Elem[Din[m],e,16], StandardFPSCRValue());
        else
            Elem[D[d],e,16] = FPConvert(Elem[Qin[m>>1],e,32], StandardFPSCRValue());
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.58 VCVT (between floating-point and integer, Advanced SIMD)

Vector Convert between floating-point and integer converts each element in a vector from floating-point to integer, or from integer to floating-point, and places the results in a second vector.

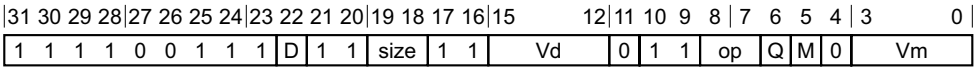
The vector elements must be 32-bit floating-point numbers, or 32-bit integers. Signed and unsigned integers are distinct.

The floating-point to integer operation uses the Round towards Zero rounding mode. The integer to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

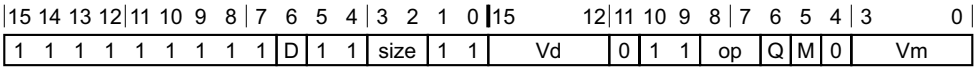
Applies when `Q = 1`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
to_integer = (op<1> == '1');  unsigned = (op<0> == '1');
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
to_integer = (op<1> == '1');  unsigned = (op<0> == '1');
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VCVT instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VCVT instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt1> Is the data type for the elements of the destination vector, encoded in the "op" field. It can have the following values:

| | |
|---|---|
| F32 | when op = 00 |
| F32 | when op = 01 |
| S32 | when op = 10 |
| U32 | when op = 11 |

<dt2> Is the data type for the elements of the source vector, encoded in the "op" field. It can have the following values:

| | |
|---|---|
| S32 | when op = 00 |
| U32 | when op = 01 |
| F32 | when op = 10 |
| F32 | when op = 11 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[m+r],e,esize];
            if to_integer then
                result = FPToFixed(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_ZERO);
            else
                result = FixedToFP(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_TIEEVEN);
            Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

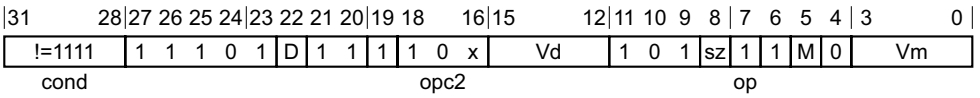
## F7.1.59 VCVT (floating-point to integer, floating-point)

Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `opc2 = 100 && sz = 0`.

```
VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

#### *Single-precision scalar variant*

Applies when `opc2 = 101 && sz = 0`.

```
VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `opc2 = 100 && sz = 1`.

```
VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

#### *Double-precision scalar variant*

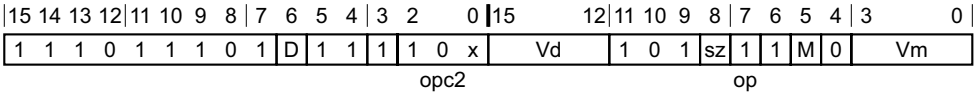
Applies when `opc2 = 101 && sz = 1`.

```
VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
to_integer = (opc2<2> == '1');  dp_operation = (sz == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Single-precision scalar variant*

Applies when `opc2 = 100 && sz = 0`.

```
VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

#### *Single-precision scalar variant*

Applies when `opc2 = 101 && sz = 0`.

```
VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `opc2 = 100 && sz = 1`.

```
VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

#### *Double-precision scalar variant*

Applies when `opc2 = 101 && sz = 1`.

```
VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
to_integer = (opc2<2> == '1');  dp_operation = (sz == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

### Notes for all encodings

Related encodings: Floating-point data-processing instructions.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| `<Sm>` | Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field. |
| `<Dm>` | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        if dp_operation then
            S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
        else
            S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
    else
        if dp_operation then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        D[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
    else
        S[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

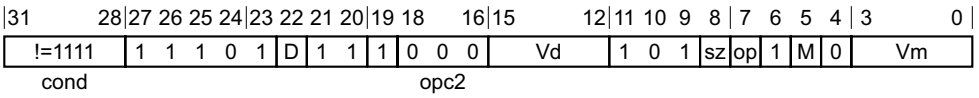
## F7.1.60 VCVT (integer to floating-point, floating-point)

Convert integer to floating-point converts a 32-bit integer to floating-point using the rounding mode specified by the FPSCR, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

#### *Double-precision scalar variant*

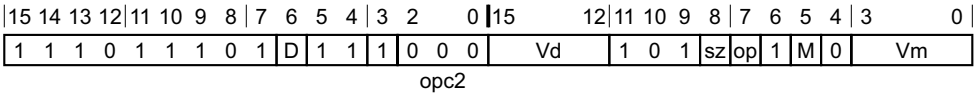
Applies when `sz = 1`.

```
VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>
```

#### *Decode for all variants of this encoding*

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
to_integer = (opc2<2> == '1');  dp_operation = (sz == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
to_integer = (opc2<2> == '1');  dp_operation = (sz == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

## Notes for all encodings

Related encodings: Floating-point data-processing instructions.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` Is the data type for the operand, encoded in the "op" field. It can have the following values:

- `U32` when `op = 0`
- `S32` when `op = 1`

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        if dp_operation then
            S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
        else
            S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
    else
        if dp_operation then
            D[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
        else
            S[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.61 VCVT (between floating-point and fixed-point, Advanced SIMD)

Vector Convert between floating-point and fixed-point converts each element in a vector from floating-point to fixed-point, or from fixed-point to floating-point, and places the results in a second vector.

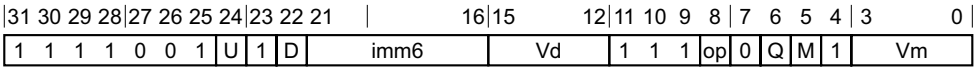
The vector elements must be 32-bit floating-point numbers, or 32-bit integers. Signed and unsigned integers are distinct.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `imm6 != 000xxx && Q = 0`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>
```

#### *128-bit SIMD vector variant*

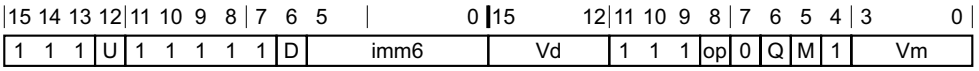
Applies when `imm6 != 000xxx && Q = 1`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>
```

#### *Decode for all variants of this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op == '1');  frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');  esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `imm6 != 000xxx && Q = 0`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>
```

#### *128-bit SIMD vector variant*

Applies when `imm6 != 000xxx && Q = 1`.

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op == '1');  frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');  esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VCVT instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VCVT instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt1> Is the data type for the elements of the destination vector, encoded in the "op:U" field. It can have the following values:

- F32 when op = 0, U = 0
- F32 when op = 0, U = 1
- S32 when op = 1, U = 0
- U32 when op = 1, U = 1

<dt2> Is the data type for the elements of the source vector, encoded in the "op:U" field. It can have the following values:

- S32 when op = 0, U = 0
- U32 when op = 0, U = 1
- F32 when op = 1, U = 0
- F32 when op = 1, U = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<fbits> The number of fraction bits in the fixed point number, in the range 1 to 32:

- (64 - <fbits>) is encoded in imm6.

An assembler can permit an <fbits> value of 0. This is encoded as floating-point to integer or integer to floating-point instruction, see VCVT (between floating-point and integer, Advanced SIMD).

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[m+r],e,esize];
            if to_fixed then
                result = FPToFixed(op1, frac_bits, unsigned, StandardFPSCRValue(),
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                        FPRounding_ZERO);
else
    result = FixedToFP(op1, frac_bits, unsigned, StandardFPSCRValue(),
                        FPRounding_TIEEVEN);
Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F7.1.62 VCVT (between floating-point and fixed-point, floating-point)

Convert between floating-point and fixed-point converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point. Software can specify the fixed-point value as either signed or unsigned.

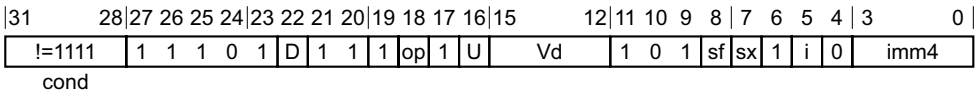
The floating-point value can be single-precision or double-precision.

The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

#### A1




##### *Single-precision scalar variant*

Applies when `op = 0 && sf = 0`.

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

##### *Single-precision scalar variant*

Applies when `op = 1 && sf = 0`.

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

##### *Double-precision scalar variant*

Applies when `op = 0 && sf = 1`.

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

##### *Double-precision scalar variant*

Applies when `op = 1 && sf = 1`.

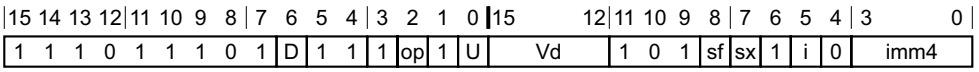
```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

##### *Decode for all variants of this encoding*

```
to_fixed = (op == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
dp_operation = (sf == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
if frac_bits < 0 then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *Single-precision scalar variant*

Applies when `op = 0 && sf = 0`.

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

#### *Single-precision scalar variant*

Applies when `op = 1 && sf = 0`.

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

#### *Double-precision scalar variant*

Applies when `op = 0 && sf = 1`.

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

#### *Double-precision scalar variant*

Applies when `op = 1 && sf = 1`.

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

#### *Decode for all variants of this encoding*

```
to_fixed = (op == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
dp_operation = (sf == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
if frac_bits < 0 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VCVT (between floating-point and fixed-point)* on page K1-5485.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` Is the data type for the fixed-point number, encoded in the "U:sx" field. It can have the following values:

| | |
|---|---|
| `S16` | when `U = 0`, `sx = 0` |
| `S32` | when `U = 0`, `sx = 1` |
| `U16` | when `U = 1`, `sx = 0` |
| `U32` | when `U = 1`, `sx = 1` |

`<Sdm>` Is the 32-bit name of the SIMD&FP destination and source register, encoded in the "Vd:D" field.

`<Ddm>` Is the 64-bit name of the SIMD&FP destination and source register, encoded in the "D:Vd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<fbits> The number of fraction bits in the fixed-point number:

- If <dt> is S16 or U16, <fbits> must be in the range 0-16. (16 - <fbits>) is encoded in [imm4, i]
- If <dt> is S32 or U32, <fbits> must be in the range 1-32. (32 - <fbits>) is encoded in [imm4, i].

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_fixed then
        bits(size) result;
        if dp_operation then
            result = FPToFixed(D[d], frac_bits, unsigned, FPSCR, FPRounding_ZERO);
            D[d] = Extend(result, 64, unsigned);
        else
            result = FPToFixed(S[d], frac_bits, unsigned, FPSCR, FPRounding_ZERO);
            S[d] = Extend(result, 32, unsigned);
    else
        if dp_operation then
            D[d] = FixedToFP(D[d]<size-1:0>, frac_bits, unsigned, FPSCR, FPRounding_TIEEVEN);
        else
            S[d] = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR, FPRounding_TIEEVEN);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.63 VCVTA (Advanced SIMD)

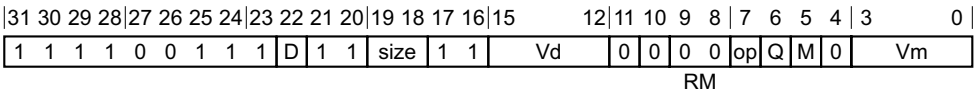
Vector Convert floating-point to integer with Round to Nearest with Ties to Away converts each element in a vector from floating-point to integer using the Round to Nearest with Ties to Away rounding mode, and places the results in a second vector.

The operand vector elements must be 32-bit floating-point numbers.

The result vector elements are 32-bit integers. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCVTA{<q>}.<dt>.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

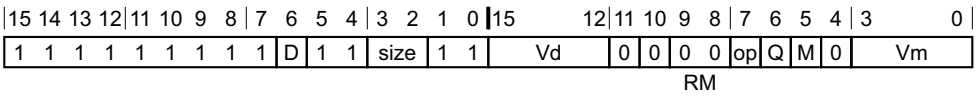
Applies when `Q = 1`.

```
VCVTA{<q>}.<dt>.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCVTA{<q>}.<dt>.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VCVTA{<q>}.<dt>.F32 <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

## Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> For encoding A1: is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

For encoding T1: is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```

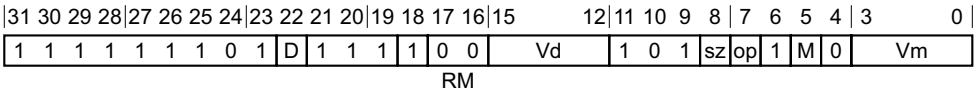

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.64 VCVTA (floating-point)

Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

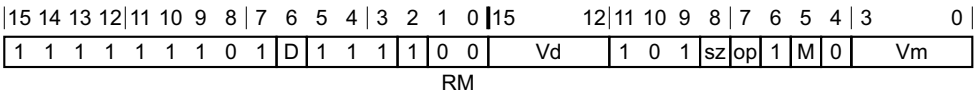
Applies when `sz = 1`.

```
VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  unsigned = (op == '0');  dp_operation = (sz == '1');
d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  unsigned = (op == '0');  dp_operation = (sz == '1');
d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if dp_operation then
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
else
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

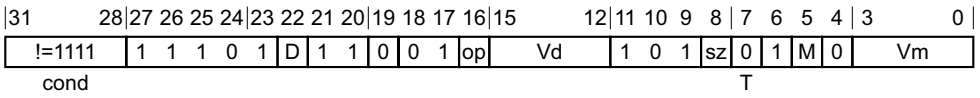
## F7.1.65 VCVTB

Convert to or from a half-precision value in the bottom half of a single-precision register does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Encoding*

Applies when `op = 0 && sz = 0`.

```
VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

#### *Encoding*

Applies when `op = 0 && sz = 1`.

```
VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

#### *Encoding*

Applies when `op = 1 && sz = 0`.

```
VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

#### *Encoding*

Applies when `op = 1 && sz = 1`.

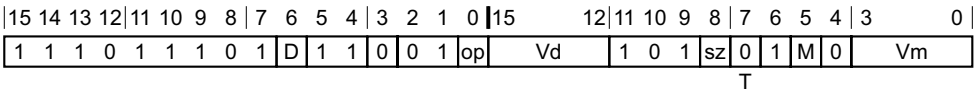
```
VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); m = UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *Encoding*

Applies when `op = 0 && sz = 0`.

```
VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

#### *Encoding*

Applies when `op = 0 && sz = 1`.

```
VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

#### *Encoding*

Applies when `op = 1 && sz = 0`.

```
VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

#### *Encoding*

Applies when `op = 1 && sz = 1`.

```
VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); m = UInt(Vm:M);
```

## Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| `<Dm>` | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| `<Sm>` | Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        D[d] = FPConvert(hp, FPSCR);
    else
        S[d] = FPConvert(hp, FPSCR);
else
    if uses_double then
        hp = FPConvert(D[m], FPSCR);
    else
        hp = FPConvert(S[m], FPSCR);
    S[d]<lowbit+15:lowbit> = hp;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.66 VCVTM (Advanced SIMD)

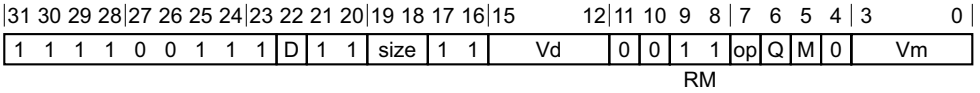
Vector Convert floating-point to integer with Round towards -Infinity converts each element in a vector from floating-point to integer using the Round towards -Infinity rounding mode, and places the results in a second vector.

The operand vector elements must be 32-bit floating-point numbers.

The result vector elements are 32-bit integers. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCVTM{<q>}.<dt>.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

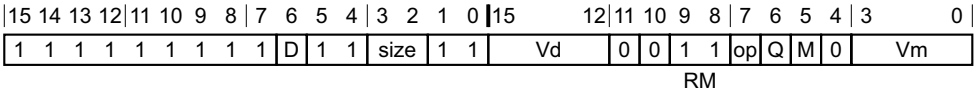
Applies when `Q = 1`.

```
VCVTM{<q>}.<dt>.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCVTM{<q>}.<dt>.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VCVTM{<q>}.<dt>.F32 <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

## Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> For encoding A1: is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

For encoding T1: is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```

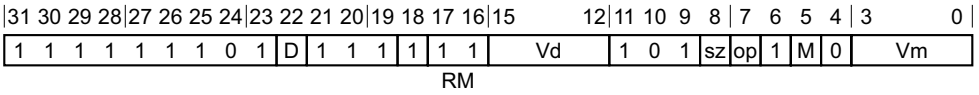

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.67 VCVTM (floating-point)

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

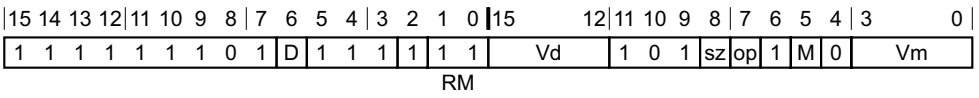
Applies when `sz = 1`.

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  unsigned = (op == '0');  dp_operation = (sz == '1');
d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  unsigned = (op == '0');  dp_operation = (sz == '1');
d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if dp_operation then
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
else
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.68 VCVTN (Advanced SIMD)

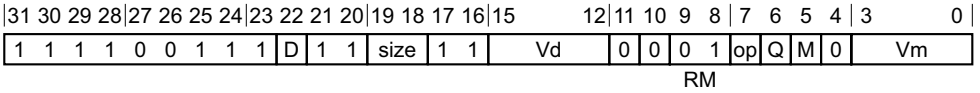
Vector Convert floating-point to integer with Round to Nearest converts each element in a vector from floating-point to integer using the Round to Nearest rounding mode, and places the results in a second vector.

The operand vector elements must be 32-bit floating-point numbers.

The result vector elements are 32-bit integers. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCVTN{<q>}.<dt>.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

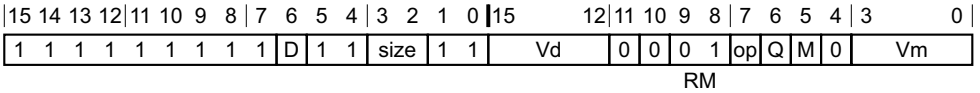
Applies when `Q = 1`.

```
VCVTN{<q>}.<dt>.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCVTN{<q>}.<dt>.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VCVTN{<q>}.<dt>.F32 <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

## Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> For encoding A1: is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

For encoding T1: is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```

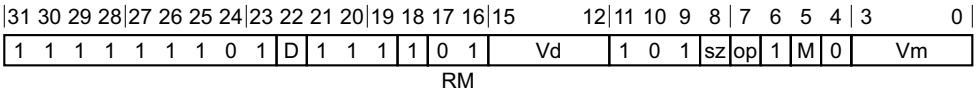

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.69 VCVTN (floating-point)

Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

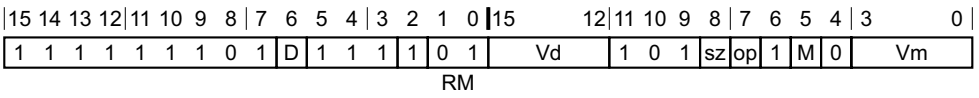
Applies when `sz = 1`.

```
VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  unsigned = (op == '0');  dp_operation = (sz == '1');
d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  unsigned = (op == '0');  dp_operation = (sz == '1');
d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if dp_operation then
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
else
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.70 VCVTP (Advanced SIMD)

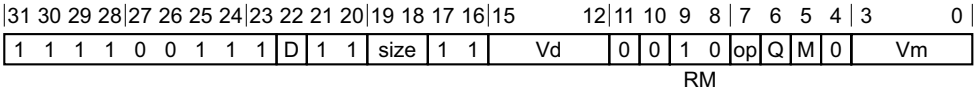
Vector Convert floating-point to integer with Round towards +Infinity converts each element in a vector from floating-point to integer using the Round towards +Infinity rounding mode, and places the results in a second vector.

The operand vector elements must be 32-bit floating-point numbers.

The result vector elements are 32-bit integers. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCVTP{<q>}.<dt>.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

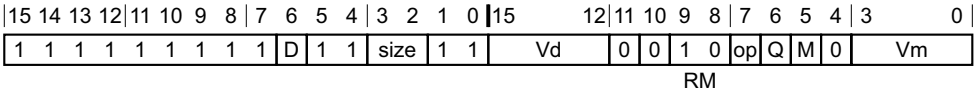
Applies when `Q = 1`.

```
VCVTP{<q>}.<dt>.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VCVTP{<q>}.<dt>.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VCVTP{<q>}.<dt>.F32 <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

## Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> For encoding A1: is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

For encoding T1: is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```

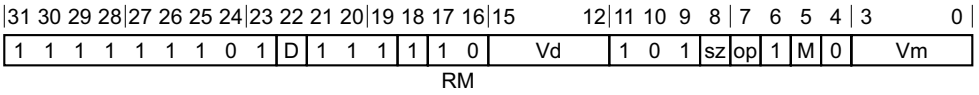

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.71 VCVTP (floating-point)

Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

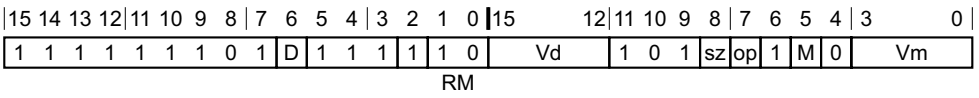
Applies when `sz = 1`.

```
VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  unsigned = (op == '0');  dp_operation = (sz == '1');
d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  unsigned = (op == '0');  dp_operation = (sz == '1');
d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

S32 when op = 0

U32 when op = 1

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if dp_operation then
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
else
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

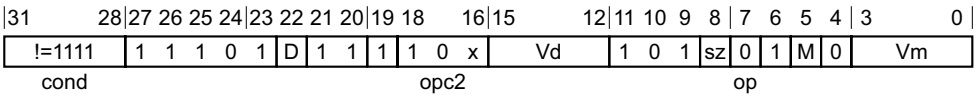
## F7.1.72 VCVTR

Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by the FPSCR and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `opc2 = 100 && sz = 0`.

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

#### *Single-precision scalar variant*

Applies when `opc2 = 101 && sz = 0`.

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `opc2 = 100 && sz = 1`.

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

#### *Double-precision scalar variant*

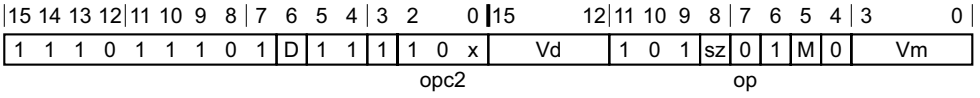
Applies when `opc2 = 101 && sz = 1`.

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
to_integer = (opc2<2> == '1');  dp_operation = (sz == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Single-precision scalar variant*

Applies when `opc2 = 100 && sz = 0`.

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

#### *Single-precision scalar variant*

Applies when `opc2 = 101 && sz = 0`.

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `opc2 = 100 && sz = 1`.

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

#### *Double-precision scalar variant*

Applies when `opc2 = 101 && sz = 1`.

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
to_integer = (opc2<2> == '1');  dp_operation = (sz == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR);
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

### Notes for all encodings

Related encodings: Floating-point data-processing instructions.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| `<Sm>` | Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field. |
| `<Dm>` | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        if dp_operation then
            S[d] = FPToFixed(D[m], 0, unsigned, FPSCR, rounding);
        else
            S[d] = FPToFixed(S[m], 0, unsigned, FPSCR, rounding);
    else
        if dp_operation then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        D[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
    else
        S[d] = FixedToFP(S[m], 0, unsigned, FPSCR, rounding);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

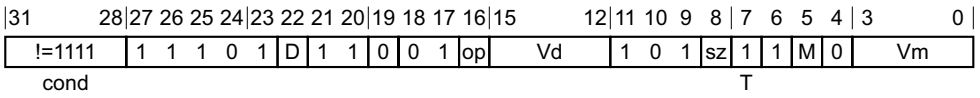
## F7.1.73 VCVTT

Convert to or from a half-precision value in the top half of a single-precision register does one of the following:

- Converts the half-precision value in the top half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the top half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Encoding*

Applies when `op = 0 && sz = 0`.

```
VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

#### *Encoding*

Applies when `op = 0 && sz = 1`.

```
VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

#### *Encoding*

Applies when `op = 1 && sz = 0`.

```
VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

#### *Encoding*

Applies when `op = 1 && sz = 1`.

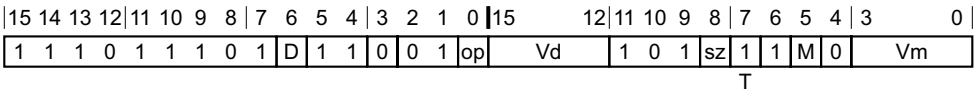
```
VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); m = UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *Encoding*

Applies when `op = 0 && sz = 0`.

```
VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

#### *Encoding*

Applies when `op = 0 && sz = 1`.

```
VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

#### *Encoding*

Applies when `op = 1 && sz = 0`.

```
VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

#### *Encoding*

Applies when `op = 1 && sz = 1`.

```
VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); m = UInt(Vm:M);
```

## Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<Sd>` | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| `<Dm>` | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| `<Sm>` | Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        D[d] = FPConvert(hp, FPSCR);
    else
        S[d] = FPConvert(hp, FPSCR);
else
    if uses_double then
        hp = FPConvert(D[m], FPSCR);
    else
        hp = FPConvert(S[m], FPSCR);
    S[d]<lowbit+15:lowbit> = hp;
```

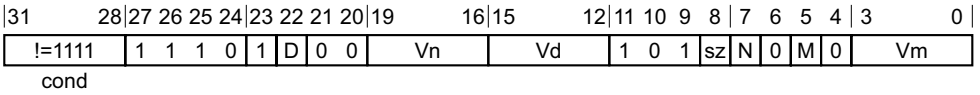

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.74 VDIV

Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VDIV{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

#### *Double-precision scalar variant*

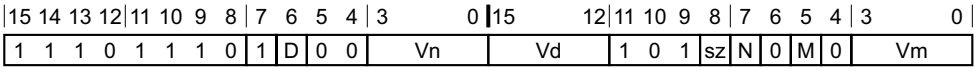
Applies when `sz = 1`.

```
VDIV{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VDIV{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VDIV{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        D[d] = FPDiv(D[n], D[m], FPSCR);
    else
        S[d] = FPDiv(S[n], S[m], FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F7.1.75 VDUP (general-purpose register)

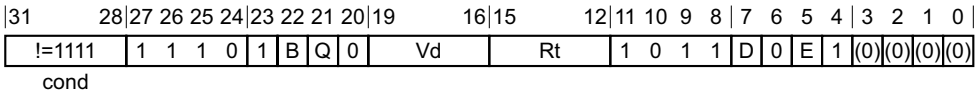
Duplicate general-purpose register to vector duplicates an element from a general-purpose register into every element of the destination vector.

The destination vector elements can be 8-bit, 16-bit, or 32-bit fields. The source element is the least significant 8, 16, or 32 bits of the general-purpose register. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

#### A1



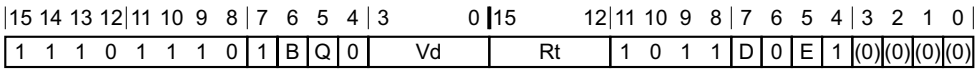

##### *A1 variant*

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // Encoded as Q = 1
VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // Encoded as Q = 0
```

##### *Decode for this encoding*

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);  regs = if Q == '0' then 1 else 2;
case B:E of
    when '00'  esize = 32;  elements = 2;
    when '01'  esize = 16;  elements = 4;
    when '10'  esize = 8;   elements = 8;
    when '11'  UNDEFINED;
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

#### T1




##### *T1 variant*

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // Encoded as Q = 1
VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // Encoded as Q = 0
```

##### *Decode for this encoding*

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);  regs = if Q == '0' then 1 else 2;
case B:E of
    when '00'  esize = 32;  elements = 2;
    when '01'  esize = 16;  elements = 4;
    when '10'  esize = 8;   elements = 8;
    when '11'  UNDEFINED;
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. ARM strongly recommends that any `VDUP` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` The data size for the elements of the destination vector. It must be one of:

- `8` Encoded as [b, e] = `0b10`.
- `16` Encoded as [b, e] = `0b01`.
- `32` Encoded as [b, e] = `0b00`.

`<Qd>` The destination vector for a quadword operation.

`<Dd>` The destination vector for a doubleword operation.

`<Rt>` The ARM source register.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = R[t]<esize-1:0>;
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.76 VDUP (scalar)

Duplicate vector element to vector duplicates a single element of a vector into every element of the destination vector.

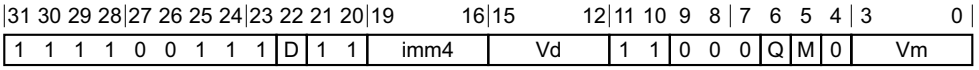
The scalar, and the destination vector elements, can be any one of 8-bit, 16-bit, or 32-bit fields. There is no distinction between data types.

For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Encoding*

Applies when `Q = 0`.

```
VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>
```

#### *Encoding*

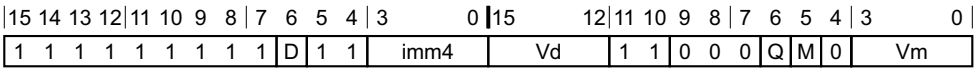
Applies when `Q = 1`.

```
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>
```

#### *Decode for all variants of this encoding*

```
if imm4 == 'x000' then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
case imm4 of
    when "xxx1"  esize = 8;  elements = 8;  index = UInt(imm4<3:1>);
    when "xx10"  esize = 16;  elements = 4;  index = UInt(imm4<3:2>);
    when "x100"  esize = 32;  elements = 2;  index = UInt(imm4<3>);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *Encoding*

Applies when `Q = 0`.

```
VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>
```

#### *Encoding*

Applies when `Q = 1`.

```
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if imm4 == 'x000' then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
case imm4 of
    when "xxx1"  esize = 8;  elements = 8;  index = UInt(imm4<3:1>);
    when "xx10"  esize = 16;  elements = 4;  index = UInt(imm4<3:2>);
    when "x100"  esize = 32;  elements = 2;  index = UInt(imm4<3>);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VDUP` instruction must be unconditional. ARM strongly recommends that a T32 `VDUP` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` The data size. It must be one of:

- `8` Encoded as imm4<0> = '1'. imm4<3:1> encodes the index`[x]` of the scalar.
- `16` Encoded as imm4<1:0> = '10'. imm4<3:2> encodes the index `[x]` of the scalar.
- `32` Encoded as imm4<2:0> = '100'. imm4<3> encodes the index `[x]` of the scalar.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm[x]>` The scalar. For details of how `[x]` is encoded, see the description of `<size>`.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = Elem[D[m],index,esize];
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

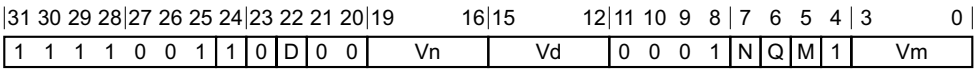
## F7.1.77 VEOR

Vector Bitwise Exclusive OR performs a bitwise Exclusive OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

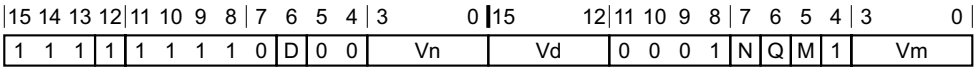
Applies when `Q = 1`.

```
VEOR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VEOR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. An A32 VEOR instruction must be unconditional. ARM strongly recommends that a T32 VEOR instruction is unconditional, see *Conditional execution* on page F2-2447. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <dt> | An optional data type. It is ignored by assemblers, and does not affect the encoding. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] EOR D[m+r];
```

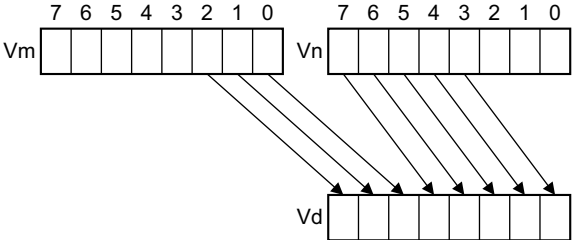

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.78 VEXT (byte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

The elements of the vectors are treated as being 8-bit fields. There is no distinction between data types.

The following figure shows an example of the operation of VEXT doubleword operation for imm = 3.

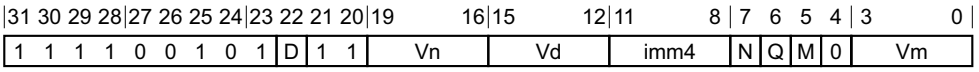



Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

This instruction is used by the alias VEXT (multibyte elements). See the *Alias conditions* on page F7-3377 table for details of when each alias is preferred.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

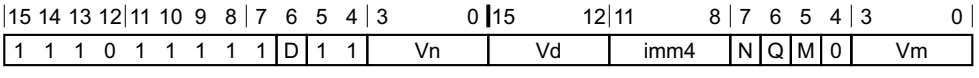
Applies when `Q = 1`.

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1');  position = 8 * UInt(imm4);
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm>
```

#### 128-bit SIMD vector variant

Applies when `Q = 1`.

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm>
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1');  position = 8 * UInt(imm4);
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Alias conditions

| Alias | is preferred when |
|---|---|
| VEXT (multibyte elements) | Never |

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VEXT instruction must be unconditional. ARM strongly recommends that a T32 VEXT instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<imm> For the 64-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 7, encoded in the "imm4" field.

For the 128-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 15, encoded in the "imm4" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        Q[d>>1] = (Q[m>>1]:Q[n>>1])<position+127:position>;
    else
        D[d] = (D[m]:D[n])<position+63:position>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

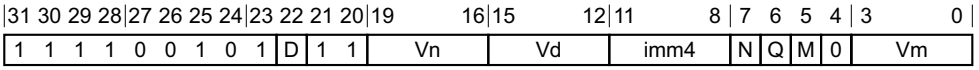
## F7.1.79 VEXT (multibyte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector

This instruction is an alias of the VEXT (byte elements) instruction. This means that:

- The encodings in this description are named to match the encodings of VEXT (byte elements).
- The description of VEXT (byte elements) gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VEXT{<c>}{<q>}.<size> {<Dd>,} <Dn>, <Dm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm*(size/8)>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

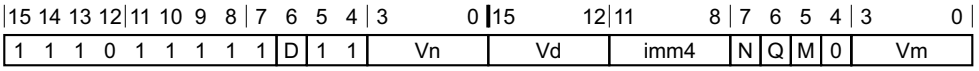
```
VEXT{<c>}{<q>}.<size> {<Qd>,} <Qn>, <Qm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm*(size/8)>
```

and is never the preferred disassembly.

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VEXT{<c>}{<q>}.<size> {<Dd>,} <Dn>, <Dm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm*(size/8)>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VEXT{<c>}{<q>}.<size> {<Qd>,} <Qn>, <Qm>, #<imm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm*(size/8)>
```

and is never the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. An A32 `VEXT` instruction must be unconditional. ARM strongly recommends that a T32 `VEXT` instruction is unconditional, see *Conditional execution* on page F2-2447. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<size>` | For the 64-bit SIMD vector variant: is the size of the operation, and can be one of 16 or 32.<br>For the 128-bit SIMD vector variant: is the size of the operation, and can be one of 16, 32 or 64. |
| `<Qd>` | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| `<Qn>` | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| `<Qm>` | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| `<Dn>` | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| `<Dm>` | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |
| `<imm>` | For the 64-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (128/`<size>`)-1.<br>For the 128-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (64/`<size>`)-1. |

### Operation for all encodings

The description of VEXT (byte elements) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

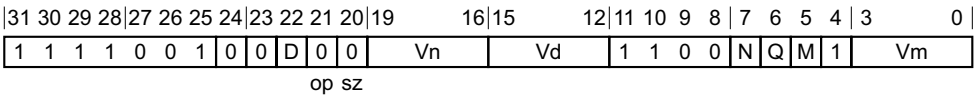
## F7.1.80 VFMA

Vector Fused Multiply Accumulate multiplies corresponding elements of two vectors, and accumulates the results into the elements of the destination vector. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VFMA{<c>}{<q>}.F32 <Dd>, <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

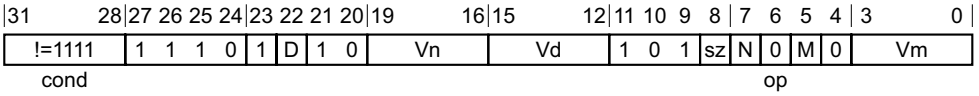
Applies when `Q = 1`.

```
VFMA{<c>}{<q>}.F32 <Qd>, <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1'); esize = 32; elements = 2;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
```

### A2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

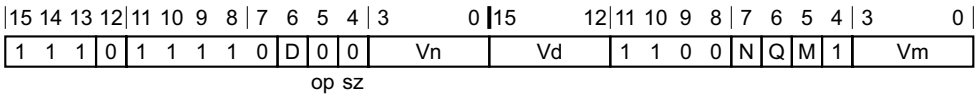
```
VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; dp_operation = (sz == '1'); op1_neg = (op == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## T1




### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VFMA{<c>}{<q>}.F32 <Dd>, <Dn>, <Dm>
```

### 128-bit SIMD vector variant

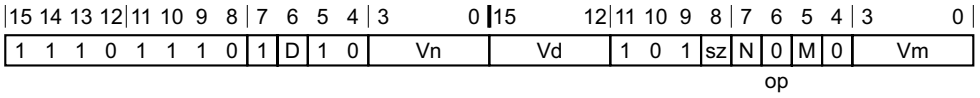
Applies when `Q = 1`.

```
VFMA{<c>}{<q>}.F32 <Qd>, <Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1'); esize = 32; elements = 2;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
```

## T2




### Single-precision scalar variant

Applies when `sz = 0`.

```
VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

### Double-precision scalar variant

Applies when `sz = 1`.

```
VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; dp_operation = (sz == '1'); op1_neg = (op == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> For encoding A1, A2 and T1: see *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VFMA or VMFS instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VFMA or VMFS instruction is unconditional, see *Conditional execution* on page F2-2447.

For encoding T2: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                bits(esize) op1 = Elem[D[n+r],e,esize];
                if op1_neg then op1 = FPNeg(op1);
                Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                                                op1, Elem[D[m+r],e,esize], StandardFPSCRValue());

    else // VFP instruction
        if dp_operation then
            op64 = if op1_neg then FPNeg(D[n]) else D[n];
            D[d] = FPMulAdd(D[d], op64, D[m], FPSCR);
        else
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
            S[d] = FPMulAdd(S[d], op32, S[m], FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

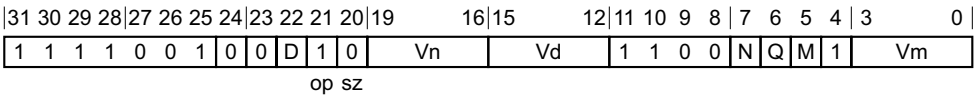
## F7.1.81 VFMS

Vector Fused Multiply Subtract negates the elements of one vector and multiplies them with the corresponding elements of another vector, adds the products to the corresponding elements of the destination vector, and places the results in the destination vector. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VFMS{<c>}{<q>}.F32 <Dd>, <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

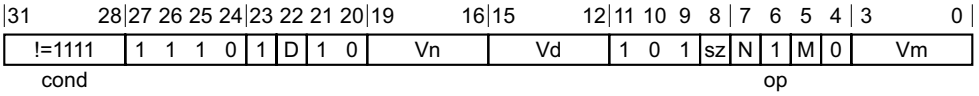
Applies when `Q = 1`.

```
VFMS{<c>}{<q>}.F32 <Qd>, <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1'); esize = 32; elements = 2;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
```

### A2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

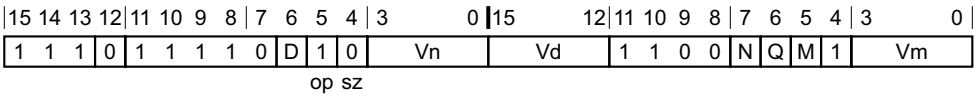
```
VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; dp_operation = (sz == '1'); op1_neg = (op == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## T1




### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VFMS{<c>}{<q>}.F32 <Dd>, <Dn>, <Dm>
```

### 128-bit SIMD vector variant

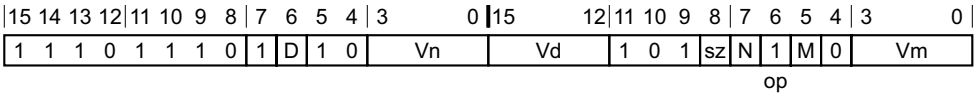
Applies when `Q = 1`.

```
VFMS{<c>}{<q>}.F32 <Qd>, <Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1'); esize = 32; elements = 2;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
```

## T2




### Single-precision scalar variant

Applies when `sz = 0`.

```
VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

### Double-precision scalar variant

Applies when `sz = 1`.

```
VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; dp_operation = (sz == '1'); op1_neg = (op == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> For encoding A1, A2 and T1: see *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VFMA or VMFS instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VFMA or VMFS instruction is unconditional, see *Conditional execution* on page F2-2447.

For encoding T2: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                bits(esize) op1 = Elem[D[n+r],e,esize];
                if op1_neg then op1 = FPNeg(op1);
                Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                                                op1, Elem[D[m+r],e,esize], StandardFPSCRValue());

    else // VFP instruction
        if dp_operation then
            op64 = if op1_neg then FPNeg(D[n]) else D[n];
            D[d] = FPMulAdd(D[d], op64, D[m], FPSCR);
        else
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
            S[d] = FPMulAdd(S[d], op32, S[m], FPSCR);
```

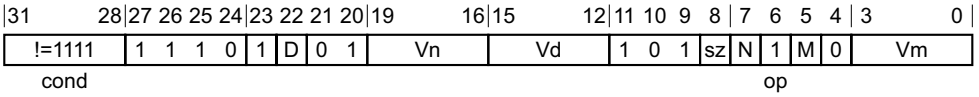

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.82 VFNMA

Vector Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

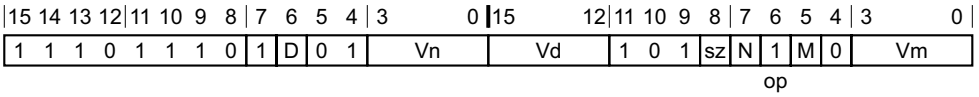
Applies when `sz = 1`.

```
VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
op1_neg = (op == '1');
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
op1_neg = (op == '1');
dp_operation = (sz == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    if dp_operation then
        op64 = if op1_neg then FPNeg(D[n]) else D[n];
        D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR);
    else
        op32 = if op1_neg then FPNeg(S[n]) else S[n];
        S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR);
```

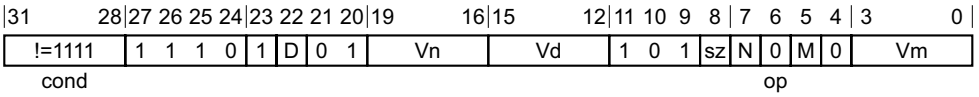

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.83 VFNMS

Vector Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

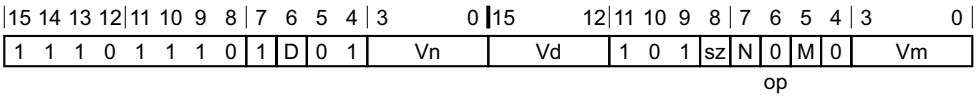
Applies when `sz = 1`.

```
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
op1_neg = (op == '1');
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
op1_neg = (op == '1');
dp_operation = (sz == '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    if dp_operation then
        op64 = if op1_neg then FPNeg(D[n]) else D[n];
        D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR);
    else
        op32 = if op1_neg then FPNeg(S[n]) else S[n];
        S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.84 VHADD

Vector Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. For rounded results, see VRHADD).

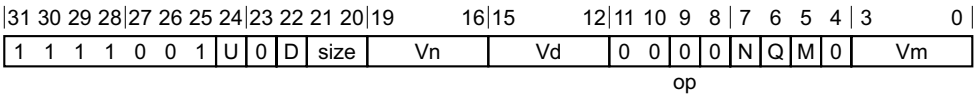
The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

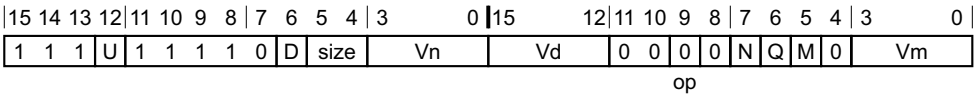
Applies when `Q = 1`.

```
VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VHADD or VHSUB instruction must be unconditional. ARM strongly recommends that a T32 VHADD or VHSUB instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.85 VHSUB

Vector Halving Subtract subtracts the elements of the second operand from the corresponding elements of the first operand, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. There is no rounding version.

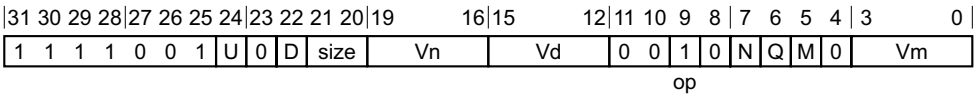
The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

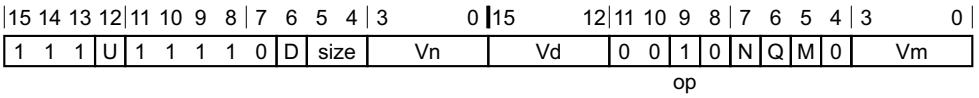
Applies when `Q = 1`.

```
VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VHADD or VHSUB instruction must be unconditional. ARM strongly recommends that a T32 VHADD or VHSUB instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

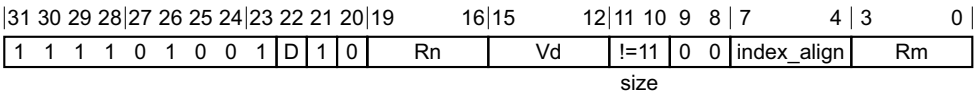
## F7.1.86 VLD1 (single element to one lane)

Load single 1-element structure to one lane of one register loads one element from memory into one element of a register. Elements of the register that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

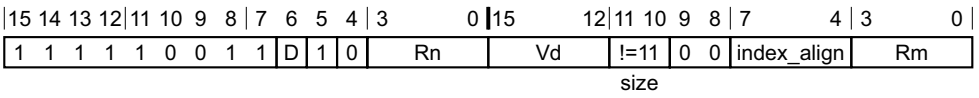
Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then SEE VLD1 (single element to all lanes);
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
    when '01'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 2;  index = UInt(index_align<3:2>);
        alignment = if index_align<0> == '0' then 1 else 2;
    when '10'
        if index_align<2> != '0' then UNDEFINED;
        if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then SEE VLD1 (single element to all lanes);
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
    when '01'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 2;  index = UInt(index_align<3:2>);
        alignment = if index_align<0> == '0' then 1 else 2;
    when '10'
        if index_align<2> != '0' then UNDEFINED;
        if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| `8` | when `size = 00` |
| `16` | when `size = 01` |
| `32` | when `size = 10` |

`<list>` Is a list containing the single 64-bit name of the SIMD&FP register holding the element. The list must be `{ <Dd>[<index>] }`. The register `<Dd>` is encoded in the "D:Vd" field. The permitted values and encoding of `<index>` depend on `<size>`:

| | |
|---|---|
| `<size> == 8` | `<index>` is in the range 0 to 7, encoded in the "index_align<3:1>" field. |
| `<size> == 16` | `<index>` is in the range 0 to 3, encoded in the "index_align<3:2>" field. |
| `<size> == 32` | `<index>` is 0 or 1, encoded in the "index_align<3>" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> When <size> == 8, <align> must be omitted, otherwise it is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and the encoding depends on <size>:

<size> == 8 Encoded in the "index_align<0>" field as 0.

<size> == 16 Encoded in the "index_align<1:0>" field as 0b00.

<size> == 32 Encoded in the "index_align<2:0>" field as 0b000.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 16 <align> is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

<size> == 32 <align> is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as 0b011.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
    Elem[D[d],index] = MemU[address,ebytes];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

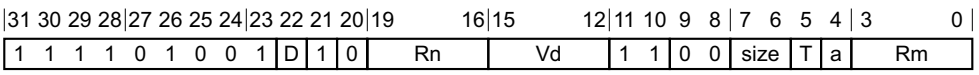
## F7.1.87 VLD1 (single element to all lanes)

Load single 1-element structure and replicate to all lanes of one register loads one element from memory into every element of one or two vectors. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

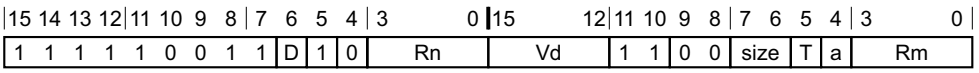
Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size);  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

### T1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size);  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD1 (single element to all lanes)* on page K1-5486.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

- `{ <Dd>[] }` Encoded in the "T" field as `0`.
- `{ <Dd>[], <Dd+1>[] }` Encoded in the "T" field as `1`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` When `<size>` == 8, `<align>` must be omitted, otherwise it is the optional alignment. Whenever `<align>` is omitted, the standard alignment is used, see , and is encoded in the "a" field as `0`. Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

- `<size> == 16` `<align>` is 16, meaning 16-bit alignment, encoded in the "a" field as `1`.
- `<size> == 32` `<align>` is 32, meaning 32-bit alignment, encoded in the "a" field as `1`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *Advanced SIMD addressing mode* on page F5-2571.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
    bits(64) replicated_element = Replicate(MemU[address,ebytes]);
    for r = 0 to regs-1
        D[d+r] = replicated_element;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

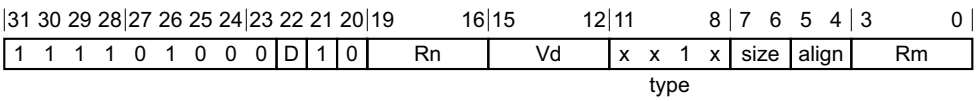
## F7.1.88 VLD1 (multiple single elements)

Load multiple single 1-element structures to one, two, three, or four registers loads elements from memory into one, two, three, or four registers, without de-interleaving. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

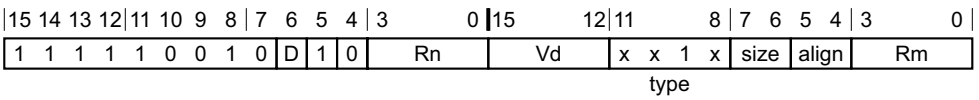
Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
case type of
    when '0111'
        regs = 1;  if align<1> == '1' then UNDEFINED;
    when '1010'
        regs = 2;  if align == '11' then UNDEFINED;
    when '0110'
        regs = 3;  if align<1> == '1' then UNDEFINED;
    when '0010'
        regs = 4;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
case type of
    when '0111'
        regs = 1;  if align<1> == '1' then UNDEFINED;
    when '1010'
        regs = 2;  if align == '11' then UNDEFINED;
    when '0110'
        regs = 3;  if align<1> == '1' then UNDEFINED;
    when '0010'
        regs = 4;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD1 (multiple single elements)* on page K1-5485.

Related encodings: Advanced SIMD element or structure load/store instructions.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`
- `64` when `size = 11`

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

- `{ <Dd> }` Encoded in the "type" field as `0b0111`.
- `{ <Dd>, <Dd+1> }` Encoded in the "type" field as `0b1010`.
- `{ <Dd>, <Dd+1>, <Dd+2> }` Encoded in the "type" field as `0b0110`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }Encoded in the "type" field as 0b0010.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as 0b01.

128 128-bit alignment, encoded in the "align" field as 0b10. Available only if <list> contains two or four registers.

256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 8*regs;
    for r = 0 to regs-1
        for e = 0 to elements-1
            bits(ebytes*8) data;
            if ebytes != 8 then
                data = MemU[address,ebytes];
            else
                data<31:0> = if BigEndian() then MemU[address+4,4] else MemU[address,4];
                data<63:32> = if BigEndian() then MemU[address,4] else MemU[address+4,4];
            Elem[D[d+r],e] = data;
            address = address + ebytes;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

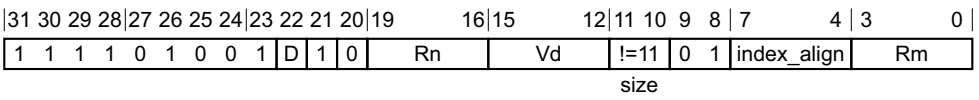
## F7.1.89 VLD2 (single 2-element structure to one lane)

Load single 2-element structure to one lane of two registers loads one 2-element structure from memory into corresponding elements of two registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

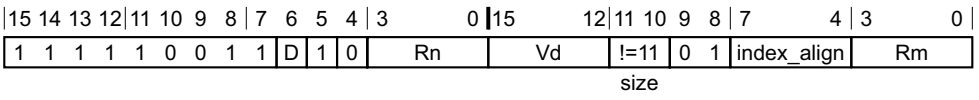
Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then SEE VLD2 (single 2-element structure to all lanes);
case size of
    when '00'
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 2;
    when '01'
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '10'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when Rm = 1111.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when Rm = 1101.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when Rm != 11x1.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then SEE VLD2 (single 2-element structure to all lanes);
case size of
    when '00'
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 2;
    when '01'
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '10'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD2 (single 2-element structure to one lane)* on page K1-5486.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> Is the data size, encoded in the "size" field. It can have the following values:

- 8 when size = 00
- 16 when size = 01
- 32 when size = 10

<list> Is a list containing the 64-bit names of the two SIMD&FP registers holding the element. The list must be one of:

- { <Dd>[<index>], <Dd+1>[<index>] } Single-spaced registers, encoded as "spacing" = 0.
- { <Dd>[<index>], <Dd+2>[<index>] } Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding of "spacing" depends on <size>:

<size> == 16 "spacing" is encoded in the "index_align<1>" field.

<size> == 32 "spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8 <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16 <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32 <index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and the encoding depends on <size>:

<size> == 8 Encoded in the "index_align<0>" field as 0.

<size> == 16 Encoded in the "index_align<0>" field as 0.

<size> == 32 Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8 <align> is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16 <align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32 <align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

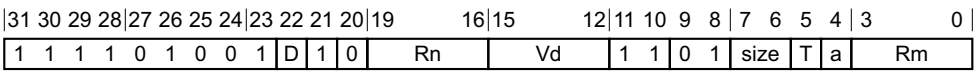
## F7.1.90 VLD2 (single 2-element structure to all lanes)

Load single 2-element structure and replicate to all lanes of two registers loads one 2-element structure from memory into all lanes of two registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

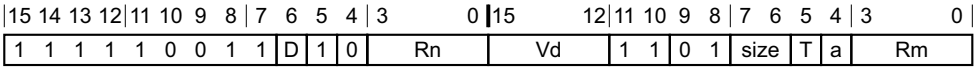
Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
ebytes = 1 << UInt(size);
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

### T1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
ebytes = 1 << UInt(size);
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD2 (single 2-element structure to all lanes)* on page K1-5487.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| `8` | when `size = 00` |
| `16` | when `size = 01` |
| `32` | when `size = 10` |

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of two SIMD&FP registers. The list must be one of:

| | |
|---|---|
| `{ <Dd>[], <Dd+1>[] }` | Single-spaced registers, encoded in the "T" field as `0`. |
| `{ <Dd>[], <Dd+2>[] }` | Double-spaced registers, encoded in the "T" field as `1`. |

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` Is the optional alignment. Whenever `<align>` is omitted, the standard alignment is used, see , and is encoded in the "a" field as `0`. Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

| | |
|---|---|
| `<size> == 8` | `<align>` is 16, meaning 16-bit alignment, encoded in the "a" field as `1`. |
| `<size> == 16` | `<align>` is 32, meaning 32-bit alignment, encoded in the "a" field as `1`. |
| `<size> == 32` | `<align>` is 64, meaning 64-bit alignment, encoded in the "a" field as `1`. |

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *Advanced SIMD addressing mode* on page F5-2571.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

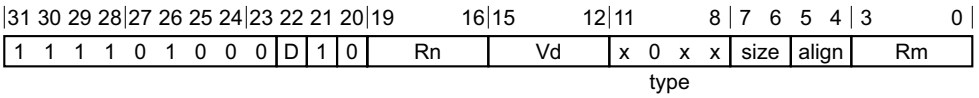
## F7.1.91 VLD2 (multiple 2-element structures)

Load multiple 2-element structures to two or four registers loads multiple 2-element structures from memory into two or four registers, with de-interleaving. For more information, see *Element and structure load/store instructions* on page F1-2428. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

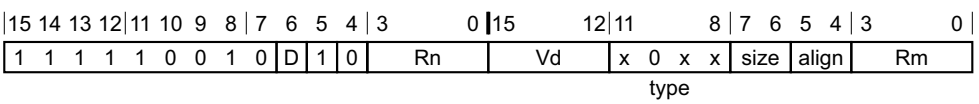
Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case type of
    when '1000'
        regs = 1;  inc = 1;  if align == '11' then UNDEFINED;
    when '1001'
        regs = 1;  inc = 2;  if align == '11' then UNDEFINED;
    when '0011'
        regs = 2;  inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case type of
    when '1000'
        regs = 1;  inc = 1;  if align == '11' then UNDEFINED;
    when '1001'
        regs = 1;  inc = 2;  if align == '11' then UNDEFINED;
    when '0011'
        regs = 2;  inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD2 (multiple 2-element structures)* on page K1-5486.

Related encodings: Advanced SIMD element or structure load/store instructions.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

- `{ <Dd>, <Dd+1> }` Single-spaced registers, encoded in the "type" field as `0b1000`.
- `{ <Dd>, <Dd+2> }` Double-spaced registers, encoded in the "type" field as `0b1001`.
- `{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Single-spaced registers, encoded in the "type" field as `0b0011`.

The register `<Dd>` is encoded in the "D:Vd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as 0b01.

128 128-bit alignment, encoded in the "align" field as 0b10.

256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 16*regs;
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r], e] = MemU[address,ebytes];
            Elem[D[d2+r],e] = MemU[address+ebytes,ebytes];
            address = address + 2*ebytes;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

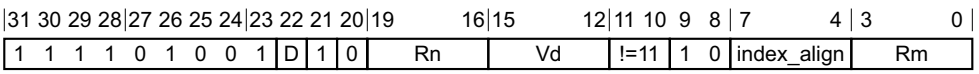
## F7.1.92 VLD3 (single 3-element structure to one lane)

Load single 3-element structure to one lane of three registers loads one 3-element structure from memory into corresponding elements of three registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

#### *Post-indexed variant*

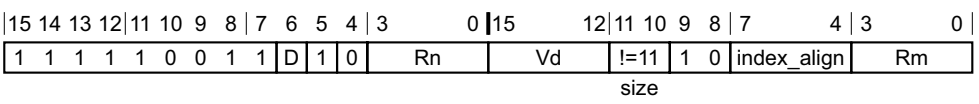
Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then SEE VLD3 (single 3-element structure to all lanes);
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
    when '01'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
    when '10'
        if index_align<1:0> != '00' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when Rm = 1111.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

#### *Post-indexed variant*

Applies when Rm = 1101.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

#### *Post-indexed variant*

Applies when Rm != 11x1.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then SEE VLD3 (single 3-element structure to all lanes);
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
    when '01'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
    when '10'
        if index_align<1:0> != '00' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD3 (single 3-element structure to one lane)* on page K1-5487.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> Is the data size, encoded in the "size" field. It can have the following values:

- 8 when size = 00
- 16 when size = 01
- 32 when size = 10

<list> Is a list containing the 64-bit names of the three SIMD&FP registers holding the element. The list must be one of:

- { <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] } Single-spaced registers, encoded as "spacing" = 0.
- { <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] } Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding of "spacing" depends on <size>:

<size> == 8 — "spacing" is encoded in the "index_align<0>" field.

<size> == 16 — "spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.

<size> == 32 — "spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8 — <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16 — <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32 — <index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Alignment

Standard alignment rules apply, see *Alignment support* on page B2-75.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index] = MemU[address+2*ebytes,ebytes];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

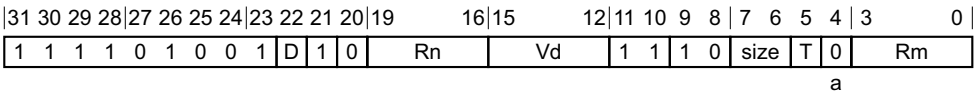
## F7.1.93 VLD3 (single 3-element structure to all lanes)

Load single 3-element structure and replicate to all lanes of three registers loads one 3-element structure from memory into all lanes of three registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

#### *Post-indexed variant*

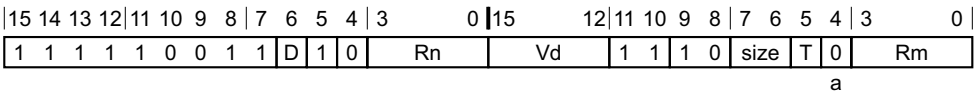
Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### T1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD3 (single 3-element structure to all lanes)* on page K1-5488.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of three SIMD&FP registers. The list must be one of:

`{ <Dd>[], <Dd+1>[], <Dd+2>[] }` Single-spaced registers, encoded in the "T" field as `0`.

`{ <Dd>[], <Dd+2>[], <Dd+4>[] }` Double-spaced registers, encoded in the "T" field as `1`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

### Alignment

Standard alignment rules apply, see *Alignment support* on page B2-75.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        R[n] = R[n] + 3*ebytes;
D[d] = Replicate(MemU[address,ebytes]);
D[d2] = Replicate(MemU[address+ebytes,ebytes]);
D[d3] = Replicate(MemU[address+2*ebytes,ebytes]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

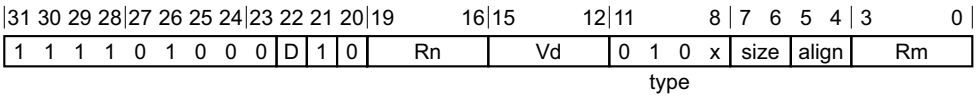
## F7.1.94 VLD3 (multiple 3-element structures)

Load multiple 3-element structures to three registers loads multiple 3-element structures from memory into three registers, with de-interleaving. For more information, see *Element and structure load/store instructions* on page F1-2428. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

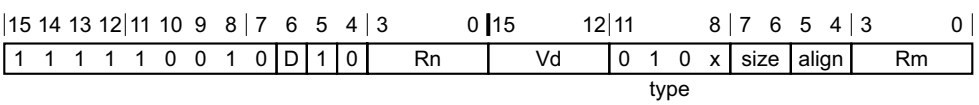
Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || align<1> == '1' then UNDEFINED;
case type of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || align<1> == '1' then UNDEFINED;
case type of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD3 (multiple 3-element structures)* on page K1-5487.

Related encodings: Advanced SIMD element or structure load/store instructions.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

- `{ <Dd>, <Dd+1>, <Dd+2> }` Single-spaced registers, encoded in the "type" field as `0b0100`.
- `{ <Dd>, <Dd+2>, <Dd+4> }` Double-spaced registers, encoded in the "type" field as `0b0101`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<align>` Is the optional alignment. Whenever `<align>` is omitted, the standard alignment is used, see , and is encoded in the "align" field as `0b00`. Whenever `<align>` is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as `0b01`. `:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *Advanced SIMD addressing mode* on page F5-2571.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
    for e = 0 to elements-1
        Elem[D[d], e] = MemU[address,ebytes];
        Elem[D[d2],e] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e] = MemU[address+2*ebytes,ebytes];
        address = address + 3*ebytes;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

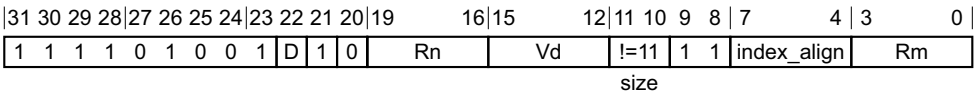
## F7.1.95 VLD4 (single 4-element structure to one lane)

Load single 4-element structure to one lane of four registers loads one 4-element structure from memory into corresponding elements of four registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

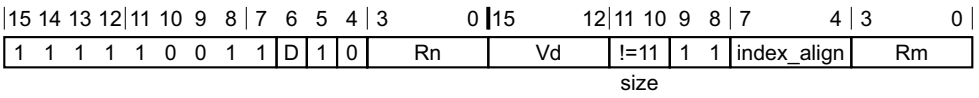
Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then SEE VLD4 (single 4-element structure to all lanes);
case size of
    when '00'
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '01'
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
    when '10'
        if index_align<1:0> == '11' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### Offset variant

Applies when Rm = 1111.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### Post-indexed variant

Applies when Rm = 1101.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### Post-indexed variant

Applies when Rm != 11x1.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### Decode for all variants of this encoding

```
if size == '11' then SEE VLD4 (single 4-element structure to all lanes);
case size of
    when '00'
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '01'
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
    when '10'
        if index_align<1:0> == '11' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD4 (single 4-element structure to one lane)* on page K1-5488.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> Is the data size, encoded in the "size" field. It can have the following values:

- 8 when size = 00
- 16 when size = 01
- 32 when size = 10

<list> Is a list containing the 64-bit names of the four SIMD&FP registers holding the element. The list must be one of:

- { <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] } Single-spaced registers, encoded as "spacing" = 0.
- { <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] } Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding of "spacing" depends on <size>:

<size> == 16 "spacing" is encoded in the "index_align<1>" field.

<size> == 32 "spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8 <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16 <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32 <index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and the encoding depends on <size>:

<size> == 8 Encoded in the "index_align<0>" field as 0.

<size> == 16 Encoded in the "index_align<0>" field as 0.

<size> == 32 Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8 <align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16 <align> is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32 <align> can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as 0b01, and 128-bit alignment is encoded in the "index_align<1:0>" field as 0b10.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index] = MemU[address+2*ebytes,ebytes];
    Elem[D[d4],index] = MemU[address+3*ebytes,ebytes];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

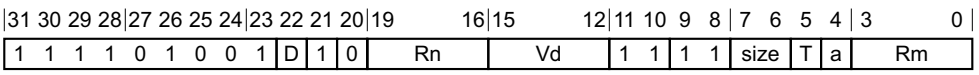
## F7.1.96 VLD4 (single 4-element structure to all lanes)

Load single 4-element structure and replicate to all lanes of four registers loads one 4-element structure from memory into all lanes of four registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

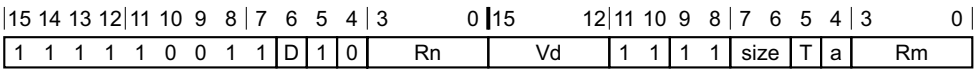
Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' && a == '0' then UNDEFINED;
if size == '11' then
    ebytes = 4;  alignment = 16;
else
    ebytes = 1 << UInt(size);
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' && a == '0' then UNDEFINED;
if size == '11' then
    ebytes = 4;  alignment = 16;
else
    ebytes = 1 << UInt(size);
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD4 (single 4-element structure to all lanes)* on page K1-5489.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| `8` | when `size = 00` |
| `16` | when `size = 01` |
| `32` | when `size = 1x` |

`<list>` Is a list containing the 64-bit names of four SIMD&FP registers. The list must be one of:

`{ <Dd>[], <Dd+1>[], <Dd+2>[], <Dd+3>[] }` Single-spaced registers, encoded in the "T" field as `0`.

`{ <Dd>[], <Dd+2>[], <Dd+4>[], <Dd+6>[] }` Double-spaced registers, encoded in the "T" field as `1`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and is encoded in the "a" field as 0. Whenever <align> is present, the permitted values and encoding depend on <size>:

| | |
|---|---|
| <size> == 8 | <align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1. |
| <size> == 16 | <align> is 64, meaning 64-bit alignment, encoded in the "a" field as 1. |
| <size> == 32 | <align> can be 64 or 128. 64-bit alignment is encoded in the "a:size<0>" field as 0b10, and 128-bit alignment is encoded in the "a:size<0>" field as 0b11. |

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes]);
    D[d4] = Replicate(MemU[address+3*ebytes,ebytes]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

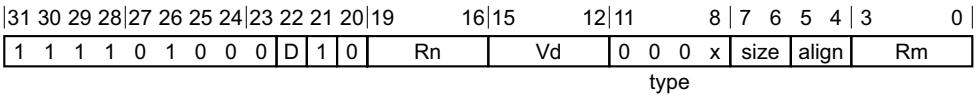
## F7.1.97 VLD4 (multiple 4-element structures)

Load multiple 4-element structures to four registers loads multiple 4-element structures from memory into four registers, with de-interleaving. For more information, see *Element and structure load/store instructions* on page F1-2428. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

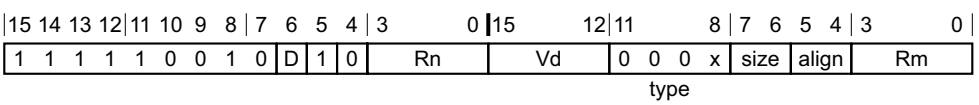
Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case type of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case type of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLD4 (multiple 4-element structures)* on page K1-5488.

Related encodings: Advanced SIMD element or structure load/store instructions.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Single-spaced registers, encoded in the "type" field as `0b0000`.

`{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> }` Double-spaced registers, encoded in the "type" field as `0b0001`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as 0b01.

128 128-bit alignment, encoded in the "align" field as 0b10.

256 256-bit alignment, encoded in the "align" field as 0b11.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 32;
    for e = 0 to elements-1
        Elem[D[d], e] = MemU[address,ebytes];
        Elem[D[d2],e] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e] = MemU[address+2*ebytes,ebytes];
        Elem[D[d4],e] = MemU[address+3*ebytes,ebytes];
        address = address + 4*ebytes;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

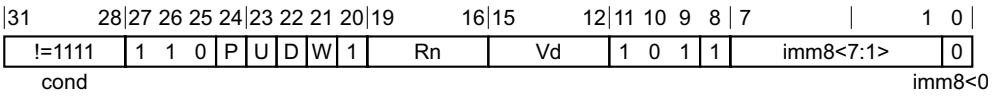
## F7.1.98 VLDM, VLDMDB, VLDMIA

Load Multiple SIMD&FP registers loads multiple registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

This instruction is used by the alias VPOP. See the *Alias conditions* on page F7-3432 table for details of when each alias is preferred.

### A1




#### *Decrement Before variant*

Applies when `P = 1 && U = 0 && W = 1`.

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

#### *Increment After variant*

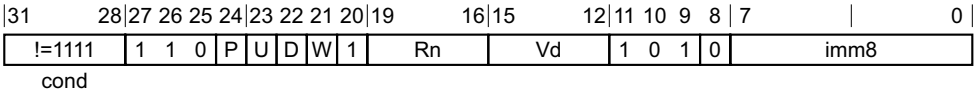
Applies when `P = 0 && U = 1`.

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

#### *Decode for all variants of this encoding*

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VLDR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

### A2




#### *Decrement Before variant*

Applies when `P = 1 && U = 0 && W = 1`.

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

#### *Increment After variant*

Applies when `P = 0 && U = 1`.

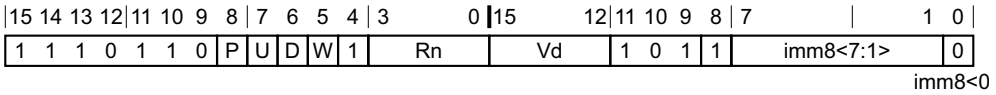
```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Decode for all variants of this encoding*

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VLDR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

### T1




#### *Decrement Before variant*

Applies when `P = 1 && U = 0 && W = 1`.

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

#### *Increment After variant*

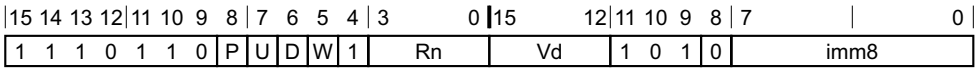
Applies when `P = 0 && U = 1`.

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

#### *Decode for all variants of this encoding*

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VLDR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

### T2




#### *Decrement Before variant*

Applies when `P = 1 && U = 0 && W = 1`.

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

#### *Increment After variant*

Applies when `P = 0 && U = 1`.

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VLDR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VLDM* on page K1-5489.

Related encodings: 64-bit transfers between ARM core and extension registers.

## Alias conditions

| Alias | is preferred when |
|---|---|
| VPOP | `P == '0' && U == '1' && W == '1' && Rn == '1101'` |

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<Rn> For encoding A1 and A2: is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

! Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.

<sreglist> Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

<dreglist> Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];  address = address+4;
        else
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian() then word1:word2 else word2:word1;
```

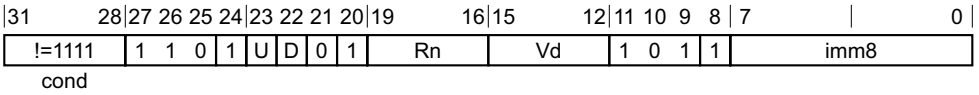

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.99 VLDR

Load SIMD&FP register loads a single register from the Advanced SIMD and floating-point register file, using an address from a general-purpose register, with an optional offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Literal variant*

Applies when `Rn = 1111`.

```
VLDR{<c>}{<q>}{.64} <Dd>, <label>
VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]
```

#### *Offset variant*

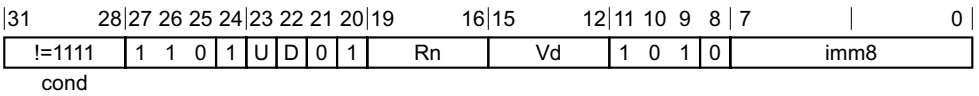
Applies when `Rn != 1111`.

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm>}]
```

#### *Decode for all variants of this encoding*

```
single_reg = FALSE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(D:Vd);  n = UInt(Rn);
```

### A2




#### *Literal variant*

Applies when `Rn = 1111`.

```
VLDR{<c>}{<q>}{.32} <Sd>, <label>
VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]
```

#### *Offset variant*

Applies when `Rn != 1111`.

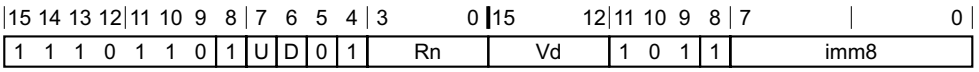
```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm>}]
```

#### *Decode for all variants of this encoding*

```
single_reg = TRUE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(Vd:D);  n = UInt(Rn);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *Literal variant*

Applies when `Rn = 1111`.

```
VLDR{<c>}{<q>}{.64} <Dd>, <label>
VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]
```

#### *Offset variant*

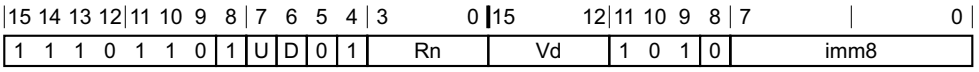
Applies when `Rn != 1111`.

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm>}]
```

#### *Decode for all variants of this encoding*

```
single_reg = FALSE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(D:Vd);  n = UInt(Rn);
```

### T2




#### *Literal variant*

Applies when `Rn = 1111`.

```
VLDR{<c>}{<q>}{.32} <Sd>, <label>
VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]
```

#### *Offset variant*

Applies when `Rn != 1111`.

```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm>}]
```

#### *Decode for all variants of this encoding*

```
single_reg = TRUE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(Vd:D);  n = UInt(Rn);
```

## Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `.64` | Optional data size specifiers. |
| `<Dd>` | The destination register for a doubleword load. |
| `.32` | Optional data size specifiers. |
| `<Sd>` | The destination register for a singleword load. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<label>` The label of the literal data item to be loaded. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`. If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- `-` when `U = 0`
- `+` when `U = 1`

`<imm>` The immediate offset used for forming the address. For the immediate forms of the syntax, `<imm>` can be omitted, in which case the #0 form of the instruction is assembled. Permitted values are multiples of 4 in the range 0 to 1020.

For the literal forms of the instruction, the base register is encoded as `0b1111` to indicate that the PC is the base register.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page F1-2409.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    if single_reg then
        S[d] = MemA[address,4];
    else
        word1 = MemA[address,4];  word2 = MemA[address+4,4];
        // Combine the word-aligned words in the correct order for current endianness.
        D[d] = if BigEndian() then word1:word2 else word2:word1;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.100 VMAX (floating-point)

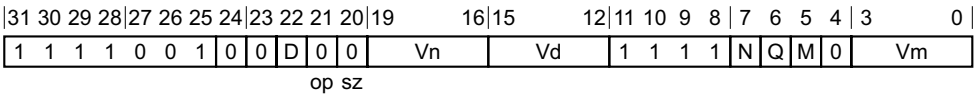
Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements are 32-bit floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMAX{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

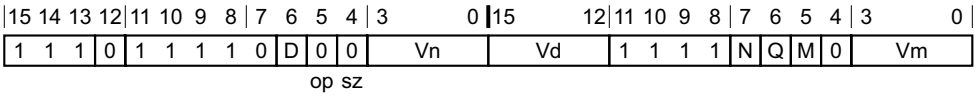
Applies when `Q = 1`.

```
VMAX{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMAX{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMAX{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VMAX or VMIN instruction must be unconditional. ARM strongly recommends that a T32 VMAX or VMIN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Floating-point maximum and minimum

- max(+0.0, -0.0) = +0.0
- If any input is a NaN, the corresponding result element is the default NaN.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRValue());
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.101 VMAX (integer)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

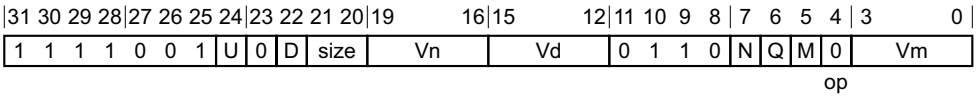
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

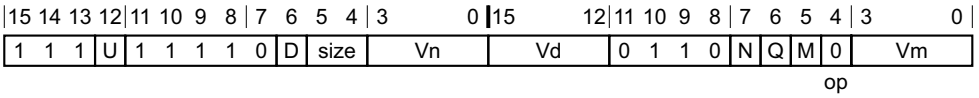
Applies when `Q = 1`.

```
VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VMAX or VMIN instruction must be unconditional. ARM strongly recommends that a T32 VMAX or VMIN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

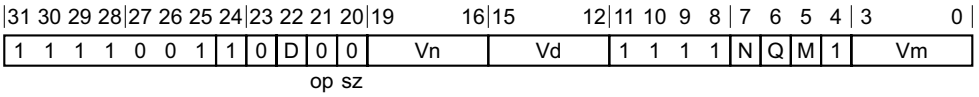
## F7.1.102 VMAXNM

This instruction determines the floating-point maximum number.

It handles NaNs in consistence with the IEEE754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point VMAX.

This instruction is not conditional.

### A1




#### *64-bit SIMD vector variant*

Applies when Q = 0.

```
VMAXNM{<q>}.F32 <Dd>, <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

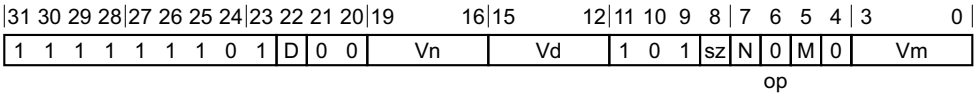
Applies when Q = 1.

```
VMAXNM{<q>}.F32 <Qd>, <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE; esize = 32; elements = 2;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### A2




#### *Single-precision scalar variant*

Applies when sz = 0.

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when sz = 1.

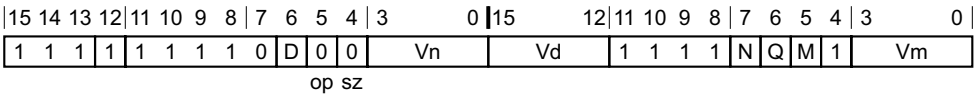
```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
advsimd = FALSE;
maximum = (op == '0'); dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMAXNM{<q>}.F32 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

#### *128-bit SIMD vector variant*

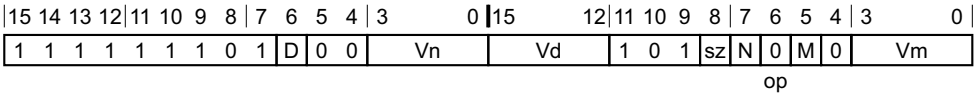
Applies when `Q = 1`.

```
VMAXNM{<q>}.F32 <Qd>, <Qn>, <Qm> // Not permitted in IT block
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE; esize = 32; elements = 2;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

#### *Decode for all variants of this encoding*

```
advsimd = FALSE;
maximum = (op == '0'); dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then             // Advanced SIMD instruction
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
            if maximum then
                Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
else                        // VFP instruction
    if dp_operation then
        if maximum then
            D[d] = FPMaxNum(D[n], D[m], FPSCR);
        else
            D[d] = FPMinNum(D[n], D[m], FPSCR);
    else
        if maximum then
            S[d] = FPMaxNum(S[n], S[m], FPSCR);
        else
            S[d] = FPMinNum(S[n], S[m], FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.103 VMIN (floating-point)

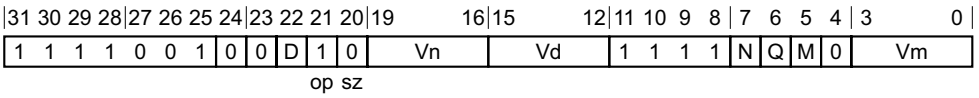
Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements are 32-bit floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMIN{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

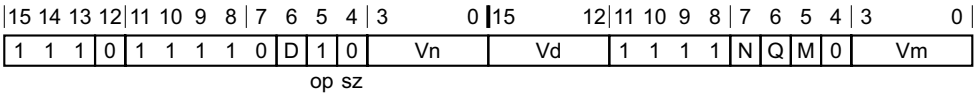
Applies when `Q = 1`.

```
VMIN{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMIN{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMIN{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VMAX or VMIN instruction must be unconditional. ARM strongly recommends that a T32 VMAX or VMIN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Floating-point minimum

- min(+0.0, -0.0) = -0.0
- If any input is a NaN, the corresponding result element is the default NaN.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRValue());
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.104 VMIN (integer)

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

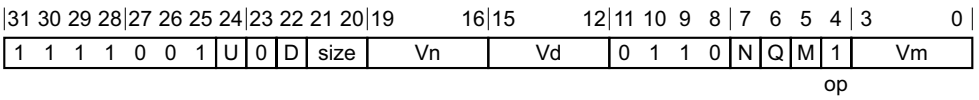
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

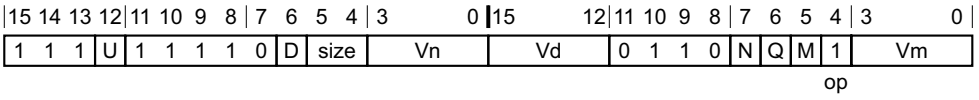
Applies when `Q = 1`.

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VMAX or VMIN instruction must be unconditional. ARM strongly recommends that a T32 VMAX or VMIN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

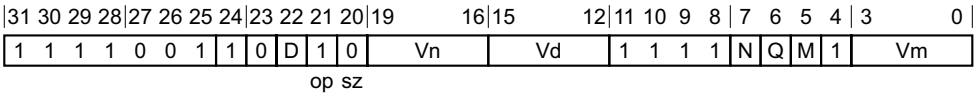
## F7.1.105 VMINNM

This instruction determines the floating point minimum number.

It handles NaNs in consistence with the IEEE754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point `VMIN`.

This instruction is not conditional.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMINNM{<q>}.F32 <Dd>, <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

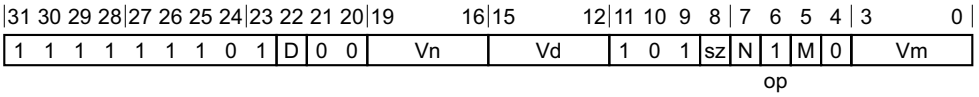
Applies when `Q = 1`.

```
VMINNM{<q>}.F32 <Qd>, <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE; esize = 32; elements = 2;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### A2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

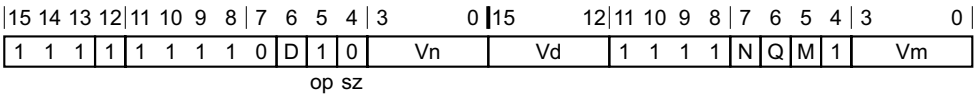
```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
advsimd = FALSE;
maximum = (op == '0'); dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VMINNM{<q>}.F32 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

#### 128-bit SIMD vector variant

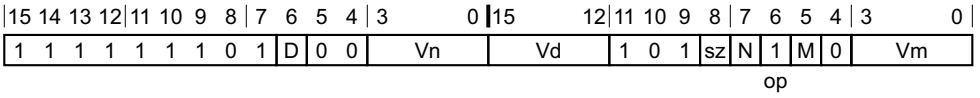
Applies when `Q = 1`.

```
VMINNM{<q>}.F32 <Qd>, <Qn>, <Qm> // Not permitted in IT block
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE; esize = 32; elements = 2;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T2




#### Single-precision scalar variant

Applies when `sz = 0`.

```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

#### Double-precision scalar variant

Applies when `sz = 1`.

```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

#### Decode for all variants of this encoding

```
advsimd = FALSE;
maximum = (op == '0'); dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then             // Advanced SIMD instruction
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
            if maximum then
                Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
else                        // VFP instruction
    if dp_operation then
        if maximum then
            D[d] = FPMaxNum(D[n], D[m], FPSCR);
        else
            D[d] = FPMinNum(D[n], D[m], FPSCR);
    else
        if maximum then
            S[d] = FPMaxNum(S[n], S[m], FPSCR);
        else
            S[d] = FPMinNum(S[n], S[m], FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

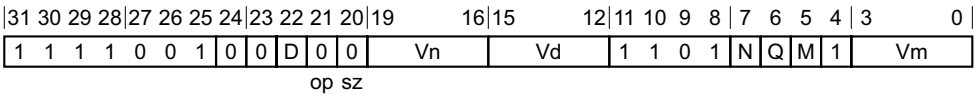
## F7.1.106 VMLA (floating-point)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VMLA{<c>}{<q>}.F32 <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0, sz = 0
```

#### 128-bit SIMD vector variant

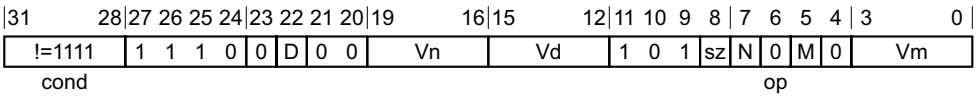
Applies when `Q = 1`.

```
VMLA{<c>}{<q>}.F32 <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1, sz = 0
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  add = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### A2




#### Single-precision scalar variant

Applies when `sz = 0`.

```
VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### Double-precision scalar variant

Applies when `sz = 1`.

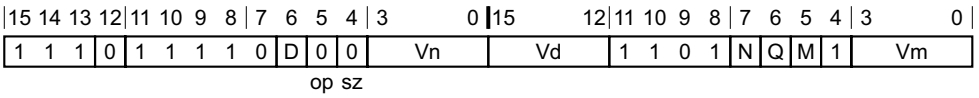
```
VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');  add = (op == '0');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VMLA{<c>}{<q>}.F32 <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0, sz = 0
```

#### 128-bit SIMD vector variant

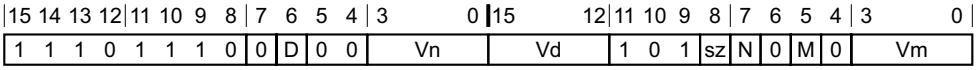
Applies when `Q = 1`.

```
VMLA{<c>}{<q>}.F32 <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1, sz = 0
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  add = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T2




#### Single-precision scalar variant

Applies when `sz = 0`.

```
VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### Double-precision scalar variant

Applies when `sz = 1`.

```
VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');  add = (op == '0');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## Assembler symbols

<c> For encoding A1, A2 and T1: see *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VMLA` or `VMLS` instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VMLA` or `VMLS` instruction is unconditional, see *Conditional execution* on page F2-2447.

For encoding T2: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, StandardFPSCRValue());
    else             // VFP instruction
        if dp_operation then
            addend64 = if add then FPMul(D[n], D[m], FPSCR) else FPNeg(FPMul(D[n], D[m], FPSCR));
            D[d] = FPAdd(D[d], addend64, FPSCR);
        else
            addend32 = if add then FPMul(S[n], S[m], FPSCR) else FPNeg(FPMul(S[n], S[m], FPSCR));
            S[d] = FPAdd(S[d], addend32, FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

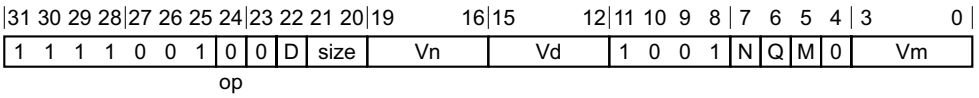
## F7.1.107 VMLA (integer)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and adds the products to the corresponding elements of the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMLA{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0
```

#### *128-bit SIMD vector variant*

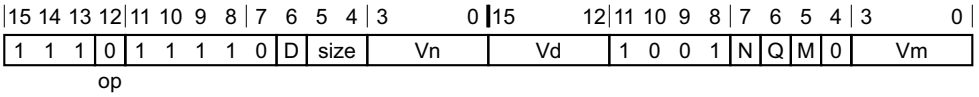
Applies when `Q = 1`.

```
VMLA{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMLA{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMLA{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> The data type for the elements of the operands. It must be one of:

- S — Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.
- U — Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.
- I — Available only in encoding T1/A1.

<size> The data size for the elements of the operands. It must be one of:

- 8 — Encoded as size = 0b00.
- 16 — Encoded as size = 0b01.
- 32 — Encoded as size = 0b10.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din

[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.108 VMLA (by scalar)

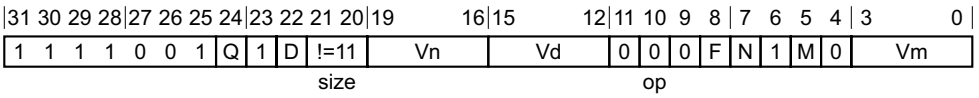
Vector Multiply Accumulate multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector.

For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMLA{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm[x]> // Encoding T1/A1, encoded as Q = 0
```

#### *128-bit SIMD vector variant*

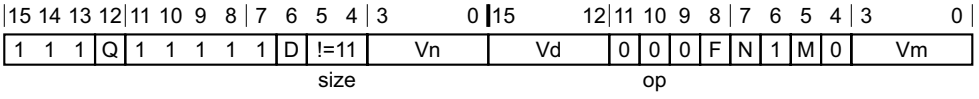
Applies when `Q = 1`.

```
VMLA{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Dm[x]> // Encoding T1/A1, encoded as Q = 1
```

#### *Decode for all variants of this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMLA{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm[x]> // Encoding T1/A1, encoded as Q = 0
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMLA{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Dm[x]> // Encoding T1/A1, encoded as Q = 1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

## Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VMLA`, `VMLAL`, `VMLS`, or `VMLSL` instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VMLA`, `VMLAL`, `VMLS`, or `VMLSL` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<type>` The data type for the elements of the operands. It must be one of:

- `S` Encoding T2/A2, encoded as U = 0.
- `U` Encoding T2/A2, encoded as U = 1.
- `I` Encoding T1/A1, encoded as F = 0.
- `F` Encoding T1/A1, encoded as F = 1. `<size>` must be 32.

`<size>` The operand element data size. It can be:

- `16` Encoded as size = 01.
- `32` Encoded as size = 10.

`<Qd>` Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` The scalar. `Dm` is restricted to D0-D7 if `<size>` is 16, or D0-D15 otherwise.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if long_destination then
    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
else
    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

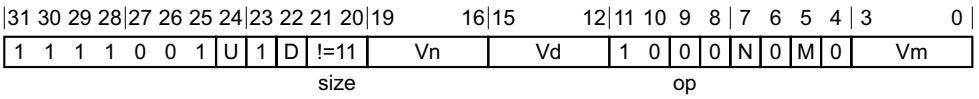
## F7.1.109 VMLAL (integer)

Vector Multiply Accumulate Long multiplies corresponding elements in two vectors, and add the products to the corresponding element of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



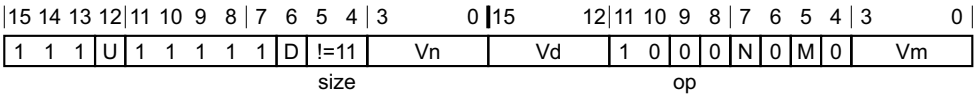

#### *A1 variant*

```
VMLAL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // Encoding T2/A2
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

### T1




#### *T1 variant*

```
VMLAL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // Encoding T2/A2
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction is unconditional, see *Conditional execution* on page F2-2447.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> The data type for the elements of the operands. It must be one of:

S Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.

U Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.

I Available only in encoding T1/A1.

<size> The data size for the elements of the operands. It must be one of:

8 Encoded as size = 0b00.

16 Encoded as size = 0b01.

32 Encoded as size = 0b10.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.110 VMLAL (by scalar)

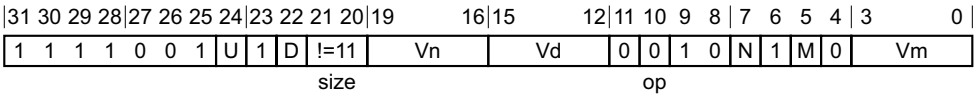
Vector Multiply Accumulate Long multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



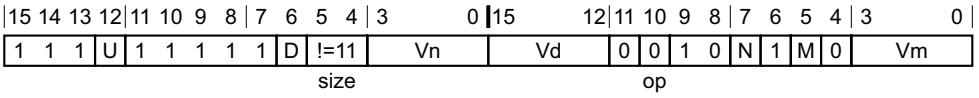

#### *A1 variant*

```
VMLAL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm[x]> // Encoding T2/A2
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

### T1




#### *T1 variant*

```
VMLAL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm[x]> // Encoding T2/A2
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> The data type for the elements of the operands. It must be one of:

- S — Encoding T2/A2, encoded as U = 0.
- U — Encoding T2/A2, encoded as U = 1.
- I — Encoding T1/A1, encoded as F = 0.
- F — Encoding T1/A1, encoded as F = 1. <size> must be 32.

<size> The operand element data size. It can be:

- 16 — Encoded as size = 01.
- 32 — Encoded as size = 10.

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> The scalar. Dm is restricted to D0-D7 if <size> is 16, or D0-D15 otherwise.

### Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDEnabled();
     op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
     for r = 0 to regs-1
         for e = 0 to elements-1
             op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
             if floating_point then
                 fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                 Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
             else
                 addend = if add then op1val*op2val else -op1val*op2val;
                 if long_destination then
                     Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                 else
                     Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F7.1.111 VMLS (floating-point)

Vector Multiply Subtract multiplies corresponding elements in two vectors, subtracts the products from corresponding elements of the destination vector, and places the results in the destination vector.

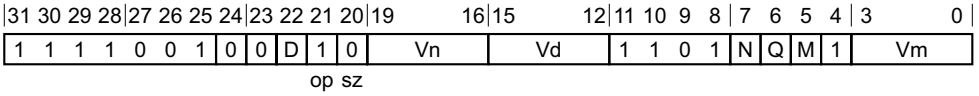
**Note**

ARM recommends that software does not use the VMLS instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

#### A1




##### *64-bit SIMD vector variant*

Applies when Q = 0.

```
VMLS{<c>}{<q>}.F32 <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0, sz = 0
```

##### *128-bit SIMD vector variant*

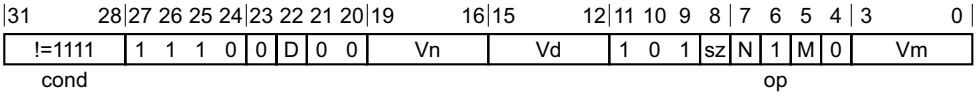
Applies when Q = 1.

```
VMLS{<c>}{<q>}.F32 <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1, sz = 0
```

##### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  add = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

#### A2




##### *Single-precision scalar variant*

Applies when sz = 0.

```
VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

##### *Double-precision scalar variant*

Applies when sz = 1.

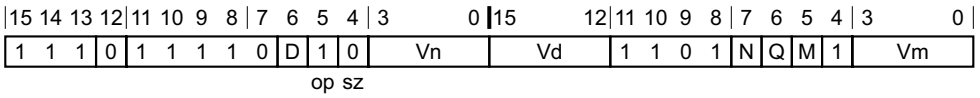
```
VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');  add = (op == '0');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## T1




### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VMLS{<c>}{<q>}.F32 <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0, sz = 0
```

### 128-bit SIMD vector variant

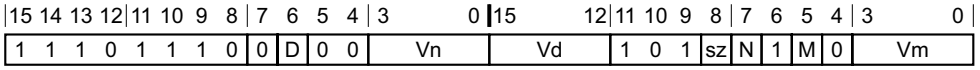
Applies when `Q = 1`.

```
VMLS{<c>}{<q>}.F32 <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1, sz = 0
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  add = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## T2




### Single-precision scalar variant

Applies when `sz = 0`.

```
VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

### Double-precision scalar variant

Applies when `sz = 1`.

```
VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');  add = (op == '0');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> For encoding A1, A2 and T1: see *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMLA or VMLS instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMLA or VMLS instruction is unconditional, see *Conditional execution* on page F2-2447.

For encoding T2: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, StandardFPSCRValue());
    else             // VFP instruction
        if dp_operation then
            addend64 = if add then FPMul(D[n], D[m], FPSCR) else FPNeg(FPMul(D[n], D[m], FPSCR));
            D[d] = FPAdd(D[d], addend64, FPSCR);
        else
            addend32 = if add then FPMul(S[n], S[m], FPSCR) else FPNeg(FPMul(S[n], S[m], FPSCR));
            S[d] = FPAdd(S[d], addend32, FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

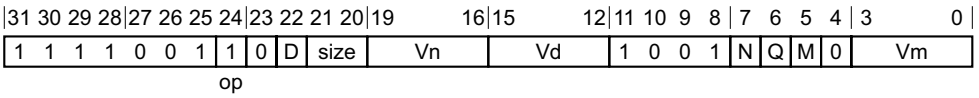
## F7.1.112 VMLS (integer)

Vector Multiply Subtract multiplies corresponding elements in two vectors, and subtracts the products from the corresponding elements of the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMLS{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0
```

#### *128-bit SIMD vector variant*

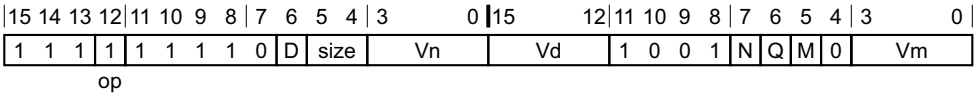
Applies when `Q = 1`.

```
VMLS{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMLS{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // Encoding T1/A1, encoded as Q = 0
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMLS{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // Encoding T1/A1, encoded as Q = 1
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> The data type for the elements of the operands. It must be one of:

- S — Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.
- U — Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.
- I — Available only in encoding T1/A1.

<size> The data size for the elements of the operands. It must be one of:

- 8 — Encoded as size = 0b00.
- 16 — Encoded as size = 0b01.
- 32 — Encoded as size = 0b10.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.113 VMLS (by scalar)

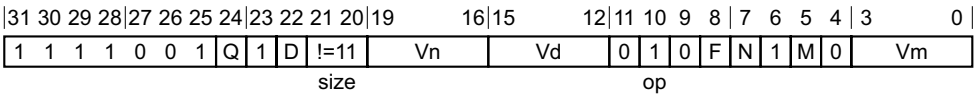
Vector Multiply Subtract multiplies elements of a vector by a scalar, and either subtracts the products from corresponding elements of the destination vector.

For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMLS{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm[x]> // Encoding T1/A1, encoded as Q = 0
```

#### *128-bit SIMD vector variant*

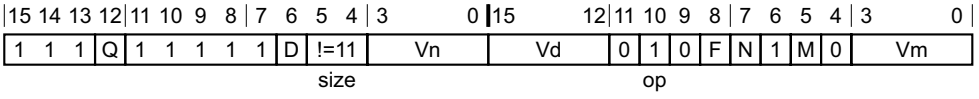
Applies when `Q = 1`.

```
VMLS{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Dm[x]> // Encoding T1/A1, encoded as Q = 1
```

#### *Decode for all variants of this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMLS{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm[x]> // Encoding T1/A1, encoded as Q = 0
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMLS{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Dm[x]> // Encoding T1/A1, encoded as Q = 1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

## Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VMLA`, `VMLAL`, `VMLS`, or `VMLSL` instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VMLA`, `VMLAL`, `VMLS`, or `VMLSL` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<type>` The data type for the elements of the operands. It must be one of:

- `S` Encoding T2/A2, encoded as U = 0.
- `U` Encoding T2/A2, encoded as U = 1.
- `I` Encoding T1/A1, encoded as F = 0.
- `F` Encoding T1/A1, encoded as F = 1. `<size>` must be 32.

`<size>` The operand element data size. It can be:

- `16` Encoded as size = 01.
- `32` Encoded as size = 10.

`<Qd>` Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm[x]>` The scalar. `Dm` is restricted to D0-D7 if `<size>` is 16, or D0-D15 otherwise.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if long_destination then
    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
else
    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

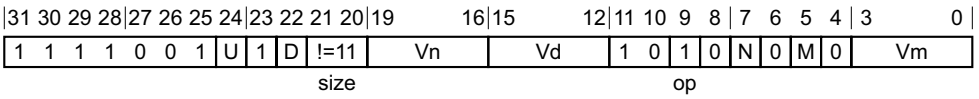
### F7.1.114 VMLSL (integer)

Vector Multiply Subtract Long multiplies corresponding elements in two vectors, and subtract the products from the corresponding elements of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

#### A1



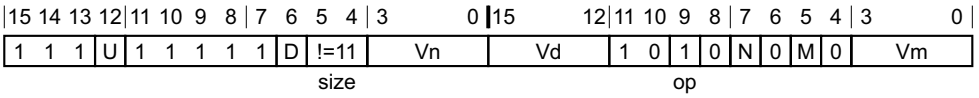

##### *A1 variant*

```
VMLSL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // Encoding T2/A2
```

##### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

#### T1




##### *T1 variant*

```
VMLSL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // Encoding T2/A2
```

##### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction is unconditional, see *Conditional execution* on page F2-2447.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> The data type for the elements of the operands. It must be one of:

S Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.

U Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.

I Available only in encoding T1/A1.

<size> The data size for the elements of the operands. It must be one of:

8 Encoded as size = 0b00.

16 Encoded as size = 0b01.

32 Encoded as size = 0b10.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F7.1.115 VMLSL (by scalar)

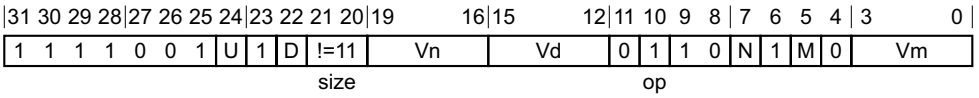
Vector Multiply Subtract Long multiplies elements of a vector by a scalar, and subtracts the products from corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

#### A1



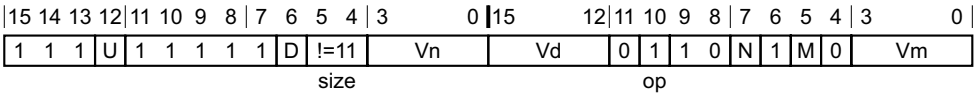

##### *A1 variant*

```
VMLSL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm[x]> // Encoding T2/A2
```

##### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

#### T1




##### *T1 variant*

```
VMLSL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm[x]> // Encoding T2/A2
```

##### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMLA, VMLAL, VMLS, or VMLSL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> The data type for the elements of the operands. It must be one of:

- S — Encoding T2/A2, encoded as U = 0.
- U — Encoding T2/A2, encoded as U = 1.
- I — Encoding T1/A1, encoded as F = 0.
- F — Encoding T1/A1, encoded as F = 1. <size> must be 32.

<size> The operand element data size. It can be:

- 16 — Encoded as size = 01.
- 32 — Encoded as size = 10.

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> The scalar. Dm is restricted to D0-D7 if <size> is 16, or D0-D15 otherwise.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

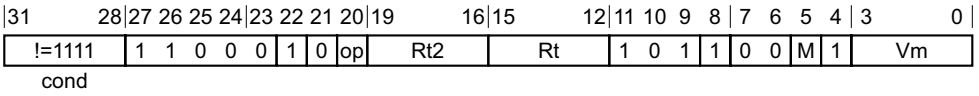

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### F7.1.116 VMOV (between two general-purpose registers and a doubleword floating-point register)

Copy two general-purpose registers to or from a SIMD&FP register copies two words from two general-purpose registers into a doubleword register in the Advanced SIMD and floating-point register file, or from a doubleword register in the Advanced SIMD and floating-point register file to two general-purpose registers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

#### A1




##### *Encoding*

Applies when `op = 1`.

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```

##### *Encoding*

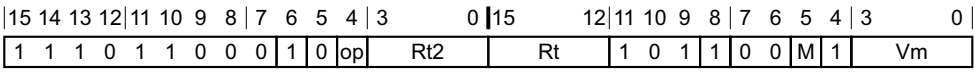
Applies when `op = 0`.

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

##### *Decode for all variants of this encoding*

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

#### T1




##### *Encoding*

Applies when `op = 1`.

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```

##### *Encoding*

Applies when `op = 0`.

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

##### *Decode for all variants of this encoding*

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
if to_arm_registers && t == t2 then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VMOV (between two general-purpose registers and a doubleword floating-point register)* on page K1-5490.

### Assembler symbols

| | |
|---|---|
| <Dm> | Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "M:Vm" field. |
| <Rt2> | Is the first general-purpose register that <Dm>[63:32] will be transferred to or from, encoded in the "Rt" field. |
| <Rt> | Is the first general-purpose register that <Dm>[31:0] will be transferred to or from, encoded in the "Rt" field. |
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = D[m]<31:0>;
        R[t2] = D[m]<63:32>;
    else
        D[m]<31:0> = R[t];
        D[m]<63:32> = R[t2];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

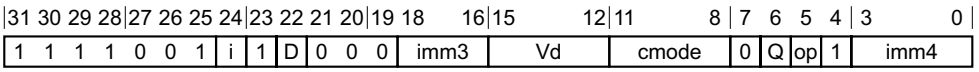
## F7.1.117 VMOV (immediate)

Copy immediate value to a SIMD&FP register places an immediate constant into every element of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>
```

#### *128-bit SIMD vector variant*

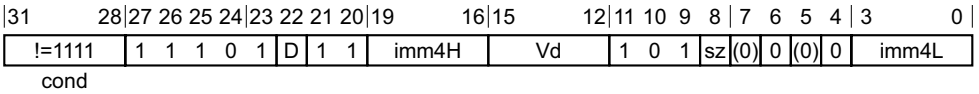
Applies when `Q = 1`.

```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE VORR (immediate);
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

### A2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```

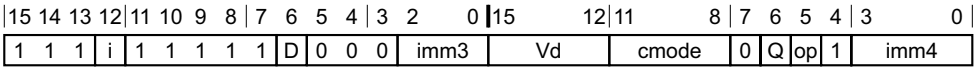
#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (sz == '0');  advsimd = FALSE;
bits(32) imm32;
bits(64) imm64;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if single_register then
    d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L);
else
    d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L);  regs = 1;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>
```

#### *128-bit SIMD vector variant*

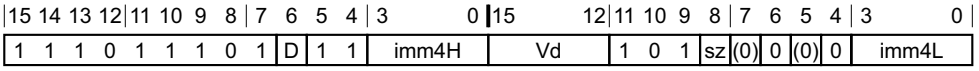
Applies when `Q = 1`.

```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE VORR (immediate);
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

### T2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (sz == '0');  advsimd = FALSE;
bits(32) imm32;
bits(64) imm64;
if single_register then
    d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L);
else
    d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L);  regs = 1;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: *One register and a modified immediate value* on page F5-2562

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMOV (immediate) instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMOV (immediate) instruction is unconditional, see *Conditional execution* on page F2-2447. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <dt> | The data type. It must be one of I8, I16, I32, I64, or F32. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <imm> | For encoding A1 and T1: is a constant of the type specified by <dt> that is replicated to fill the destination register. For details of the range of constants available and the encoding of <dt> and <imm>, see *One register and a modified immediate value* on page F5-2562.<br>For encoding A2 and T2: is a floating-point constant. For details of the range of constants available and the encoding of <imm>, see . |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = imm32;
    else
        for r = 0 to regs-1
            D[d+r] = imm64;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

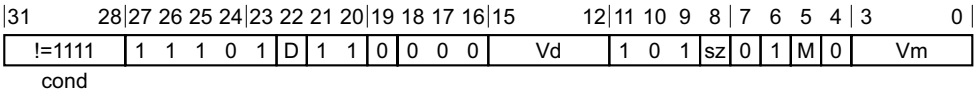
## F7.1.118 VMOV (register)

Copy between FP registers copies the contents of one FP register to another.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

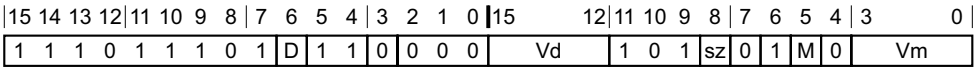
Applies when `sz = 1`.

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (sz == '0');  advsimd = FALSE;
if single_register then
    d = UInt(Vd:D);  m = UInt(Vm:M);
else
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;
```

### T2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (sz == '0');  advsimd = FALSE;
if single_register then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    d = UInt(Vd:D);  m = UInt(Vm:M);
else
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMOV (register) instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMOV (register) instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = S[m];
    else
        for r = 0 to regs-1
            D[d+r] = D[m+r];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

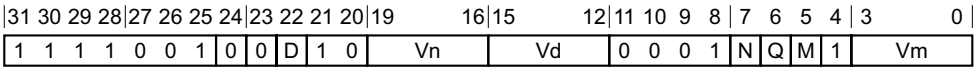
## F7.1.119 VMOV (register, SIMD)

Copy between SIMD registers copies the contents of one SIMD register to another

This instruction is an alias of the VORR (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (register).
- The description of VORR (register) gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMOV{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

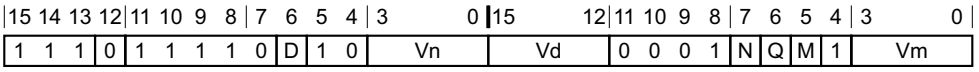
```
VMOV{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMOV{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMOV{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

`VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>`

and is the preferred disassembly when `N:Vn == M:Vm`.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VMOV` (register) instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VMOV` (register) instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` An optional data type. `<dt>` must not be `F64`, but it is otherwise ignored.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

### Operation for all encodings

The description of VORR (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.120 VMOV (general-purpose register to scalar)

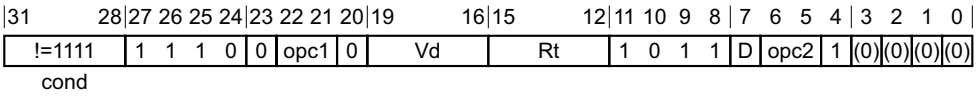
Copy a general-purpose register to a vector element copies a byte, halfword, or word from a general-purpose register into an Advanced SIMD scalar.

On a Floating-point-only system, this instruction transfers one word to the upper or lower half of a double-precision floating-point register from a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1



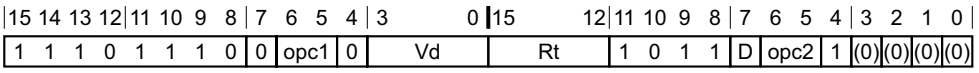

#### *A1 variant*

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

#### *Decode for this encoding*

```
case opc1:opc2 of
    when "1xxx"  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when "0xx1"  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when "0x00"  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when "0x10"  UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T1




#### *T1 variant*

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

#### *Decode for this encoding*

```
case opc1:opc2 of
    when "1xxx"  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when "0xx1"  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when "0x00"  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when "0x10"  UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` The data size. It must be one of:

- `8` Encoded as `opc1<1>` = 1. `[x]` is encoded in `opc1<0>`, `opc2`.
- `16` Encoded as `opc1<1>` = 0, `opc2<0>` = 1. `[x]` is encoded in `opc1<0>`, `opc2<1>`.
- `32` Encoded as `opc1<1>` = 0, `opc2` = `0b00`. `[x]` is encoded in `opc1<0>`.
- `omitted` Equivalent to `32`.

`<Dd[x]>` The scalar. The register `<Dd>` is encoded in D:Vd. For details of how `[x]` is encoded, see the description of `<size>`.

`<Rt>` The source general-purpose register.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    Elem[D[d],index,esize] = R[t]<esize-1:0>;
```

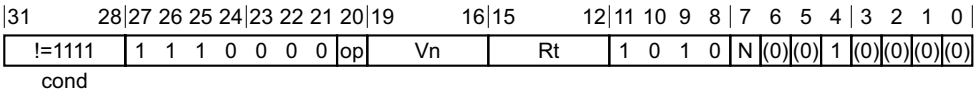

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.121 VMOV (between general-purpose register and single-precision register)

Copy a general-purpose register to or from a 32-bit SIMD&FP register transfers the contents of a single-precision Floating-point register to a general-purpose register, or the contents of a general-purpose register to a single-precision Floating-point register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Encoding*

Applies when `op = 1`.

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

#### *Encoding*

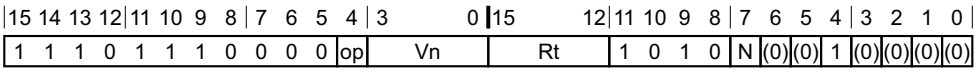
Applies when `op = 0`.

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

#### *Decode for all variants of this encoding*

```
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T1




#### *Encoding*

Applies when `op = 0`.

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

#### *Encoding*

Applies when `op = 1`.

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

#### *Decode for all variants of this encoding*

```
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<Rt> Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field.

<Sn> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_register then
        R[t] = S[n];
    else
        S[n] = R[t];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.122 VMOV (scalar to general-purpose register)

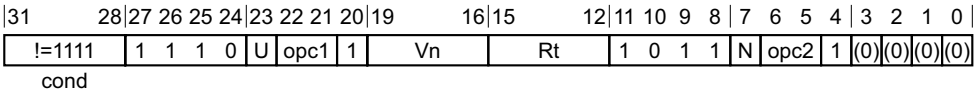
Copy a vector element to a general-purpose register with sign or zero extension copies a byte, halfword, or word from an Advanced SIMD scalar to a general-purpose register. Bytes and halfwords can be either zero-extended or sign-extended.

On a Floating-point-only system, this instruction transfers one word from the upper or lower half of a double-precision floating-point register to a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1



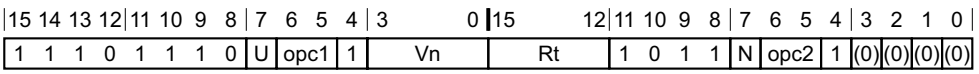

#### *A1 variant*

```
VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>
```

#### *Decode for this encoding*

```
case U:opc1:opc2 of
    when "x1xxx"  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when "x0xx1"  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when "00x00"  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when "10x00"  UNDEFINED;
    when "x0x10"  UNDEFINED;
t = UInt(Rt);  n = UInt(N:Vn);  unsigned = (U == '1');
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T1




#### *T1 variant*

```
VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>
```

#### *Decode for this encoding*

```
case U:opc1:opc2 of
    when "x1xxx"  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when "x0xx1"  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when "00x00"  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when "10x00"  UNDEFINED;
    when "x0x10"  UNDEFINED;
t = UInt(Rt);  n = UInt(N:Vn);  unsigned = (U == '1');
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` The data type. It must be one of:

| | |
|---|---|
| `S8` | Encoded as U = 0, `opc1<1>` = 1. `[x]` is encoded in `opc1<0>`, `opc2`. |
| `S16` | Encoded as U = 0, `opc1<1>` = 0, `opc2<0>` = 1. `[x]` is encoded in `opc1<0>`, `opc2<1>`. |
| `U8` | Encoded as U = 1, `opc1<1>` = 1. `[x]` is encoded in `opc1<0>`, `opc2`. |
| `U16` | Encoded as U = 1, `opc1<1>` = 0, `opc2<0>` = 1. `[x]` is encoded in `opc1<0>`, `opc2<1>`. |
| `32` | Encoded as U = 0, `opc1<1>` = 0, `opc2` = `0b00`. `[x]` is encoded in `opc1<0>`. |
| `omitted` | Equivalent to `32`. |

`<Rt>` The destination general-purpose register.

`<Dn[x]>` The scalar. For details of how `[x]` is encoded see the description of `<dt>`.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if unsigned then
        R[t] = ZeroExtend(Elem[D[n],index,esize], 32);
    else
        R[t] = SignExtend(Elem[D[n],index,esize], 32);
```

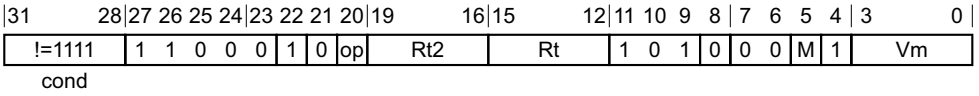

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.123 VMOV (between two general-purpose registers and two single-precision registers)

Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers transfers the contents of two consecutively numbered single-precision Floating-point registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision Floating-point registers. The general-purpose registers do not have to be contiguous.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Encoding*

Applies when `op = 1`.

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
```

#### *Encoding*

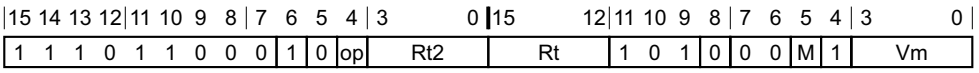
Applies when `op = 0`.

```
VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

#### *Decode for all variants of this encoding*

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if CurrentInstrSet() != InstrSet_A32 && (t == 13 || t2 == 13) then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

### T1




#### *Encoding*

Applies when `op = 1`.

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
```

#### *Encoding*

Applies when `op = 0`.

```
VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

#### *Decode for all variants of this encoding*

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if CurrentInstrSet() != InstrSet_A32 && (t == 13 || t2 == 13) then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VMOV (between two general-purpose registers and two single-precision registers)* on page K1-5490.

### Assembler symbols

| | |
|---|---|
| <Rt2> | Is the second general-purpose register that <Sm1> will be transferred to or from, encoded in the "Rt" field. |
| <Rt> | Is the first general-purpose register that <Sm> will be transferred to or from, encoded in the "Rt" field. |
| <Sm1> | Is the 32-bit name of the second SIMD&FP register to be transferred. This is the next SIMD&FP register after <Sm>. |
| <Sm> | Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Vm:M" field. |
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = S[m];
        R[t2] = S[m+1];
    else
        S[m] = R[t];
        S[m+1] = R[t2];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

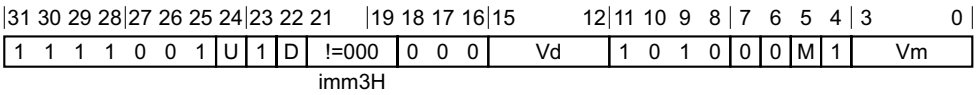
## F7.1.124 VMOVL

Vector Move Long takes each element in a doubleword vector, sign or zero-extends them to twice their original length, and places the results in a quadword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



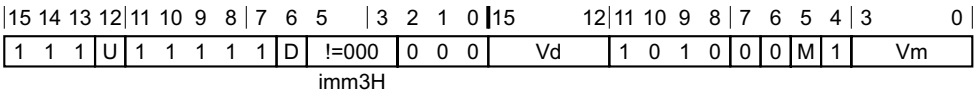

#### *A1 variant*

```
VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>
```

#### *Decode for this encoding*

```
if imm3 == '000' then SEE "Related encodings";
if imm3 != '001' && imm3 != '010' && imm3 != '100' then SEE VSHLL;
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>
```

#### *Decode for this encoding*

```
if imm3 == '000' then SEE "Related encodings";
if imm3 != '001' && imm3 != '010' && imm3 != '100' then SEE VSHLL;
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VMOVL instruction must be unconditional. ARM strongly recommends that a T32 VMOVL instruction is unconditional, see *Conditional execution* on page F2-2447.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> The data type for the elements of the operand. It must be one of:

- S8 Encoded as U = 0, imm3 = 0b001.
- S16 Encoded as U = 0, imm3 = 0b010.
- S32 Encoded as U = 0, imm3 = 0b100.
- U8 Encoded as U = 1, imm3 = 0b001.
- U16 Encoded as U = 1, imm3 = 0b010.
- U32 Encoded as U = 1, imm3 = 0b100.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.125 VMOVN

Vector Move and Narrow copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

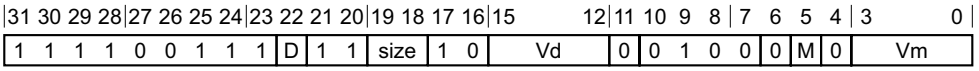
The operand vector elements can be any one of 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

This instruction is used by the aliases VRSHRN (zero) and VSHRN (zero). See the *Alias conditions* on page F7-3494 table for details of when each alias is preferred.

### A1



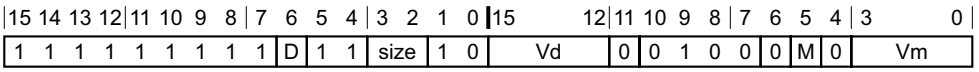

#### *A1 variant*

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

#### *Decode for this encoding*

```
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

#### *Decode for this encoding*

```
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Alias conditions

| Alias | is preferred when |
|---|---|
| VRSHRN (zero) | Never |
| VSHRN (zero) | Never |

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VMOVN instruction must be unconditional. ARM strongly recommends that a T32 VMOVN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

I16 when size = 00

I32 when size = 01

I64 when size = 10

The encoding size = 11 is reserved.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        Elem[D[d],e,esize] = Elem[Qin[m>>1],e,2*esize]<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.126 VMRS

Move SIMD&FP Special register to general-purpose register moves the value of an Advanced SIMD and floating-point System register to a general-purpose register. When the specified System register is the FPSCR, a form of the instruction transfers the FPSCR.{N, Z, C, V} condition flags to the APSR.{N, Z, C, V} condition flags.

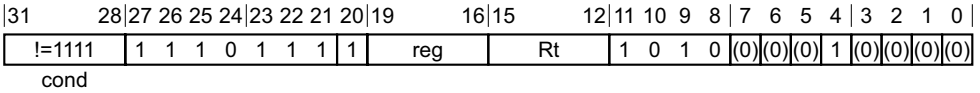
Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

When these settings permit the execution of floating-point and Advanced SIMD instructions, if the specified floating-point System register is not the FPSCR, the instruction is UNDEFINED if executed in User mode.

In an implementation that includes EL2, when HCR.TID0 is set to 1, any VMRS access to FPSID from a Non-secure EL1 mode that would be permitted if HCR.TID0 was set to 0 generates a Hyp Trap exception. For more information, see *ID group 0, Primary device identification registers* on page G1-3863.

For simplicity, the VMRS pseudocode does not show the possible trap to Hyp mode.

### A1



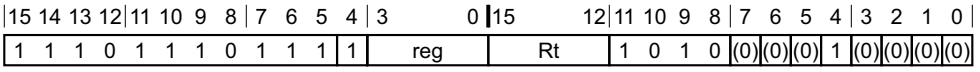

#### *A1 variant*

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

#### *Decode for this encoding*

```
t = UInt(Rt);
if !(reg IN {'000x', '0101', '011x', '1000'}) then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T1




#### *T1 variant*

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

#### *Decode for this encoding*

```
t = UInt(Rt);
if !(reg IN {'000x', '0101', '011x', '1000'}) then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VMRS* on page K1-5491.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<Rt> Is the general-purpose destination register, encoded in the "Rt" field. Is one of:

R0-R14 General-purpose register.

APSR_nzcv If <spec_reg> is FPSCR, encoded as 0b1111. This instruction transfers the FPSCR.{N, Z, C, V} condition flags to the APSR.{N, Z, C, V} condition flags.

<spec_reg> Is the source Advanced SIMD and floating-point System register, encoded in the "reg" field. It can have the following values:

FPSID when reg = 0000

FPSCR when reg = 0001

MVFR2 when reg = 0101

MVFR1 when reg = 0110

MVFR0 when reg = 0111

FPEXC when reg = 1000

The following encodings are reserved:

- reg = 001x.
- reg = 0100.
- reg = 1001.
- reg = 101x.
- reg = 11xx.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then                 // FPSCR
        CheckVFPEnabled(TRUE);
        if t == 15 then
            PSTATE.<N,Z,C,V> = FPSR.<N,Z,C,V>;
        else
            R[t] = FPSCR;
    elsif PSTATE.EL == EL0 then
        UNDEFINED;                        // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEnabled(FALSE);           // Non-FPSCR registers are not affected by FPEXC.EN
        case reg of
            // Pseudocode does not consider possible HCR.TIDn Hyp Traps of Non-secure register reads
            when '0000'  R[t] = FPSID;
            when '0101'  R[t] = MVFR2;
            when '0110'  R[t] = MVFR1;
            when '0111'  R[t] = MVFR0;
            when '1000'  R[t] = FPEXC;
            otherwise    Unreachable();   // Dealt with above or in encoding-specific pseudocode
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.127 VMSR

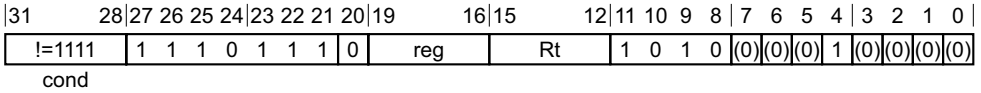
Move general-purpose register to SIMD&FP Special register moves the value of a general-purpose register to a floating-point System register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

When these settings permit the execution of floating-point and Advanced SIMD instructions:

- If the specified floating-point System register is not the FPSCR, the instruction is UNDEFINED if executed in User mode.
- If the specified floating-point System register is the FPEXC and the instruction is executed in a mode other than User mode the instruction is ignored.

### A1



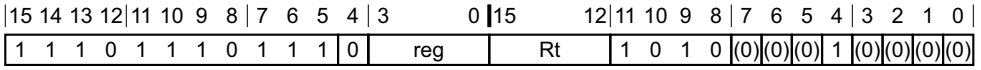

#### *A1 variant*

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

#### *Decode for this encoding*

```
t = UInt(Rt);
if reg != '000x' && reg != '1000' then UNPREDICTABLE;
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### T1




#### *T1 variant*

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

#### *Decode for this encoding*

```
t = UInt(Rt);
if reg != '000x' && reg != '1000' then UNPREDICTABLE;
if t == 15 then UNPREDICTABLE; // ARMv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VMSR* on page K1-5491.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<spec_reg> Is the destination Advanced SIMD and floating-point System register, encoded in the "reg" field. It can have the following values:

FPSID when reg = 0000

FPSCR when reg = 0001

FPEXC when reg = 1000

The following encodings are reserved:

- reg = 001x.
- reg = 01xx.
- reg = 1001.
- reg = 101x.
- reg = 11xx.

<Rt> Is the general-purpose source register, encoded in the "Rt" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then                // FPSCR
        CheckVFPEnabled(TRUE);
        FPSCR = R[t];
    elsif PSTATE.EL == EL0 then
        UNDEFINED;                       // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEnabled(FALSE);          // Non-FPSCR registers are not affected by FPEXC.EN
        case reg of
            when '0000'                  // VMSR access to FPSID is ignored
            when '1000'  FPEXC = R[t];
            otherwise    Unreachable();  // Dealt with above or in encoding-specific pseudocode
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

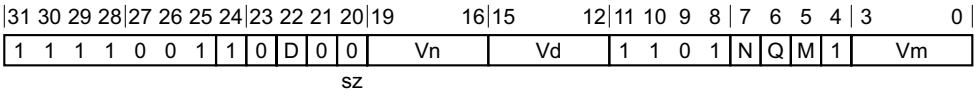
## F7.1.128 VMUL (floating-point)

Vector Multiply multiplies corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMUL{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

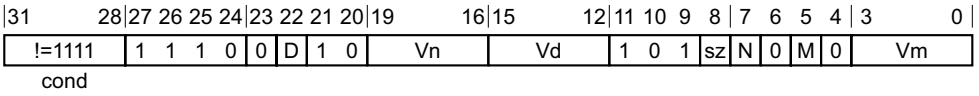
Applies when `Q = 1`.

```
VMUL{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### A2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

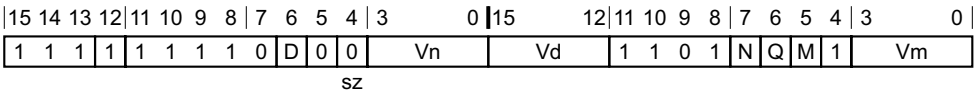
```
VMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMUL{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

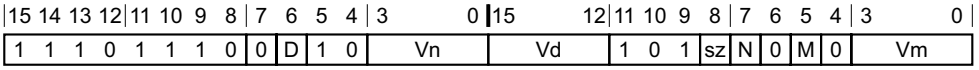
Applies when `Q = 1`.

```
VMUL{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## Assembler symbols

<c> For encoding A1, A2 and T1: see *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VMUL` instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VMUL` instruction is unconditional, see *Conditional execution* on page F2-2447.

For encoding T2: see *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize],
StandardFPSCRValue());
    else             // VFP instruction
        if dp_operation then
            D[d] = FPMul(D[n], D[m], FPSCR);
        else
            S[d] = FPMul(S[n], S[m], FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.129 VMUL (integer and polynomial)

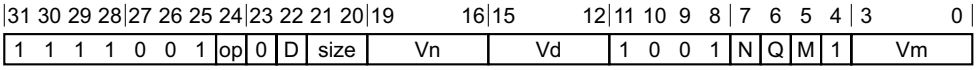
Vector Multiply multiplies corresponding elements in two vectors.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* on page A1-45.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

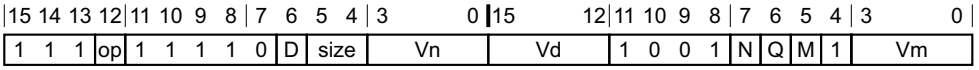
Applies when `Q = 1`.

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
polynomial = (op == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
polynomial = (op == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMUL or VMULL instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMUL or VMULL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "op:size" field. It can have the following values:

| | |
|---|---|
| I8 | when op = 0, size = 00 |
| I16 | when op = 0, size = 01 |
| I32 | when op = 0, size = 10 |
| P8 | when op = 1, size = 00 |

The following encodings are reserved:

- op = 0, size = 11.
- op = 1, size = 01.
- op = 1, size = 1x.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize];  op2val = Int(op2, unsigned);
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<2*esize-1:0>;
            if long_destination then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[Q[d>>1],e,2*esize] = product;
    else
        Elem[D[d+r],e,esize] = product<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.130 VMUL (by scalar)

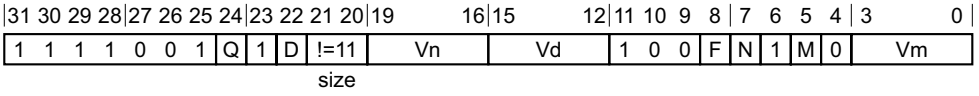
Vector Multiply multiplies each element in a vector by a scalar, and places the results in a second vector.

For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMUL{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>[<index>]
```

#### *128-bit SIMD vector variant*

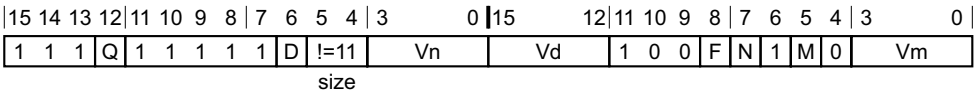
Applies when `Q = 1`.

```
VMUL{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>[<index>]
```

#### *Decode for all variants of this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMUL{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>[<index>]
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMUL{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>[<index>]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

## Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMUL or VMULL instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMUL or VMULL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in the "F:size" field. It can have the following values:

I16 when F = 0, size = 01

I32 when F = 0, size = 10

F32 when F = 1, size = 10

The following encodings are reserved:

- F = 0, size = 00.
- F = 0, size = 11.
- F = 1, size = 0x.
- F = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is I16, otherwise the "Vm" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is I16, otherwise in range 0 to 1, encoded in the "M" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, StandardFPSCRValue());
            else
                if long_destination then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
    else
        Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.131 VMULL (integer and polynomial)

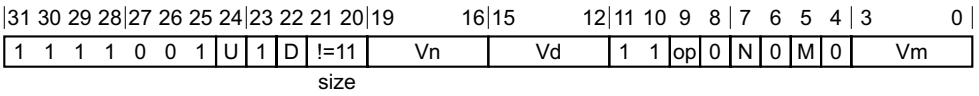
Vector Multiply Long multiplies corresponding elements in two vectors. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* on page A1-45.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



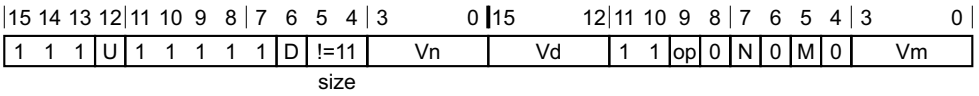

#### *A1 variant*

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
    if U == '1' || size == '01' then UNDEFINED;
    if size == '10' then    // .p64
        if !HaveCryptoExt() then UNDEFINED;
        if InITBlock() then UNPREDICTABLE;
        esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;
```

### T1




#### *T1 variant*

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
    if U == '1' || size == '01' then UNDEFINED;
    if size == '10' then    // .p64
        if !HaveCryptoExt() then UNDEFINED;
        if InITBlock() then UNPREDICTABLE;
        esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMUL or VMULL instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMUL or VMULL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "op:U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when op = 0, U = 0, size = 00 |
| S16 | when op = 0, U = 0, size = 01 |
| S32 | when op = 0, U = 0, size = 10 |
| U8 | when op = 0, U = 1, size = 00 |
| U16 | when op = 0, U = 1, size = 01 |
| U32 | when op = 0, U = 1, size = 10 |
| P8 | when op = 1, U = 0, size = 00 |
| P64 | when op = 1, U = 0, size = 10 |

The following encodings are reserved:

- op = 0, U = 0, size = 11.
- op = 0, U = 1, size = 11.
- op = 1, U = 0, size = 01.
- op = 1, U = 0, size = 11.
- op = 1, U = 1, size = xx.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize];  op2val = Int(op2, unsigned);
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<2*esize-1:0>;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.132 VMULL (by scalar)

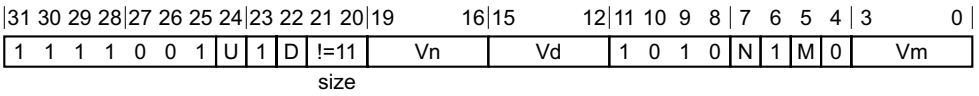
Vector Multiply Long multiplies each element in a vector by a scalar, and places the results in a second vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



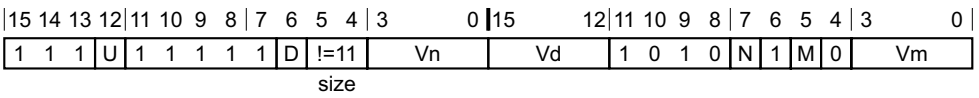

#### *A1 variant*

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

### T1




#### *T1 variant*

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VMUL or VMULL instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VMUL or VMULL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in the "U:size" field. It can have the following values:

S16 when U = 0, size = 01

S32 when U = 0, size = 10

U16 when U = 1, size = 01

U32 when U = 1, size = 10

The following encodings are reserved:

- U = 0, size = 00.
- U = 0, size = 11.
- U = 1, size = 00.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16 or U16, otherwise the "Vm" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16 or U16, otherwise in range 0 to 1, encoded in the "M" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, StandardFPSCRValue());
            else
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

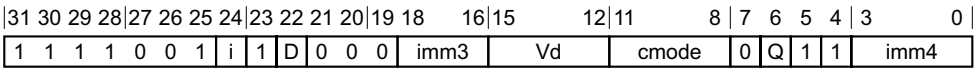
### F7.1.133 VMVN (immediate)

Vector Bitwise NOT (immediate) places the bitwise inverse of an immediate integer constant into every element of the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

#### A1




##### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMVN{<c>}{<q>}.<dt> <Dd>, #<imm>
```

##### *128-bit SIMD vector variant*

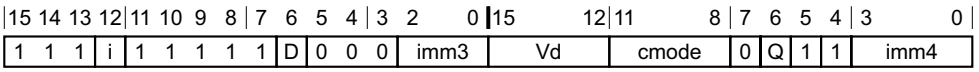
Applies when `Q = 1`.

```
VMVN{<c>}{<q>}.<dt> <Qd>, #<imm>
```

##### *Decode for all variants of this encoding*

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

#### T1




##### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMVN{<c>}{<q>}.<dt> <Dd>, #<imm>
```

##### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMVN{<c>}{<q>}.<dt> <Qd>, #<imm>
```

##### *Decode for all variants of this encoding*

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. An A32 VMVN instruction must be unconditional. ARM strongly recommends that a T32 VMVN instruction is unconditional, see *Conditional execution* on page F2-2447. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <dt> | The data type. It must be either I16 or I32. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <imm> | Is a constant of the type specified by <dt> that is replicated to fill the destination register. For details of the range of constants available and the encoding of <dt> and <imm>, see *One register and a modified immediate value* on page F5-2562. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(imm64);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

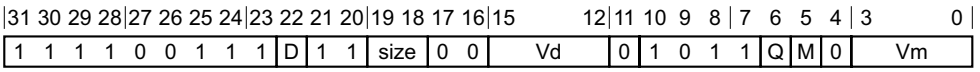
### F7.1.134 VMVN (register)

Vector Bitwise NOT (register) takes a value from a register, inverts the value of each bit, and places the result in the destination register. The registers can be either doubleword or quadword.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

#### A1




##### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

##### *128-bit SIMD vector variant*

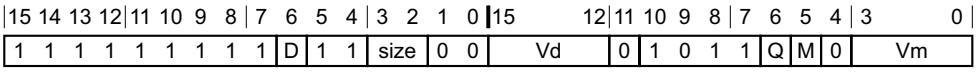
Applies when `Q = 1`.

```
VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

##### *Decode for all variants of this encoding*

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

#### T1




##### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

##### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

##### *Decode for all variants of this encoding*

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VMVN` instruction must be unconditional. ARM strongly recommends that a T32 `VMVN` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(D[m+r]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

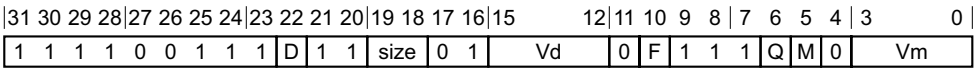
## F7.1.135 VNEG

Vector Negate negates each element in a vector, and places the results in a second vector. The floating-point version only inverts the sign bit.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

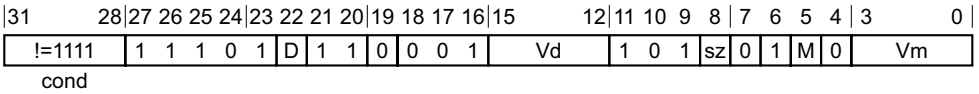
Applies when `Q = 1`.

```
VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### A2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VNEG{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

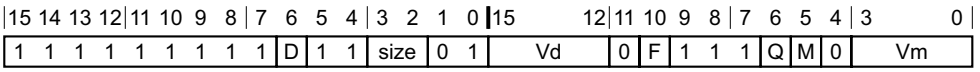
```
VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

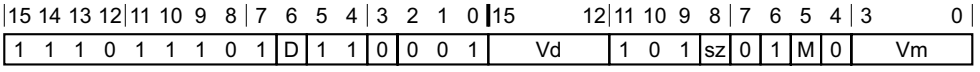
Applies when `Q = 1`.

```
VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VNEG{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VNEG` instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VNEG` instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<dt> Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:

S8 when F = 0, size = 00

S16 when F = 0, size = 01

S32 when F = 0, size = 10

F32 when F = 1, size = 10

The following encodings are reserved:

- F = 0, size = 11.
- F = 1, size = 0x.
- F = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPNeg(Elem[D[m+r],e,esize]);
                else
                    result = -SInt(Elem[D[m+r],e,esize]);
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else             // VFP instruction
        if dp_operation then
            D[d] = FPNeg(D[m]);
        else
            S[d] = FPNeg(S[m]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.136 VNMLA

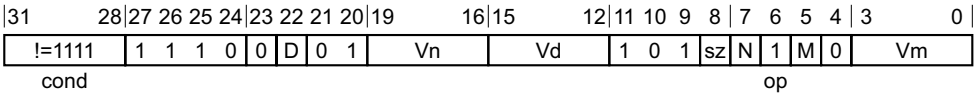
Vector Negate Multiply Accumulate multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

**Note**

ARM recommends that software does not use the VNMLA instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when sz = 0.

```
VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

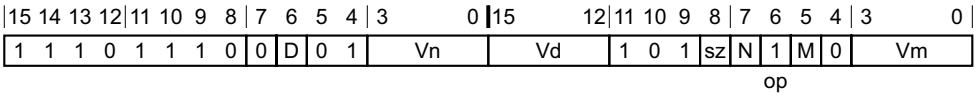
Applies when sz = 1.

```
VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
type = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Single-precision scalar variant*

Applies when sz = 0.

```
VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when sz = 1.

```
VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
type = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        product64 = FPMul(D[n], D[m], FPSCR);
        case type of
            when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR);
            when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR);
            when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
    else
        product32 = FPMul(S[n], S[m], FPSCR);
        case type of
            when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR);
            when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR);
            when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
```

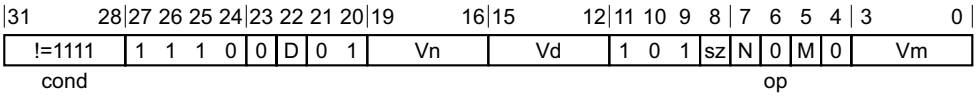

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.137 VNMLS

Vector Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

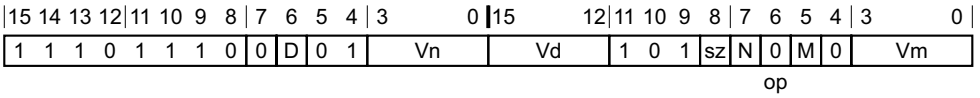
Applies when `sz = 1`.

```
VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
type = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
type = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        product64 = FPMul(D[n], D[m], FPSCR);
        case type of
            when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR);
            when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR);
            when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
    else
        product32 = FPMul(S[n], S[m], FPSCR);
        case type of
            when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR);
            when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR);
            when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
```

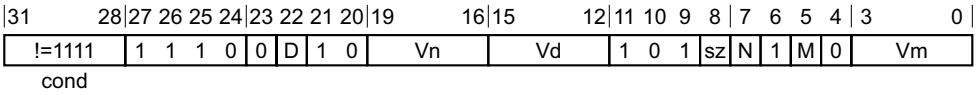

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.138 VNMUL

Vector Negate Multiply multiplies together two floating-point register values, and writes the negation of the result to the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

#### *Double-precision scalar variant*

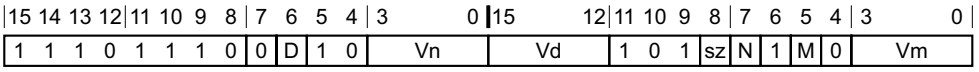
Applies when `sz = 1`.

```
VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
type = VFPNegMul_VNMUL;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
type = VFPNegMul_VNMUL;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        product64 = FPMul(D[n], D[m], FPSCR);
        case type of
            when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR);
            when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR);
            when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
    else
        product32 = FPMul(S[n], S[m], FPSCR);
        case type of
            when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR);
            when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR);
            when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

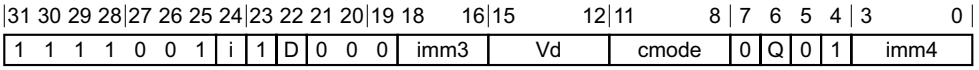
## F7.1.139 VORN (immediate)

Vector Bitwise OR NOT (immediate) performs a bitwise OR between a register value and the complement of an immediate value, and returns the result into the destination vector

This instruction is an alias of the VORR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (immediate).
- The description of VORR (immediate) gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VORN{<c>}{<q>}.<dt> {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.<dt> <Dd>, #~<imm>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

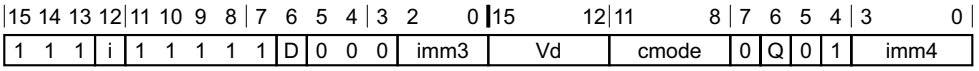
```
VORN{<c>}{<q>}.<dt> {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.<dt> <Qd>, #~<imm>
```

and is never the preferred disassembly.

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VORN{<c>}{<q>}.<dt> {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.<dt> <Dd>, #~<imm>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VORN{<c>}{<q>}.<dt> {<Qd>,} <Qd>, #<imm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

```
VORR{<c>}{<q>}.<dt> <Qd>, #~<imm>
```

and is never the preferred disassembly.

### Assembler symbols

| | |
|---|---|
| `<c>` | See *Standard assembler syntax fields* on page F2-2446. An A32 `VORR` instruction must be unconditional. ARM strongly recommends that a T32 `VORR` instruction is unconditional, see *Conditional execution* on page F2-2447. |
| `<q>` | See *Standard assembler syntax fields* on page F2-2446. |
| `<dt>` | The data type used for `<imm>`. It can be either `I16` or `I32`. `I8`, `I64`, and `F32` are also permitted, but the resulting syntax is a pseudo-instruction. |
| `<Qd>` | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| `<Dd>` | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| `<imm>` | Is a constant of the type specified by `<dt>` that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<dt>` and `<imm>`, see *One register and a modified immediate value* on page F5-2562. |

### Operation for all encodings

The description of VORR (immediate) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

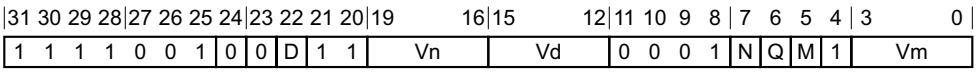
## F7.1.140 VORN (register)

Vector bitwise OR NOT (register) performs a bitwise OR NOT operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VORN{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

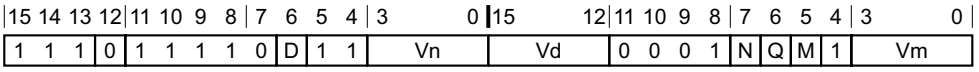
Applies when `Q = 1`.

```
VORN{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VORN{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VORN{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. An A32 VORN instruction must be unconditional. ARM strongly recommends that a T32 VORN instruction is unconditional, see *Conditional execution* on page F2-2447. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <dt> | An optional data type. It is ignored by assemblers, and does not affect the encoding. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR NOT(D[m+r]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.141 VORR (immediate)

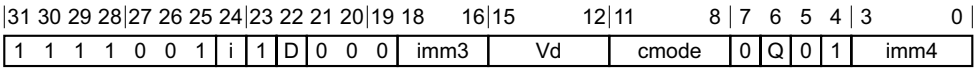
Vector Bitwise OR (immediate) performs a bitwise OR between a register value and an immediate value, and returns the result into the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

This instruction is used by the alias VORN (immediate). See the *Alias conditions* on page F7-3530 table for details of when each alias is preferred.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VORR{<c>}{<q>}.<dt> {<Dd>,} <Dd>, #<imm>
```

#### *128-bit SIMD vector variant*

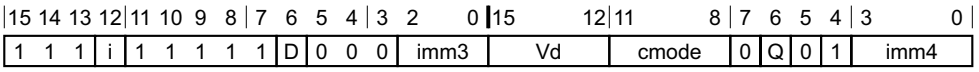
Applies when `Q = 1`.

```
VORR{<c>}{<q>}.<dt> {<Qd>,} <Qd>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE VMOV (immediate);
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VORR{<c>}{<q>}.<dt> {<Dd>,} <Dd>, #<imm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VORR{<c>}{<q>}.<dt> {<Qd>,} <Qd>, #<imm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Decode for all variants of this encoding*

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE VMOV (immediate);
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

### Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562

### Alias conditions

| Alias | is preferred when |
|---|---|
| VORN (immediate) | Never |

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VORR instruction must be unconditional. ARM strongly recommends that a T32 VORR instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> The data type used for <imm>. It can be either I16 or I32. I8, I64, and F32 are also permitted, but the resulting syntax is a pseudo-instruction.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<imm> Is a constant of the type specified by <dt> that is replicated to fill the destination register. For details of the range of constants available and the encoding of <dt> and <imm>, see *One register and a modified immediate value* on page F5-2562.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] OR imm64;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.142 VORR (register)

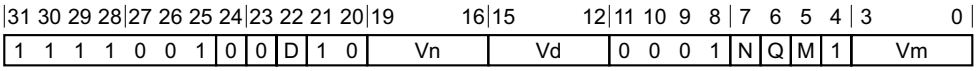
Vector bitwise OR (register) performs a bitwise OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

This instruction is used by the aliases VMOV (register, SIMD), VRSHR (zero), and VSHR (zero). See the *Alias conditions* on page F7-3532 table for details of when each alias is preferred.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VORR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

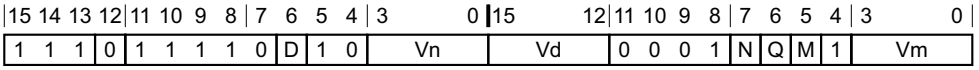
Applies when `Q = 1`.

```
VORR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VORR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VORR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Alias conditions

| Alias | is preferred when |
|---|---|
| VMOV (register, SIMD) | `N:Vn == M:Vm` |
| VRSHR (zero) | `Never` |
| VSHR (zero) | `Never` |

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VORR` instruction must be unconditional. ARM strongly recommends that a T32 `VORR` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

`<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

`<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR D[m+r];
```

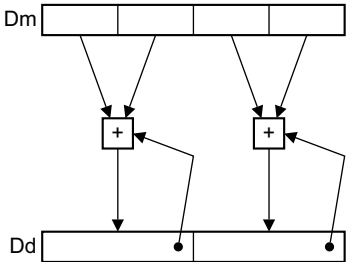

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.143 VPADAL

Vector Pairwise Add and Accumulate Long adds adjacent pairs of elements of a vector, and accumulates the results into the elements of the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows an example of the operation of VPADAL doubleword operation for data type S16.




Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | D | 1 | 1 | size | | 0 | 0 | Vd | | | | 0 | 1 | 1 | 0 | op | Q | M | 0 | Vm | | | |

#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | size | | 0 | 0 | Vd | | | | 0 | 1 | 1 | 0 | op | Q | M | 0 | Vm | | | |

#### *64-bit SIMD vector variant*

Applies when `Q = 0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### 128-bit SIMD vector variant

Applies when Q = 1.

```
VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VPADAL instruction must be unconditional. ARM strongly recommends that a T32 VPADAL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "op:size" field. It can have the following values:

| | |
|---|---|
| S8 | when op = 0, size = 00 |
| S16 | when op = 0, size = 01 |
| S32 | when op = 0, size = 10 |
| U8 | when op = 1, size = 00 |
| U16 | when op = 1, size = 01 |
| U32 | when op = 1, size = 10 |

The following encodings are reserved:

- op = 0, size = 11.
- op = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = Elem[D[d+r],e,2*esize] + result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.144 VPADD (floating-point)

Vector Pairwise Add (floating-point) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

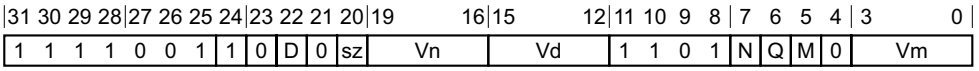
The operands and result are doubleword vectors.

The operand and result elements are 32-bit floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



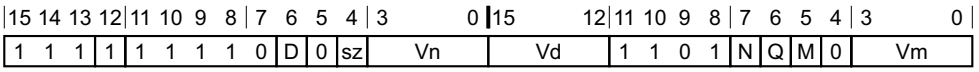

#### *A1 variant*

```
VPADD{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if sz == '1' || Q == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VPADD{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if sz == '1' || Q == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VPADD instruction must be unconditional. ARM strongly recommends that a T32 VPADD instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        Elem[dest,e,esize]   = FPAdd(Elem[D[n],2*e,esize], Elem[D[n],2*e+1,esize],
StandardFPSCRValue());
        Elem[dest,e+h,esize] = FPAdd(Elem[D[m],2*e,esize], Elem[D[m],2*e+1,esize],
StandardFPSCRValue());

    D[d] = dest;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

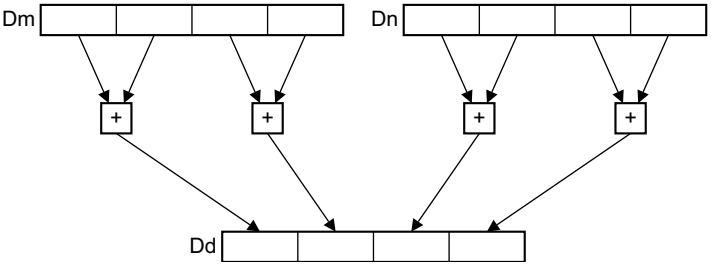
## F7.1.145 VPADD (integer)

Vector Pairwise Add (integer) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements must all be the same type, and can be 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The following figure shows an example of the operation of VPADD doubleword operation for data type I16.




Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19–16 | 15–12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3–0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | D | size | | Vn | Vd | 1 | 0 | 1 | 1 | N | Q | M | 1 | Vm |

#### *A1 variant*

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3–0 | 15–12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3–0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | D | size | | Vn | Vd | 1 | 0 | 1 | 1 | N | Q | M | 1 | Vm |

#### *T1 variant*

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VPADD instruction must be unconditional. ARM strongly recommends that a T32 VPADD instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

I8 when size = 00

I16 when size = 01

I32 when size = 10

The encoding size = 11 is reserved.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        Elem[dest,e,esize]   = Elem[D[n],2*e,esize] + Elem[D[n],2*e+1,esize];
        Elem[dest,e+h,esize] = Elem[D[m],2*e,esize] + Elem[D[m],2*e+1,esize];

    D[d] = dest;
```

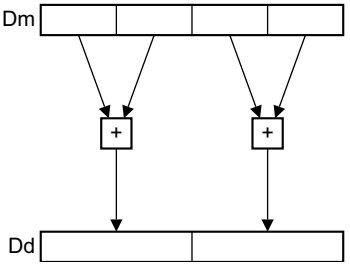

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.146 VPADDL

Vector Pairwise Add Long adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows an example of the operation of VPADDL doubleword operation for data type S16.




Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | D | 1 | 1 | size | | 0 | 0 | Vd | | | | 0 | 0 | 1 | 0 | op | Q | M | 0 | Vm | | | |

#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | | size | | 0 | 0 | Vd | | | | 0 | 0 | 1 | 0 | op | Q | M | 0 | Vm | | | |

#### *64-bit SIMD vector variant*

Applies when `Q = 0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### 128-bit SIMD vector variant

Applies when Q = 1.

```
VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VPADDL instruction must be unconditional. ARM strongly recommends that a T32 VPADDL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "op:size" field. It can have the following values:

| | |
|---|---|
| S8 | when op = 0, size = 00 |
| S16 | when op = 0, size = 01 |
| S32 | when op = 0, size = 10 |
| U8 | when op = 1, size = 00 |
| U16 | when op = 1, size = 01 |
| U32 | when op = 1, size = 10 |

The following encodings are reserved:

- op = 0, size = 11.
- op = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = result<2*esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

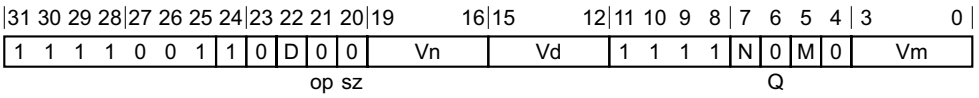
## F7.1.147 VPMAX (floating-point)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



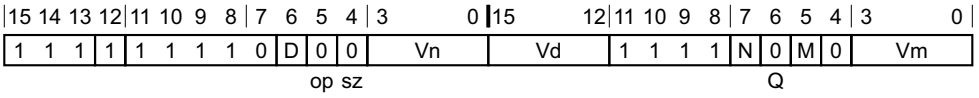

#### *A1 variant*

```
VPMAX{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if sz == '1' || Q == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VPMAX{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if sz == '1' || Q == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VPMAX or VPMIN instruction must be unconditional. ARM strongly recommends that a T32 VPMAX or VPMIN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDEnabled();
     bits(64) dest;
     h = elements DIV 2;

     for e = 0 to h-1
         op1 = Elem[D[n],2*e,esize];  op2 = Elem[D[n],2*e+1,esize];
         Elem[dest,e,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else
FPMin(op1,op2,StandardFPSCRValue());
         op1 = Elem[D[m],2*e,esize];  op2 = Elem[D[m],2*e+1,esize];
         Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else
FPMin(op1,op2,StandardFPSCRValue());

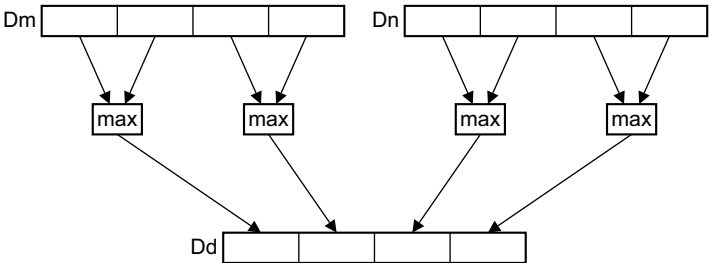
     D[d] = dest;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.148 VPMAX (integer)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

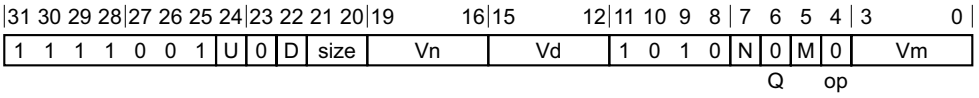
The following figure shows an example of the operation of VPMAX doubleword operation for data type S16 or U16.




Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



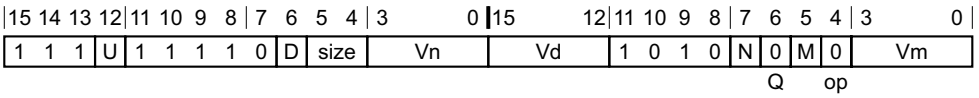

#### *A1 variant*

```
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' || Q == '1' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' || Q == '1' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VPMAX or VPMIN instruction must be unconditional. ARM strongly recommends that a T32 VPMAX or VPMIN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8 when U = 0, size = 00

S16 when U = 0, size = 01

S32 when U = 0, size = 10

U8 when U = 1, size = 00

U16 when U = 1, size = 01

U32 when U = 1, size = 10

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

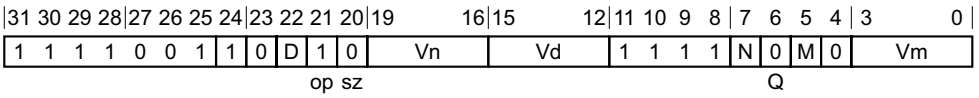
## F7.1.149 VPMIN (floating-point)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



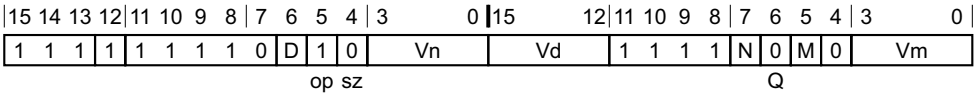

#### *A1 variant*

```
VPMIN{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if sz == '1' || Q == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VPMIN{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if sz == '1' || Q == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VPMAX or VPMIN instruction must be unconditional. ARM strongly recommends that a T32 VPMAX or VPMIN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize];  op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else
FPMin(op1,op2,StandardFPSCRValue());
        op1 = Elem[D[m],2*e,esize];  op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else
FPMin(op1,op2,StandardFPSCRValue());

    D[d] = dest;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

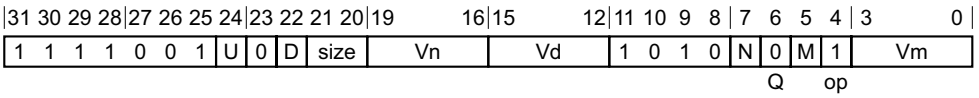
## F7.1.150 VPMIN (integer)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



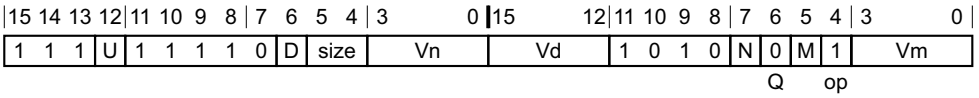

#### *A1 variant*

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' || Q == '1' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' || Q == '1' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VPMAX or VPMIN instruction must be unconditional. ARM strongly recommends that a T32 VPMAX or VPMIN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

S8 when U = 0, size = 00

S16 when U = 0, size = 01


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

S32 when U = 0, size = 10

U8 when U = 1, size = 00

U16 when U = 1, size = 01

U32 when U = 1, size = 10

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

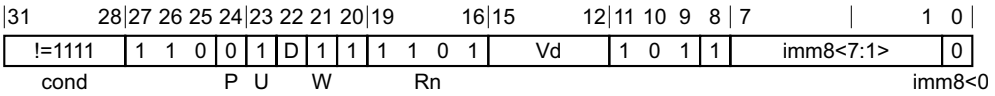
## F7.1.151 VPOP

Pop SIMD&FP registers from Stack loads multiple consecutive Advanced SIMD and floating-point register file registers from the stack

This instruction is an alias of the VLDM, VLDMDB, VLDMIA instruction. This means that:

- The encodings in this description are named to match the encodings of VLDM, VLDMDB, VLDMIA.
- The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.

### A1




#### *Increment After variant*

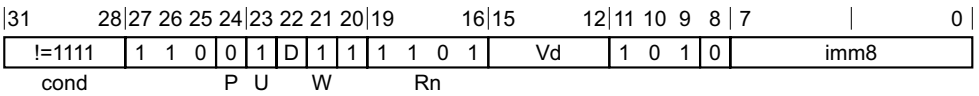
```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

### A2




#### *Increment After variant*

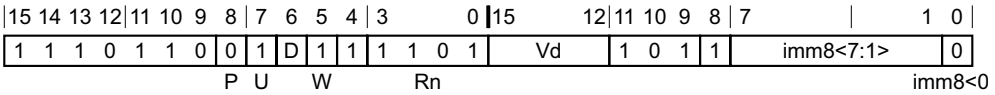
```
VPOP{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

### T1




#### *Increment After variant*

```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

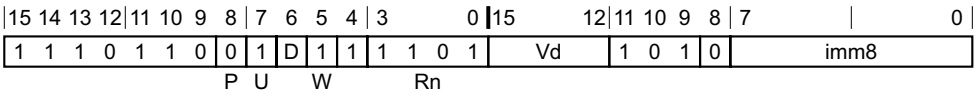
is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *Increment After variant*

`VPOP{<c>}{<q>}{.<size>} <sreglist>`

is equivalent to

`VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>`

and is always the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

`<sreglist>` Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

`<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

### Operation for all encodings

The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

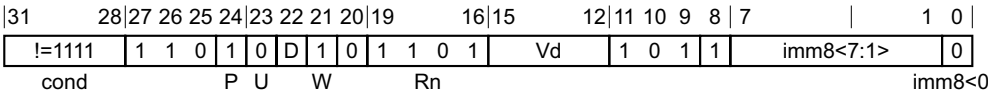
## F7.1.152 VPUSH

Push SIMD&FP registers to Stack stores multiple consecutive registers from the Advanced SIMD and floating-point register file to the stack

This instruction is an alias of the VSTM, VSTMDB, VSTMIA instruction. This means that:

- The encodings in this description are named to match the encodings of VSTM, VSTMDB, VSTMIA.
- The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

### A1




#### *Decrement Before variant*

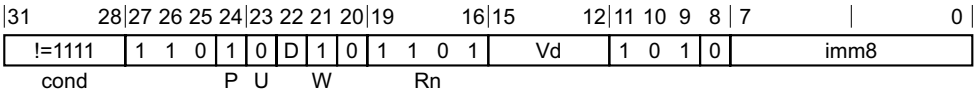
```
VPUSH{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

### A2




#### *Decrement Before variant*

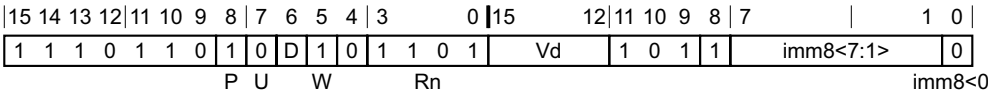
```
VPUSH{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

### T1




#### *Decrement Before variant*

```
VPUSH{<c>}{<q>}{.<size>} <dreglist>
```

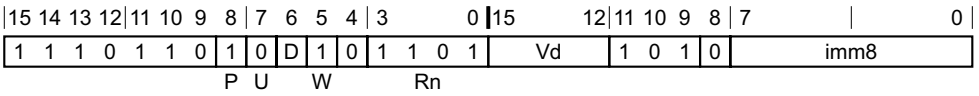
is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T2




#### *Decrement Before variant*

`VPUSH{<c>}{<q>}{.<size>} <sreglist>`

is equivalent to

`VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>`

and is always the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

`<sreglist>` Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

`<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

### Operation for all encodings

The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.153 VQABS

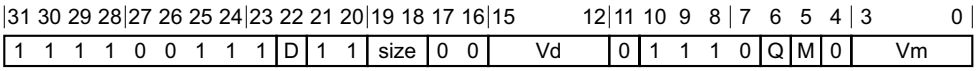
Vector Saturating Absolute takes the absolute value of each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

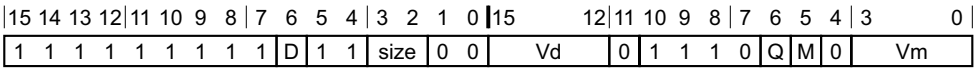
Applies when `Q = 1`.

```
VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQABS instruction must be unconditional. ARM strongly recommends that a T32 VQABS instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

S8 when size = 00

S16 when size = 01

S32 when size = 10

The encoding size = 11 is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Abs(SInt(Elem[D[m+r],e,esize]));
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.154 VQADD

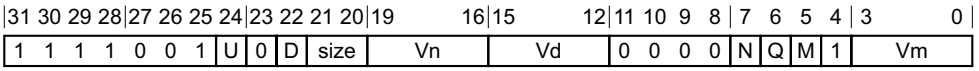
Vector Saturating Add adds the values of corresponding elements of two vectors, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

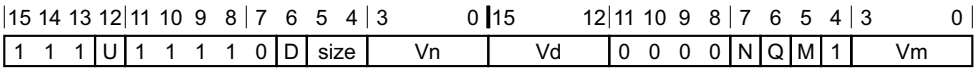
Applies when `Q = 1`.

```
VQADD{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VQADD{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQADD instruction must be unconditional. ARM strongly recommends that a T32 VQADD instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| S64 | when U = 0, size = 11 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |
| U64 | when U = 1, size = 11 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            sum = Int(Elem[D[n+r],e,esize], unsigned) + Int(Elem[D[m+r],e,esize], unsigned);
            (Elem[D[d+r],e,esize], sat) = SatQ(sum, esize, unsigned);
            if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.155 VQDMLAL

Vector Saturating Doubling Multiply Accumulate Long multiplies corresponding elements in two doubleword vectors, doubles the products, and accumulates the results into the elements of a quadword vector.

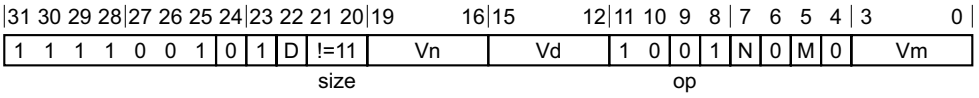
The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



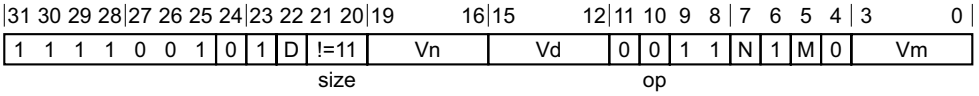

#### *A1 variant*

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

### A2




#### *A2 variant*

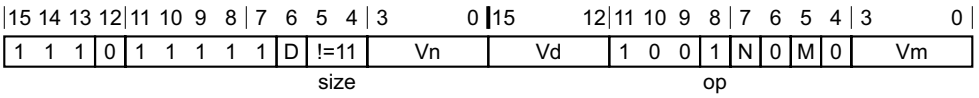
```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



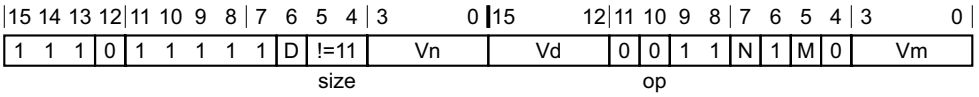

#### *T1 variant*

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

### T2




#### *T2 variant*

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

## Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQDMLAL or VQDMLSL instruction must be unconditional. ARM strongly recommends that a T32 VQDMLAL or VQDMLSL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

S16 when size = 01

S32 when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> For encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

For encoding A2 and T2: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16, otherwise the "Vm" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16, otherwise in range 0 to 1, encoded in the "M" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2*esize]) - SInt(product);
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.156 VQDMLSL

Vector Saturating Doubling Multiply Subtract Long multiplies corresponding elements in two doubleword vectors, subtracts double the products from corresponding elements of a quadword vector, and places the results in the same quadword vector.

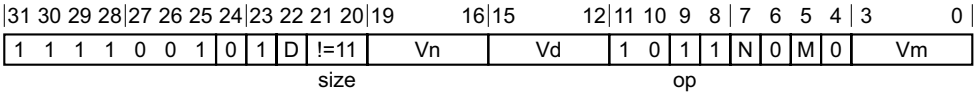
The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



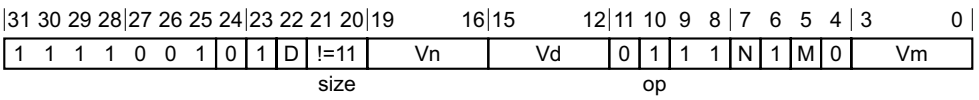

#### *A1 variant*

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

### A2




#### *A2 variant*

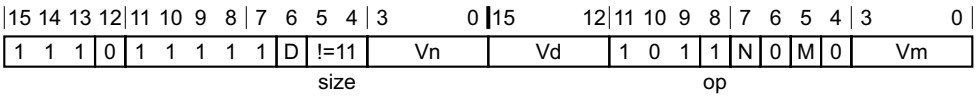
```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1



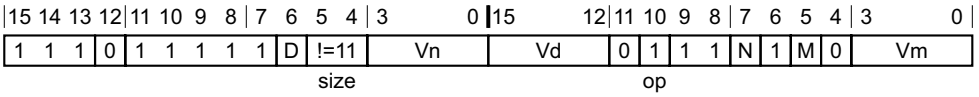

#### *T1 variant*

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

### T2




#### *T2 variant*

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

## Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQDMLAL or VQDMLSL instruction must be unconditional. ARM strongly recommends that a T32 VQDMLAL or VQDMLSL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

S16 when size = 01

S32 when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> For encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

For encoding A2 and T2: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16, otherwise the "Vm" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16, otherwise in range 0 to 1, encoded in the "M" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2*esize]) - SInt(product);
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.157 VQDMULH

Vector Saturating Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are truncated, for rounded results see VQRDMULH.

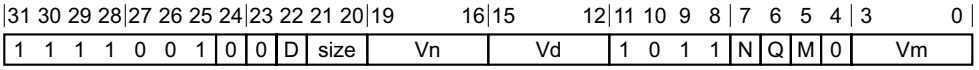
The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

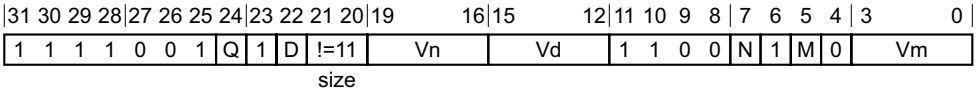
Applies when `Q = 1`.

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### A2




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

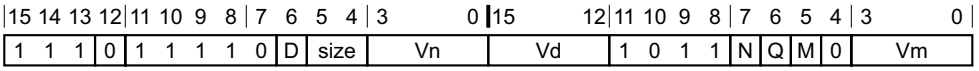
```
VQDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

## T1




### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

### 128-bit SIMD vector variant

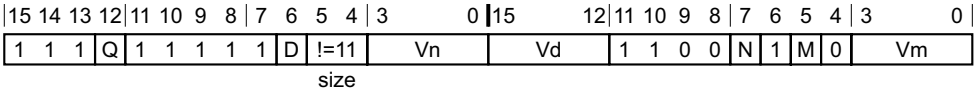
Applies when `Q = 1`.

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## T2




### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

### 128-bit SIMD vector variant

Applies when `Q = 1`.

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```

### Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQDMULH instruction must be unconditional. ARM strongly recommends that a T32 VQDMULH instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

S16 when size = 01

S32 when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> The scalar for either a quadword or a doubleword scalar operation. If <dt> is S16, Dm is restricted to D0-D7. If <dt> is S32, Dm is restricted to D0-D15.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            op1 = SInt(Elem[D[n+r],e,esize]);
            // The following only saturates if both op1 and op2 equal -(2^(esize-1))
            (result, sat) = SignedSatQ((2*op1*op2) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.158 VQDMULL

Vector Saturating Doubling Multiply Long multiplies corresponding elements in two doubleword vectors, doubles the products, and places the results in a quadword vector.

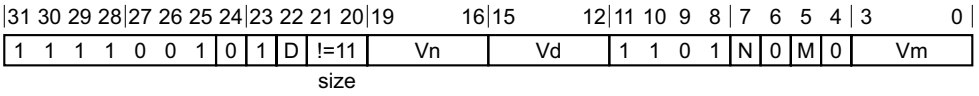
The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



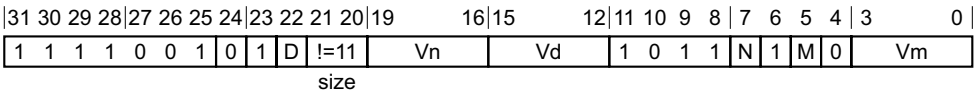

#### *A1 variant*

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

### A2



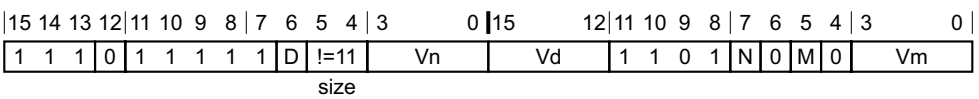

#### *A2 variant*

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]> // Encoding T2/A2
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

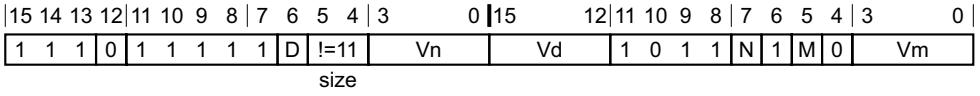
### *T1 variant*

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

## T2




### *T2 variant*

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]> // Encoding T2/A2
```

### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

## Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQDMULL instruction must be unconditional. ARM strongly recommends that a T32 VQDMULL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

S16 when size = 01

S32 when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> The scalar for a scalar operation. If <dt> is S16, Dm is restricted to D0-D7. If <dt> is S32, Dm is restricted to D0-D15.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat) = SignedSatQ(2*op1*op2, 2*esize);
        Elem[Q[d>>1],e,2*esize] = product;
        if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.159 VQMOVN, VQMOVUN

Vector Saturating Move and Narrow copies each element of the operand vector to the corresponding element of the destination vector.

The operand is a quadword vector. The elements can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result is a doubleword vector. The elements are half the length of the operand vector elements. If the operand is unsigned, the results are unsigned. If the operand is signed, the results can be signed or unsigned.

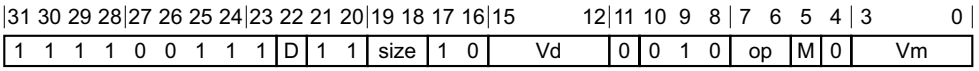
If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

This instruction is used by the aliases VQRSHRN (zero), VQRSHRUN (zero), VQSHRN (zero), and VQSHRUN (zero). See the *Alias conditions* on page F7-3570 table for details of when each alias is preferred.

### A1




#### *Signed result variant*

Applies when `op = 1x`.

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

#### *Unsigned result variant*

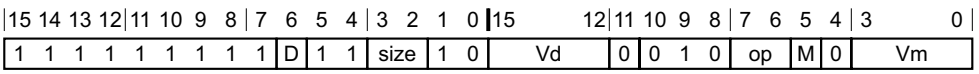
Applies when `op = 01`.

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if op == '00' then SEE VMOVN;
if size == '11' || Vm<0> == '1' then UNDEFINED;
src_unsigned = (op == '11');  dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### T1




#### *Signed result variant*

Applies when `op = 1x`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

#### *Unsigned result variant*

Applies when `op = 01`.

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if op == '00' then SEE VMOVN;
if size == '11' || Vm<0> == '1' then UNDEFINED;
src_unsigned = (op == '11');  dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### Alias conditions

| Alias | is preferred when |
|---|---|
| VQRSHRN (zero) | Never |
| VQRSHRUN (zero) | Never |
| VQSHRN (zero) | Never |
| VQSHRUN (zero) | Never |

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VQMOVN` or `VQMOVUN` instruction must be unconditional. ARM strongly recommends that a T32 `VQMOVN` or `VQMOVUN` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` For the signed result variant: is the data type for the elements of the operand, encoded in the "op:size" field. It can have the following values:

`S16` when `op = 10`, `size = 00`

`S32` when `op = 10`, `size = 01`

`S64` when `op = 10`, `size = 10`

`U8` when `op = 11`, `size = 00`

`U32` when `op = 11`, `size = 01`

`U64` when `op = 11`, `size = 10`

The following encodings are reserved:

- `op = 0x`, `size = xx`.
- `op = 10`, `size = 11`.
- `op = 11`, `size = 11`.

For the unsigned result variant: is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`S16` when `size = 00`

`S32` when `size = 01`

`S64` when `size = 10`

The encoding `size = 11` is reserved.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
        (Elem[D[d],e,esize], sat) = SatQ(operand, esize, dest_unsigned);
        if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.160 VQNEG

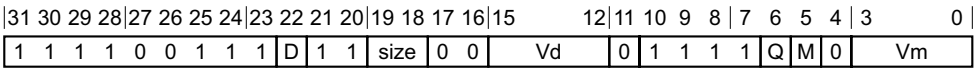
Vector Saturating Negate negates each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

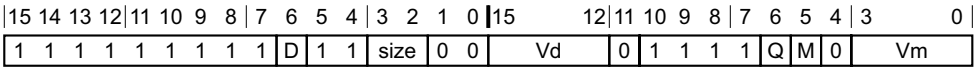
Applies when `Q = 1`.

```
VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQNEG instruction must be unconditional. ARM strongly recommends that a T32 VQNEG instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| S8 | when size = 00 |
| S16 | when size = 01 |
| S32 | when size = 10 |

The encoding size = 11 is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = -SInt(Elem[D[m+r],e,esize]);
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.161 VQRDMULH

Vector Saturating Rounding Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are rounded. For truncated results see VQDMULH.

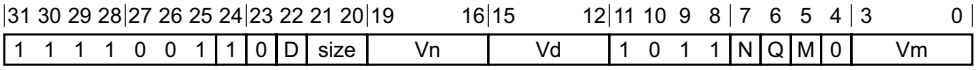
The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page F5-2552.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

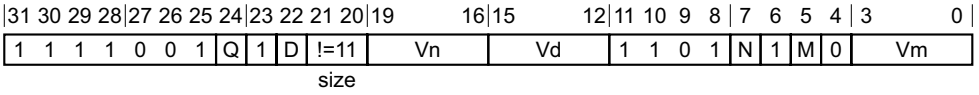
Applies when `Q = 1`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### A2




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

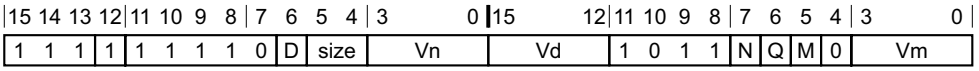
```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

### T1




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### 128-bit SIMD vector variant

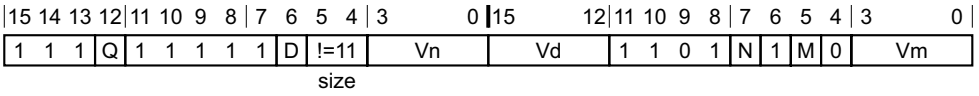
Applies when `Q = 1`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T2




#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

#### 128-bit SIMD vector variant

Applies when `Q = 1`.

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```

#### Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQRDMULH instruction must be unconditional. ARM strongly recommends that a T32 VQRDMULH instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

S16 when size = 01

S32 when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> The scalar for either a quadword or a doubleword scalar operation. If <dt> is S16, Dm is restricted to D0-D7. If <dt> is S32, Dm is restricted to D0-D15.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = SInt(Elem[D[n+r],e,esize]);
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            (result, sat) = SignedSatQ((2*op1*op2 + round_const) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.162 VQRSHL

Vector Saturating Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

For truncated results see VQSHL (register).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

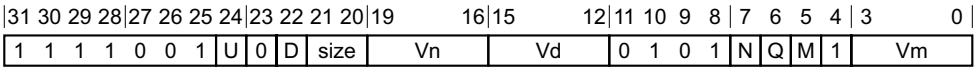
The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

#### *128-bit SIMD vector variant*

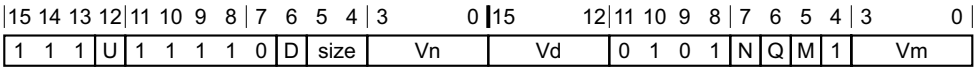
Applies when `Q = 1`.

```
VQRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### 128-bit SIMD vector variant

Applies when Q = 1.

```
VQRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQRSHL instruction must be unconditional. ARM strongly recommends that a T32 VQRSHL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| S64 | when U = 0, size = 11 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |
| U64 | when U = 1, size = 11 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            round_const = 1 << (-1-shift); // 0 for left shift, 2^(n-1) for right shift
            operand = Int(Elem[D[m+r],e,esize], unsigned);
            (result, sat) = SatQ((operand + round_const) << shift, esize, unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

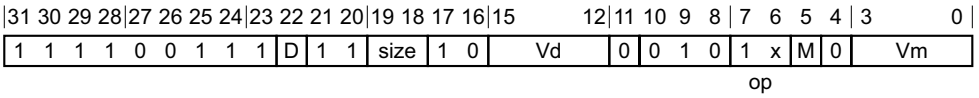
## F7.1.163 VQRSHRN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed rounded results in a doubleword vector

This instruction is an alias of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

### A1




#### *Signed result variant*

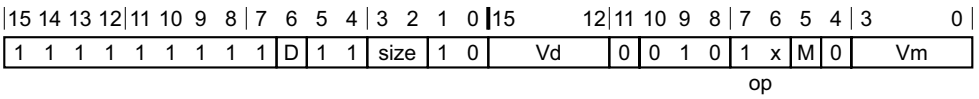
```
VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

### T1




#### *Signed result variant*

```
VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 `VQMOVN` or `VQMOVUN` instruction must be unconditional. ARM strongly recommends that a T32 `VQMOVN` or `VQMOVUN` instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operand, encoded in the "op:size" field. It can have the following values:

- `S16` when `op = 10, size = 00`
- `S32` when `op = 10, size = 01`
- `S64` when `op = 10, size = 10`
- `U8` when `op = 11, size = 00`
- `U32` when `op = 11, size = 01`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

U64 when op = 11, size = 10

The following encodings are reserved:

- op = 0x, size = xx.
- op = 10, size = 11.
- op = 11, size = 11.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.164 VQRSHRN, VQRSHRUN

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the rounded results in a doubleword vector.

For truncated results, see VQSHL (register).

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

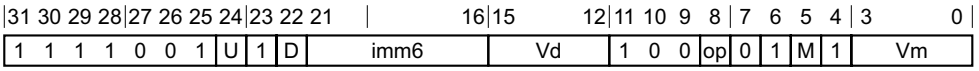
The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Signed result variant*

Applies when `!(imm6 == 000xxx) && op = 1`.

```
VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

#### *Unsigned result variant*

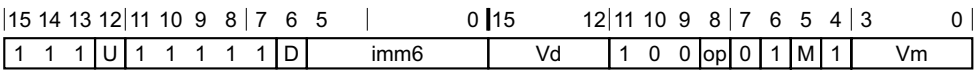
Applies when `U = 1 && !(imm6 == 000xxx) && op = 0`.

```
VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE VRSHRN;
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when "001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "01xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "1xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Signed result variant*

Applies when `!(imm6 == 000xxx) && op = 1`.

```
VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

#### *Unsigned result variant*

Applies when `U = 1 && !(imm6 == 000xxx) && op = 0`.

```
VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE VRSHRN;
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when "001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "01xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "1xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VQRSHRN` or `VQRSHRUN` instruction must be unconditional. ARM strongly recommends that a T32 `VQRSHRN` or `VQRSHRUN` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<type>` For the signed result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

- `S` when `U = 0`
- `U` when `U = 1`

For the unsigned result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

- `S` when `U = 1`

`<size>` Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:

- `16` when `imm6<5:3> = 001`
- `32` when `imm6<5:3> = 01x`
- `64` when `imm6<5:3> = 1xx`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<imm>` Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to elements-1
    operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
    (result, sat) = SatQ((operand + round_const) >> shift_amount, esize, dest_unsigned);
    Elem[D[d],e,esize] = result;
    if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

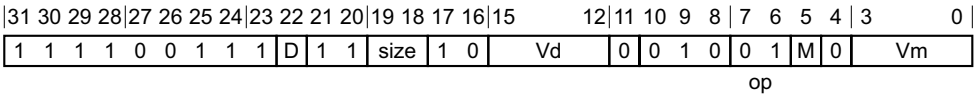
### F7.1.165 VQRSHRUN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned rounded results in a doubleword vector

This instruction is an alias of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

#### A1




##### *Unsigned result variant*

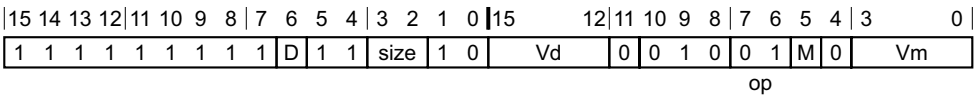
```
VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

#### T1




##### *Unsigned result variant*

```
VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

#### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 `VQMOVN` or `VQMOVUN` instruction must be unconditional. ARM strongly recommends that a T32 `VQMOVN` or `VQMOVUN` instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`S16` when `size = 00`

`S32` when `size = 01`

`S64` when `size = 10`

The encoding `size = 11` is reserved.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.166 VQSHL, VQSHLU (immediate)

Vector Saturating Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in a second vector.

The operand elements must all be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

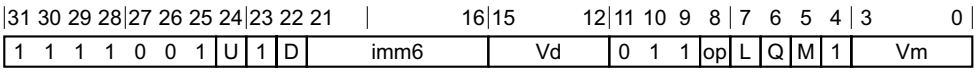
The result elements are the same size as the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *VQSHL,double,signed-result variant*

Applies when `!(imm6 == 000xxx && L == 0) && op = 1 && Q = 0`.

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *VQSHL,quad,signed-result variant*

Applies when `!(imm6 == 000xxx && L == 0) && op = 1 && Q = 1`.

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

#### *VQSHLU,double,unsigned-result variant*

Applies when `U = 1 && !(imm6 == 000xxx && L == 0) && op = 0 && Q = 0`.

```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *VQSHLU,quad,unsigned-result variant*

Applies when `U = 1 && !(imm6 == 000xxx && L == 0) && op = 0 && Q = 1`.

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

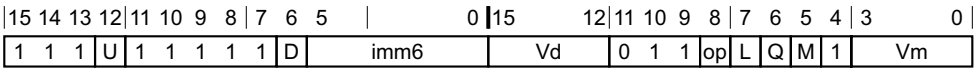
#### *Decode for all variants of this encoding*

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *VQSHL,double,signed-result variant*

Applies when `!(imm6 == 000xxx && L == 0) && op = 1 && Q = 0`.

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *VQSHL,quad,signed-result variant*

Applies when `!(imm6 == 000xxx && L == 0) && op = 1 && Q = 1`.

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

#### *VQSHLU,double,unsigned-result variant*

Applies when `U = 1 && !(imm6 == 000xxx && L == 0) && op = 0 && Q = 0`.

```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *VQSHLU,quad,unsigned-result variant*

Applies when `U = 1 && !(imm6 == 000xxx && L == 0) && op = 0 && Q = 1`.

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 `VQSHL` or `VQSHLU` instruction must be unconditional. ARM strongly recommends that a T32 `VQSHL` or `VQSHLU` instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

S when `U = 0`


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`U` when `U = 1`

`<size>` Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

- `8` when `L = 0`, `imm6<5:3> = 001`
- `16` when `L = 0`, `imm6<5:3> = 01x`
- `32` when `L = 0`, `imm6<5:3> = 1xx`
- `64` when `L = 1`, `imm6<5:3> = xxx`

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

`<imm>` Is an immediate value, in the range 0 to `<size>`-1, encoded in the "imm6" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            operand = Int(Elem[D[m+r],e,esize], src_unsigned);
            (result, sat) = SatQ(operand << shift_amount, esize, dest_unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.167 VQSHL (register)

Vector Saturating Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results are truncated. For rounded results, see VQRSHL.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

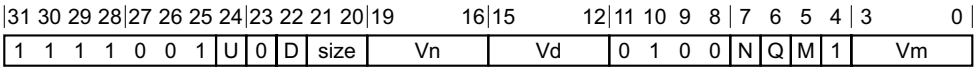
The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

#### *128-bit SIMD vector variant*

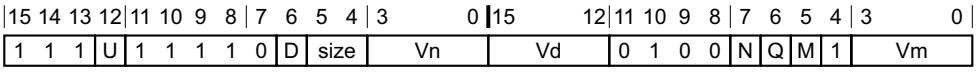
Applies when `Q = 1`.

```
VQSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### 128-bit SIMD vector variant

Applies when Q = 1.

```
VQSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

#### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQSHL instruction must be unconditional. ARM strongly recommends that a T32 VQSHL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| S64 | when U = 0, size = 11 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |
| U64 | when U = 1, size = 11 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            operand = Int(Elem[D[m+r],e,esize], unsigned);
            (result,sat) = SatQ(operand << shift, esize, unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

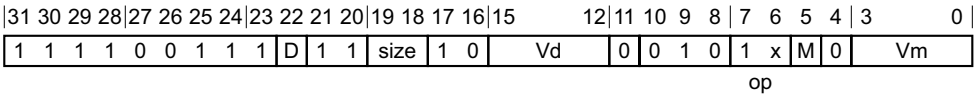
## F7.1.168 VQSHRN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed truncated results in a doubleword vector.

This instruction is an alias of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

### A1




#### *Signed result variant*

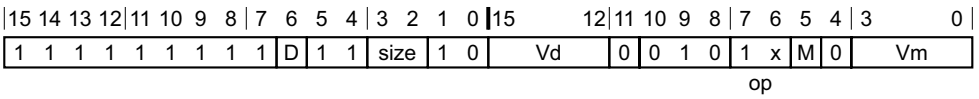
```
VQSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

### T1




#### *Signed result variant*

```
VQSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQMOVN or VQMOVUN instruction must be unconditional. ARM strongly recommends that a T32 VQMOVN or VQMOVUN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operand, encoded in the "op:size" field. It can have the following values:

- S16 when op = 10, size = 00
- S32 when op = 10, size = 01
- S64 when op = 10, size = 10
- U8 when op = 11, size = 00
- U32 when op = 11, size = 01


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

U64 when op = 11, size = 10

The following encodings are reserved:

- op = 0x, size = xx.
- op = 10, size = 11.
- op = 11, size = 11.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.169 VQSHRN, VQSHRUN

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the truncated results in a doubleword vector.

For rounded results, see VQRSHRN, VQRSHRUN.

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

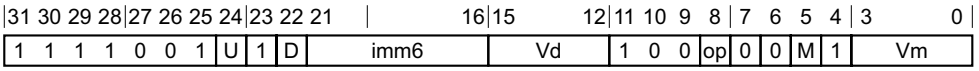
The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Signed result variant*

Applies when `!(imm6 == 000xxx) && op = 1`.

```
VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

#### *Unsigned result variant*

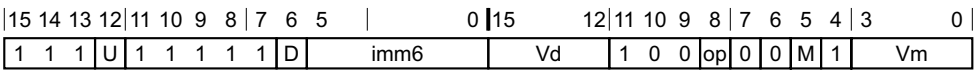
Applies when `U = 1 && !(imm6 == 000xxx) && op = 0`.

```
VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE VSHRN;
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when "001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "01xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "1xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Signed result variant*

Applies when `!(imm6 == 000xxx) && op = 1`.

```
VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

#### *Unsigned result variant*

Applies when `U = 1 && !(imm6 == 000xxx) && op = 0`.

```
VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE VSHRN;
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when "001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "01xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "1xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VQSHRN` or `VQSHRUN` instruction must be unconditional. ARM strongly recommends that a T32 `VQSHRN` or `VQSHRUN` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<type>` For the signed result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

- `S` when `U = 0`
- `U` when `U = 1`

For the unsigned result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

- `S` when `U = 1`

`<size>` Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:

- `16` when `imm6<5:3> = 001`
- `32` when `imm6<5:3> = 01x`
- `64` when `imm6<5:3> = 1xx`

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<imm>` Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
(result, sat) = SatQ(operand >> shift_amount, esize, dest_unsigned);
Elem[D[d],e,esize] = result;
if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

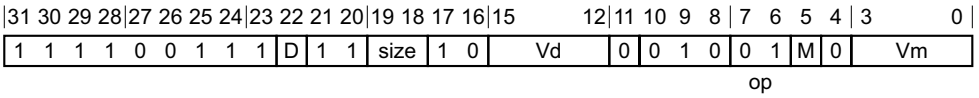
### F7.1.170 VQSHRUN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned truncated results in a doubleword vector.

This instruction is an alias of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

#### A1




##### *Unsigned result variant*

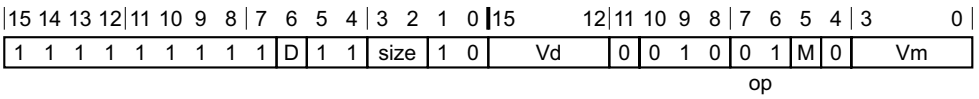
```
VQSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

#### T1




##### *Unsigned result variant*

```
VQSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQMOVN or VQMOVUN instruction must be unconditional. ARM strongly recommends that a T32 VQMOVN or VQMOVUN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

S16 when size = 00

S32 when size = 01

S64 when size = 10

The encoding size = 11 is reserved.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.171 VQSUB

Vector Saturating Subtract subtracts the elements of the second operand vector from the corresponding elements of the first operand vector, and places the results in the destination vector. Signed and unsigned operations are distinct.

The operand and result elements must all be the same type, and can be any one of:

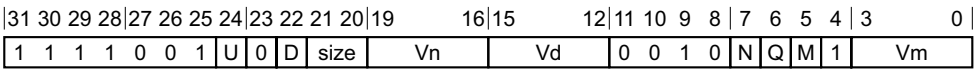
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-2315.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQSUB{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

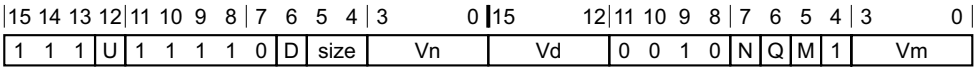
Applies when `Q = 1`.

```
VQSUB{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VQSUB{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VQSUB{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VQSUB instruction must be unconditional. ARM strongly recommends that a T32 VQSUB instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| S64 | when U = 0, size = 11 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |
| U64 | when U = 1, size = 11 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            diff = Int(Elem[D[n+r],e,esize], unsigned) - Int(Elem[D[m+r],e,esize], unsigned);
            (Elem[D[d+r],e,esize], sat) = SatQ(diff, esize, unsigned);
            if sat then FPSR.QC = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.172 VRADDHN

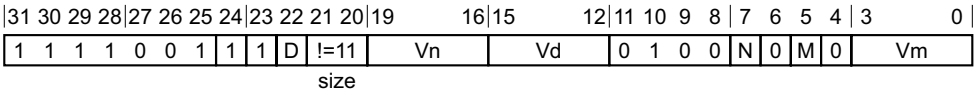
Vector Rounding Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are rounded. For truncated results, see VADDHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



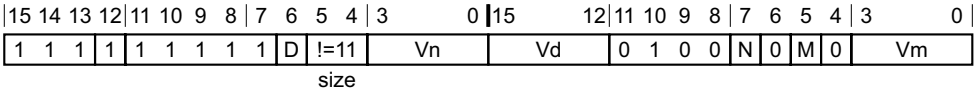

#### *A1 variant*

```
VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VRADDHN instruction must be unconditional. ARM strongly recommends that a T32 VRADDHN instruction is unconditional, see *Conditional execution* on page F2-2447.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

I16 when size = 00

I32 when size = 01

I64 when size = 10

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.173 VRECPE

Vector Reciprocal Estimate finds an approximate reciprocal of each element in the operand vector, and places the results in the destination vector.

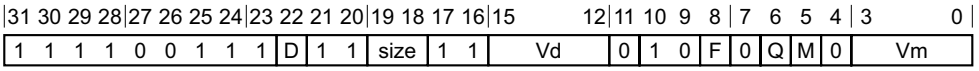
The operand and result elements are the same type, and can be 32-bit floating-point numbers, or 32-bit unsigned integers.

For details of the operation performed by this instruction see *Floating-point reciprocal square root estimate and step* on page E1-2347.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

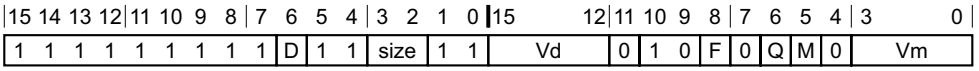
Applies when `Q = 1`.

```
VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
floating_point = (F == '1');  esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
floating_point = (F == '1');  esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VRECPE instruction must be unconditional. ARM strongly recommends that a T32 VRECPE instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:

S32 when F = 0, size = 10

F32 when F = 1, size = 10

The following encodings are reserved:

- F = 0, size = 0x.
- F = 0, size = 11.
- F = 1, size = 0x.
- F = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see *Floating-point reciprocal estimate and step* on page E1-2344.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,32] = FPRecipEstimate(Elem[D[m+r],e,32], StandardFPSCRValue());
            else
                Elem[D[d+r],e,32] = UnsignedRecipEstimate(Elem[D[m+r],e,32]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.174 VRECPS

Vector Reciprocal Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 2.0, and places the results into the elements of the destination vector.

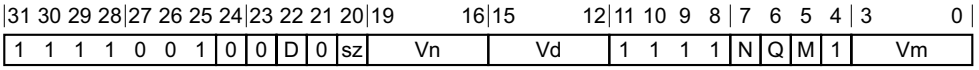
The operand and result elements are 32-bit floating-point numbers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step* on page E1-2344.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRECPS{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

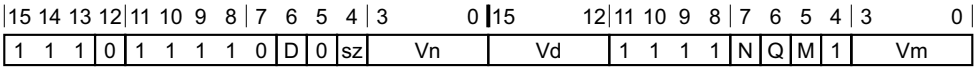
Applies when `Q = 1`.

```
VRECPS{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRECPS{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRECPS{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VRECPS instruction must be unconditional. ARM strongly recommends that a T32 VRECPS instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see *Floating-point reciprocal estimate and step* on page E1-2344.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,32] = FPRecipStep(Elem[D[n+r],e,32], Elem[D[m+r],e,32]);
```

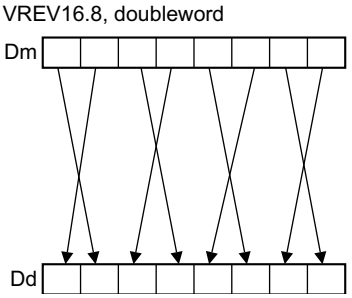

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.175 VREV16

Vector Reverse in halfwords reverses the order of 8-bit elements in each halfword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows an example of the operation of VREV16 doubleword operation.




Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | D | 1 | 1 | size | | 0 | 0 | Vd | | | | 0 | 0 | 0 | 1 | 0 | Q | M | 0 | Vm | | | |
| | | | | | | | | | | | | | | | | | | | | | | | op | | | | | | | | |

#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

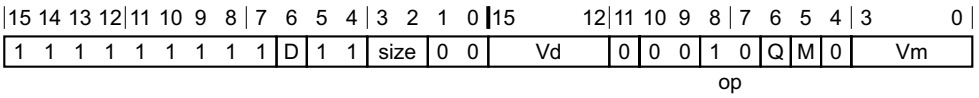
```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 `VREV` instruction must be unconditional. ARM strongly recommends that a T32 `VREV` instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

The following encodings are reserved:

- `size = 01`.
- `size = 1x`.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(64) result;
integer element;
integer rev_element;
for r = 0 to regs-1
    element = 0;
    for c = 0 to containers-1
        rev_element = element + elements_per_container - 1;
        for e = 0 to elements_per_container-1
            Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
            element = element + 1;
            rev_element = rev_element - 1;
    D[d+r] = result;
```

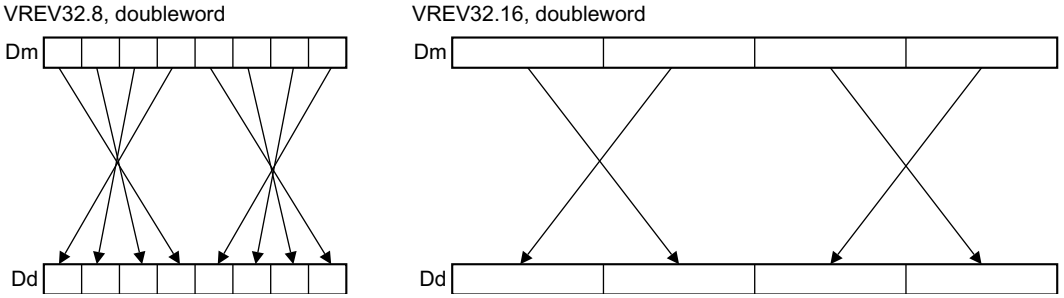

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.176 VREV32

Vector Reverse in words reverses the order of 8-bit or 16-bit elements in each word of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

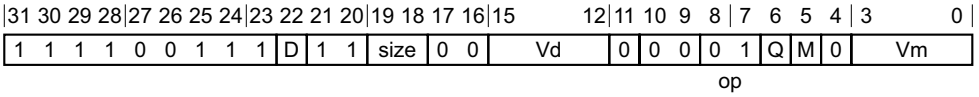
The following figure shows an example of the operation of VREV32 doubleword operations.




Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

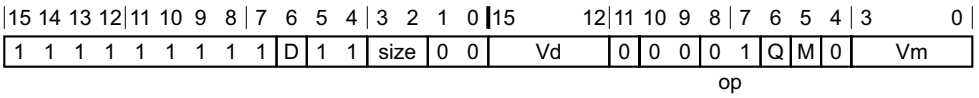
```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 `VREV` instruction must be unconditional. ARM strongly recommends that a T32 `VREV` instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| 8 | when `size = 00` |
| 16 | when `size = 01` |

The encoding `size = 1x` is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    bits(64) result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
integer element;
integer rev_element;
for r = 0 to regs-1
    element = 0;
    for c = 0 to containers-1
        rev_element = element + elements_per_container - 1;
        for e = 0 to elements_per_container-1
            Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
            element = element + 1;
            rev_element = rev_element - 1;
    D[d+r] = result;
```

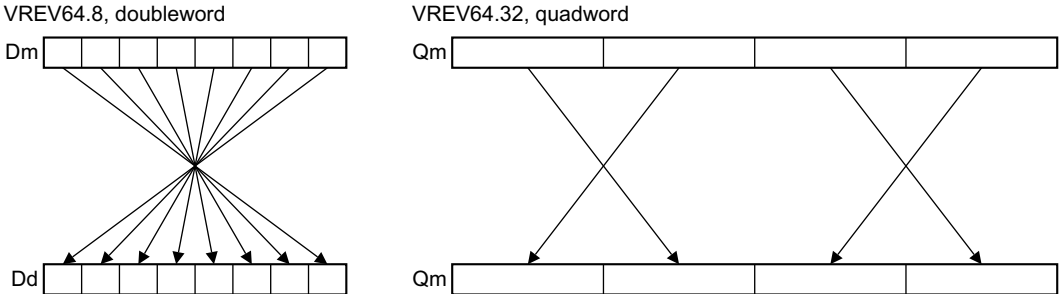

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.177 VREV64

Vector Reverse in doublewords reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

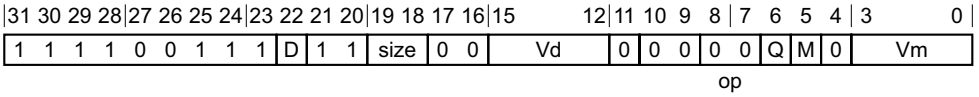
The following figure shows an example of the operation of VREV64 doubleword operations.




Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

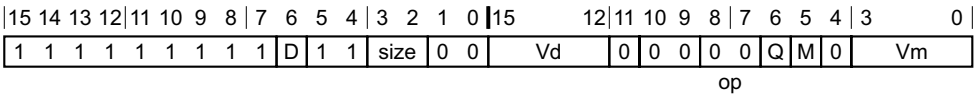
```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 `VREV` instruction must be unconditional. ARM strongly recommends that a T32 `VREV` instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| `8` | when `size = 00` |
| `16` | when `size = 01` |
| `32` | when `size = 10` |

The encoding `size = 11` is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(64) result;
integer element;
integer rev_element;
for r = 0 to regs-1
    element = 0;
    for c = 0 to containers-1
        rev_element = element + elements_per_container - 1;
        for e = 0 to elements_per_container-1
            Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
            element = element + 1;
            rev_element = rev_element - 1;
    D[d+r] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.178 VRHADD

Vector Rounding Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector.

The operand and result elements are all the same type, and can be any one of:

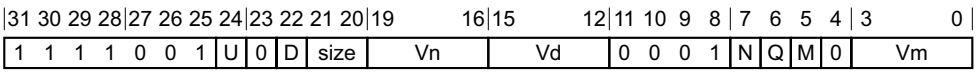
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The results of the halving operations are rounded. For truncated results, see VHADD.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

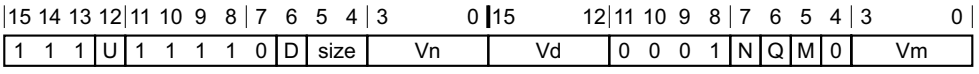
Applies when `Q = 1`.

```
VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VRHADD instruction must be unconditional. ARM strongly recommends that a T32 VRHADD instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |

The following encodings are reserved:

- U = 0, size = 11.
- U = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

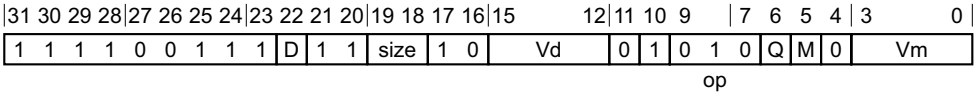
```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = op1 + op2 + 1;
            Elem[D[d+r],e,esize] = result<esize:1>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.179 VRINTA (Advanced SIMD)

Vector Round floating-point to integer towards Nearest with Ties to Away rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTA{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

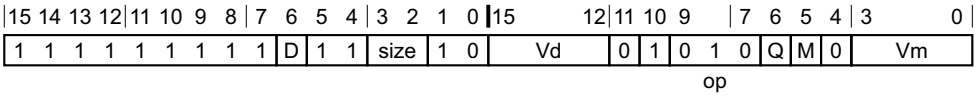
Applies when `Q = 1`.

```
VRINTA{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTA{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRINTA{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

Related encodings:

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qm> | Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

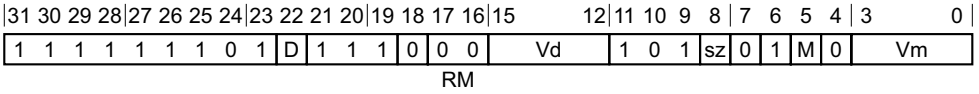
```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.180 VRINTA (floating-point)

Round floating-point to integer to Nearest with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTA{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

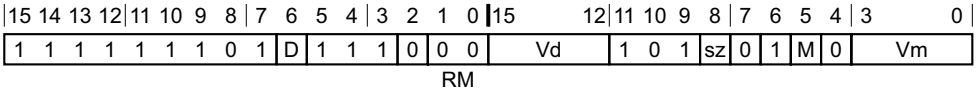
Applies when `sz = 1`.

```
VRINTA{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  exact = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTA{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VRINTA{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  exact = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

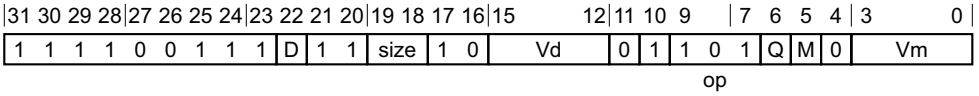
```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if dp_operation then
    D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
else
    S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.181 VRINTM (Advanced SIMD)

Vector Round floating-point to integer towards -Infinity rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTM{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

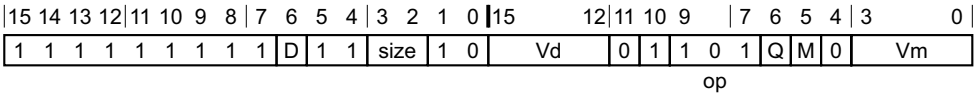
Applies when `Q = 1`.

```
VRINTM{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTM{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRINTM{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

Related encodings:

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qm> | Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

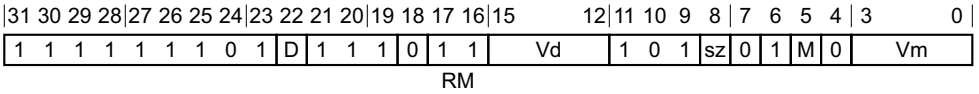
```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.182 VRINTM (floating-point)

Round floating-point to integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTM{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

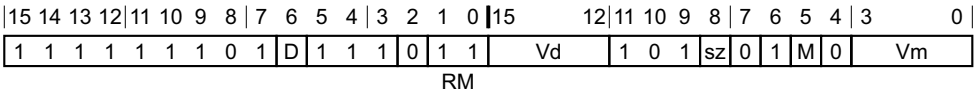
Applies when `sz = 1`.

```
VRINTM{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  exact = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTM{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VRINTM{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  exact = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

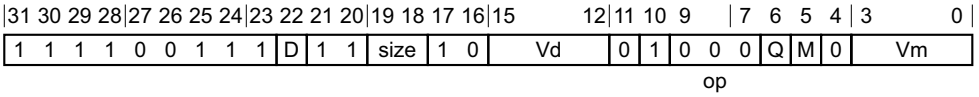
```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if dp_operation then
    D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
else
    S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.183 VRINTN (Advanced SIMD)

Vector Round floating-point to integer to Nearest rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTN{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

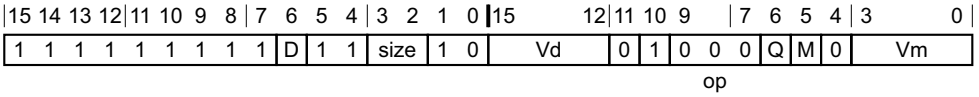
Applies when `Q = 1`.

```
VRINTN{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTN{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRINTN{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

Related encodings:

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qm> | Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

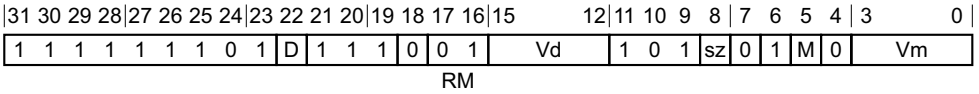
```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.184 VRINTN (floating-point)

Round floating-point to integer to Nearest rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTN{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

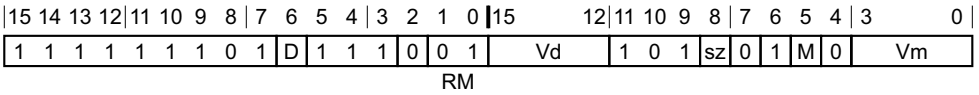
Applies when `sz = 1`.

```
VRINTN{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  exact = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTN{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VRINTN{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  exact = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

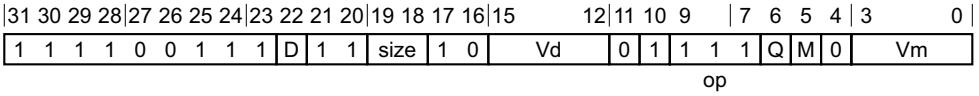
```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if dp_operation then
    D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
else
    S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.185 VRINTP (Advanced SIMD)

Vector Round floating-point to integer towards +Infinity rounds a vector of floating-point values to integral floating-point values of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTP{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

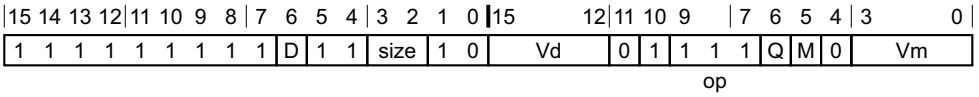
Applies when `Q = 1`.

```
VRINTP{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTP{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRINTP{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

Related encodings:

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qm> | Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

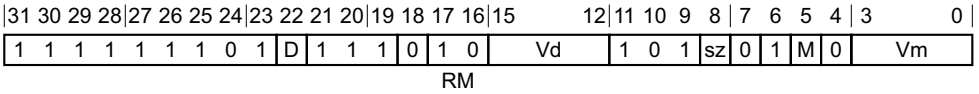
```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.186 VRINTP (floating-point)

Round floating-point to integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTP{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

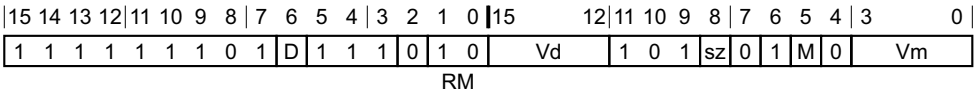
Applies when `sz = 1`.

```
VRINTP{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  exact = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTP{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VRINTP{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = FPDecodeRM(RM);  exact = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
if InITBlock() then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

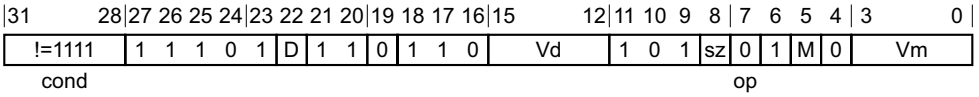
```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if dp_operation then
    D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
else
    S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.187 VRINTR

Round floating-point to integer rounds a floating-point value to an integral floating-point value of the same size using the rounding mode specified in the FPSCR. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTR{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

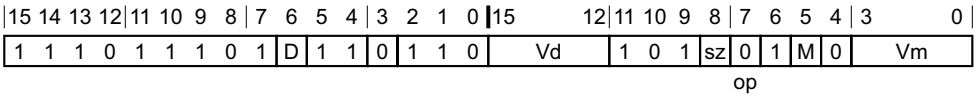
Applies when `sz = 1`.

```
VRINTR{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
dp_operation = (sz == '1');  exact = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTR{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VRINTR{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
dp_operation = (sz == '1');  exact = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

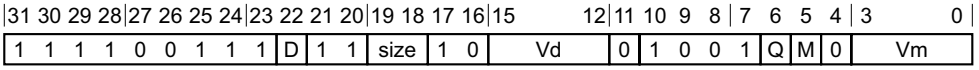
```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    if dp_operation then
        D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
    else
        S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.188 VRINTX (Advanced SIMD)

Vector round floating-point to integer inexact rounds a vector of floating-point values to integral floating-point values of the same size, using the Round to Nearest rounding mode, and raises the Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTX{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

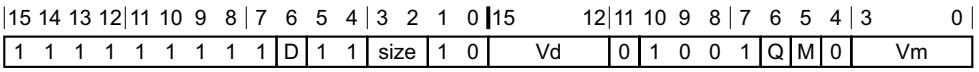
Applies when `Q = 1`.

```
VRINTX{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPRounding_TIEEVEN;  exact = TRUE;
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTX{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRINTX{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPRounding_TIEEVEN;  exact = TRUE;
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qm> | Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

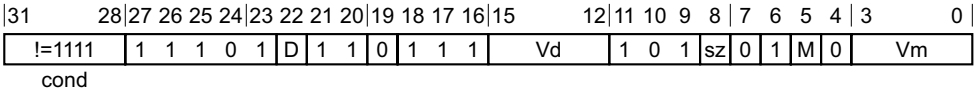
```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.189 VRINTX (floating-point)

Round floating-point to integer inexact rounds a floating-point value to an integral floating-point value of the same size, using the rounding mode specified in the FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTX{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

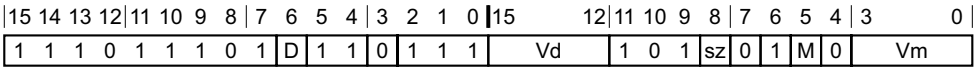
Applies when `sz = 1`.

```
VRINTX{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
exact = TRUE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTX{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VRINTX{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
exact = TRUE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

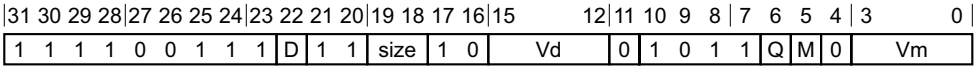
```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    rounding = FPRoundingMode(FPSCR);
    if dp_operation then
        D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
    else
        S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.190 VRINTZ (Advanced SIMD)

Vector round floating-point to integer towards Zero rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTZ{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

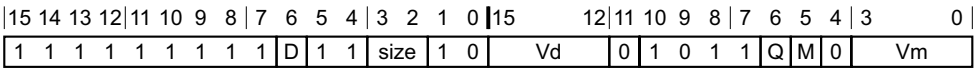
Applies when `Q = 1`.

```
VRINTZ{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPRounding_ZERO;  exact = FALSE;
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRINTZ{<q>}.F32.F32 <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRINTZ{<q>}.F32.F32 <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
rounding = FPRounding_ZERO;  exact = FALSE;
esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler symbols

| | |
|---|---|
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qm> | Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field. |

### Operation for all encodings

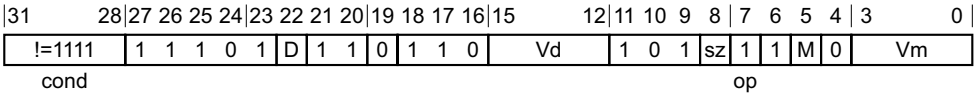
```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.191 VRINTZ (floating-point)

Round floating-point to integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTZ{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

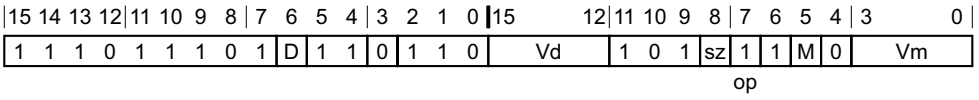
Applies when `sz = 1`.

```
VRINTZ{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
dp_operation = (sz == '1');  exact = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VRINTZ{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VRINTZ{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR);
dp_operation = (sz == '1');  exact = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    if dp_operation then
        D[d] = FPRoundInt(D[m], FPSCR, rounding, exact);
    else
        S[d] = FPRoundInt(S[m], FPSCR, rounding, exact);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.192 VRSHL

Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see VSHL.

The first operand and result elements are the same data type, and can be any one of:

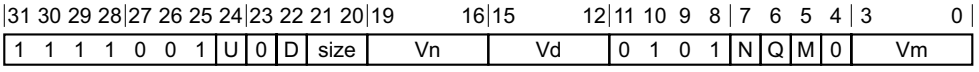
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when Q = 0.

```
VRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

#### *128-bit SIMD vector variant*

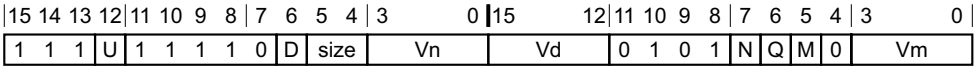
Applies when Q = 1.

```
VRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when Q = 0.

```
VRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

#### *128-bit SIMD vector variant*

Applies when Q = 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VRSHL instruction must be unconditional. ARM strongly recommends that a T32 VRSHL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| S64 | when U = 0, size = 11 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |
| U64 | when U = 1, size = 11 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            round_const = 1 << (-shift-1); // 0 for left shift, 2^(n-1) for right shift
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.193 VRSHR

Vector Rounding Shift Right takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see VSHR.

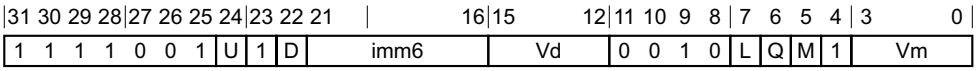
The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

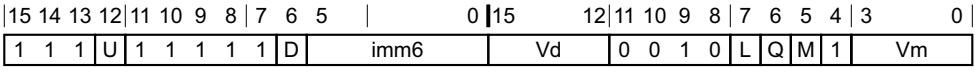
Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

### Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VRSHR instruction must be unconditional. ARM strongly recommends that a T32 VRSHR instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

- S when U = 0
- U when U = 1

<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

- 8 when L = 0, imm6<5:3> = 001
- 16 when L = 0, imm6<5:3> = 01x
- 32 when L = 0, imm6<5:3> = 1xx
- 64 when L = 1, imm6<5:3> = xxx

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

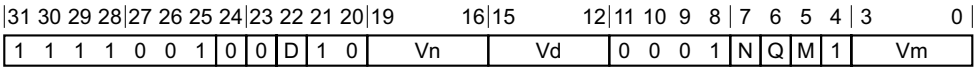
## F7.1.194 VRSHR (zero)

Vector Rounding Shift Right copies the contents of one SIMD register to another

This instruction is an alias of the VORR (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (register).
- The description of VORR (register) gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

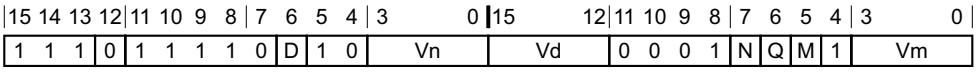
```
VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is never the preferred disassembly.

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

`VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>`

and is never the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VMOV` (register) instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VMOV` (register) instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

### Operation for all encodings

The description of VORR (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.195 VRSHRN

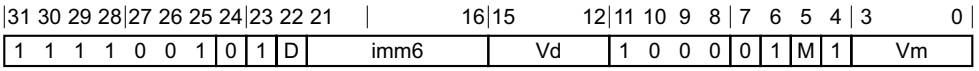
Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see VSHRN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *A1 variant*

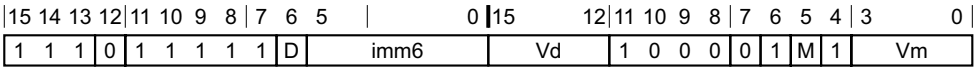
Applies when `imm6 != 000xxx`.

```
VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

#### *Decode for this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when "001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "01xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "1xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

Applies when `imm6 != 000xxx`.

```
VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

#### *Decode for this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when "001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "01xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "1xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VRSHRN instruction must be unconditional. ARM strongly recommends that a T32 VRSHRN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:

- 16 when imm6<5:3> = 001
- 32 when imm6<5:3> = 01x
- 64 when imm6<5:3> = 1xx

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount-1);
    for e = 0 to elements-1
        result = LSR(Elem[Qin[m>>1],e,2*esize] + round_const, shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

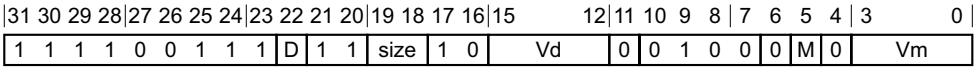
## F7.1.196 VRSHRN (zero)

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector

This instruction is an alias of the VMOVN instruction. This means that:

- The encodings in this description are named to match the encodings of VMOVN.
- The description of VMOVN gives the operational pseudocode for this instruction.

### A1




#### *A1 variant*

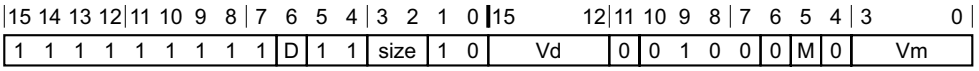
```
VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

### T1




#### *T1 variant*

```
VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VMOVN` instruction must be unconditional. ARM strongly recommends that a T32 `VMOVN` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

`I16` when `size = 00`

`I32` when `size = 01`

`I64` when `size = 10`

The encoding `size = 11` is reserved.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

The description of VMOVN gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.197 VRSQRTE

Vector Reciprocal Square Root Estimate finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

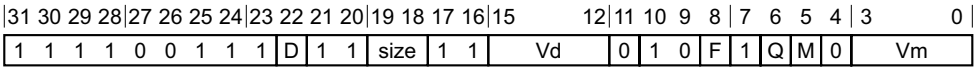
The operand and result elements are the same type, and can be 32-bit floating-point numbers, or 32-bit unsigned integers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step* on page E1-2344.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

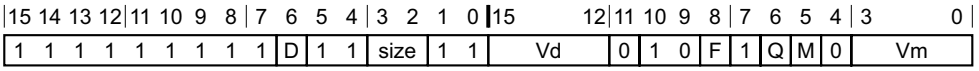
Applies when `Q = 1`.

```
VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
floating_point = (F == '1');  esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
floating_point = (F == '1');  esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VRSQRTE instruction must be unconditional. ARM strongly recommends that a T32 VRSQRTE instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:

S32 when F = 0, size = 10

F32 when F = 1, size = 10

The following encodings are reserved:

- F = 0, size = 0x.
- F = 1, size = 11.
- F = 1, size = 0x.
- F = 1, size = 11.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see *Floating-point reciprocal estimate and step* on page E1-2344.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,32] = FPRSqrtEstimate(Elem[D[m+r],e,32], StandardFPSCRValue());
            else
                Elem[D[d+r],e,32] = UnsignedRSqrtEstimate(Elem[D[m+r],e,32]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.198 VRSQRTS

Vector Reciprocal Square Root Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 3.0, divides these results by 2.0, and places the results into the elements of the destination vector.

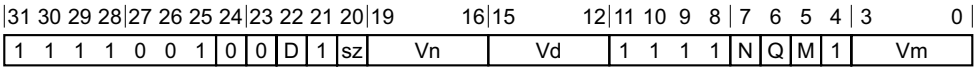
The operand and result elements are 32-bit floating-point numbers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step* on page E1-2344.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRSQRTS{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

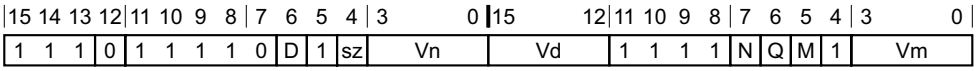
Applies when `Q = 1`.

```
VRSQRTS{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VRSQRTS{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VRSQRTS{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. An A32 VRSQRTS instruction must be unconditional. ARM strongly recommends that a T32 VRSQRTS instruction is unconditional, see *Conditional execution* on page F2-2447. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |

## Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see *Floating-point reciprocal estimate and step* on page E1-2344.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,32] = FPRSqrtStep(Elem[D[n+r],e,32], Elem[D[m+r],e,32]);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.199 VRSRA

Vector Rounding Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the rounded results into the destination vector.For truncated results, see VSRA.

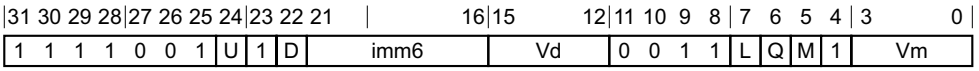
The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VRSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

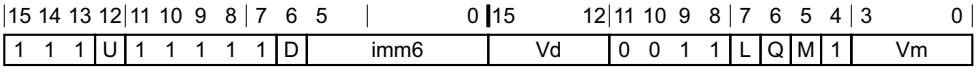
Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.

```
VRSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VRSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VRSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

### Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VRSRA instruction must be unconditional. ARM strongly recommends that a T32 VRSRA instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

- S when U = 0
- U when U = 1

<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

- 8 when L = 0, imm6<5:3> = 001
- 16 when L = 0, imm6<5:3> = 01x
- 32 when L = 0, imm6<5:3> = 1xx
- 64 when L = 1, imm6<5:3> = xxx

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.200 VRSUBHN

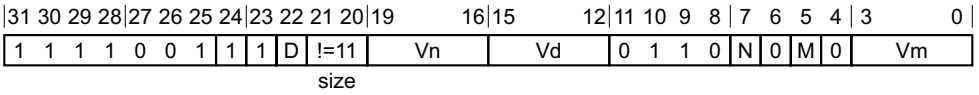
Vector Rounding Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are rounded. For truncated results, see VSUBHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



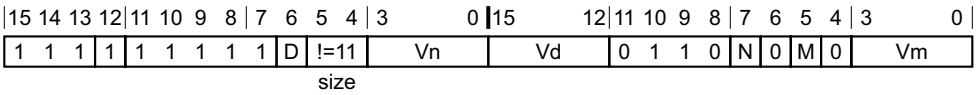

#### *A1 variant*

```
VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VRSUBHN instruction must be unconditional. ARM strongly recommends that a T32 VRSUBHN instruction is unconditional, see *Conditional execution* on page F2-2447.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

I16 when size = 00

I32 when size = 01

I64 when size = 10

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

## Operation for all encodings

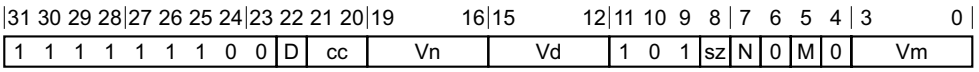
```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.201 VSELEQ, VSELGE, VSELGT, VSELVS

Floating-point conditional select allows the destination register to take the value in either one or the other source register according to the condition codes in the APSR.

### A1




#### *VSELEQ,doubleprec variant*

Applies when `cc = 00 && sz = 1`.

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
```

#### *VSELEQ,singleprec variant*

Applies when `cc = 00 && sz = 0`.

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

#### *VSELGE,doubleprec variant*

Applies when `cc = 10 && sz = 1`.

```
VSELGE.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
```

#### *VSELGE,singleprec variant*

Applies when `cc = 10 && sz = 0`.

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

#### *VSELGT,doubleprec variant*

Applies when `cc = 11 && sz = 1`.

```
VSELGT.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
```

#### *VSELGT,singleprec variant*

Applies when `cc = 11 && sz = 0`.

```
VSELGT.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

#### *VSELVS,doubleprec variant*

Applies when `cc = 01 && sz = 1`.

```
VSELVS.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
```

#### *VSELVS,singleprec variant*

Applies when `cc = 01 && sz = 0`.

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
```

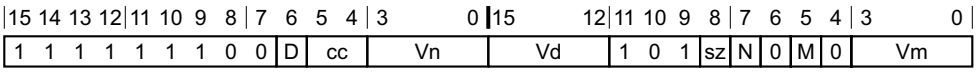
#### *Decode for all variants of this encoding*

```
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
cond = cc:(cc<1> EOR cc<0>):'0';
if InITBlock() then UNPREDICTABLE;
```

### T1




#### *VSELEQ,doubleprec variant*

Applies when cc = 00 && sz = 1.

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

#### *VSELEQ,singleprec variant*

Applies when cc = 00 && sz = 0.

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

#### *VSELGE,doubleprec variant*

Applies when cc = 10 && sz = 1.

```
VSELGE.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

#### *VSELGE,singleprec variant*

Applies when cc = 10 && sz = 0.

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

#### *VSELGT,doubleprec variant*

Applies when cc = 11 && sz = 1.

```
VSELGT.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

#### *VSELGT,singleprec variant*

Applies when cc = 11 && sz = 0.

```
VSELGT.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

#### *VSELVS,doubleprec variant*

Applies when cc = 01 && sz = 1.

```
VSELVS.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

#### *VSELVS,singleprec variant*

Applies when cc = 01 && sz = 0.

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

#### *Decode for all variants of this encoding*

```
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
cond = cc:(cc<1> EOR cc<0>):'0';
if InITBlock() then UNPREDICTABLE;
```

### Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

### Operation for all encodings

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if dp_operation then
    D[d] = if ConditionHolds(cond) then D[n] else D[m];
else
    S[d] = if ConditionHolds(cond) then S[n] else S[m];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.202 VSHL (immediate)

Vector Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector.

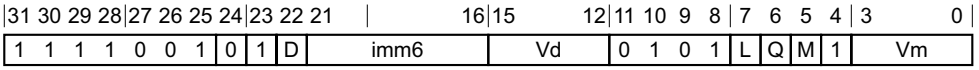
Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VSHL{<c>}{<q>}.I<size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

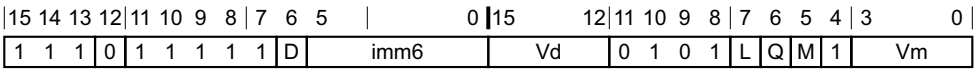
Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.

```
VSHL{<c>}{<q>}.I<size> {<Qd>,} <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VSHL{<c>}{<q>}.I<size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VSHL{<c>}{<q>}.I<size> {<Qd>,} <Qm>, #<imm>
```

### *Decode for all variants of this encoding*

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VSHL instruction must be unconditional. ARM strongly recommends that a T32 VSHL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

- 8 when L = 0, imm6<5:3> = 001
- 16 when L = 0, imm6<5:3> = 01x
- 32 when L = 0, imm6<5:3> = 1xx
- 64 when L = 1, imm6<5:3> = xxx

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = LSL(Elem[D[m+r],e,esize], shift_amount);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.203 VSHL (register)

Vector Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

For a rounding shift, see VRSHL.

The first operand and result elements are the same data type, and can be any one of:

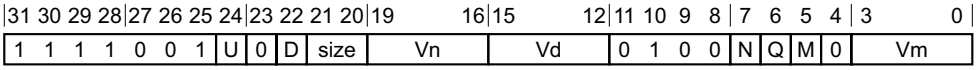
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

#### *128-bit SIMD vector variant*

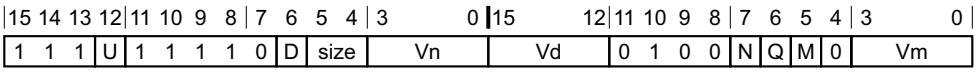
Applies when `Q = 1`.

```
VSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### 128-bit SIMD vector variant

Applies when Q = 1.

```
VSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

### Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VSHL instruction must be unconditional. ARM strongly recommends that a T32 VSHL instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| S64 | when U = 0, size = 11 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |
| U64 | when U = 1, size = 11 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            result = Int(Elem[D[m+r],e,esize], unsigned) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.204 VSHLL

Vector Shift Left Long takes each element in a doubleword vector, left shifts them by an immediate value, and places the results in a quadword vector.

The operand elements can be:

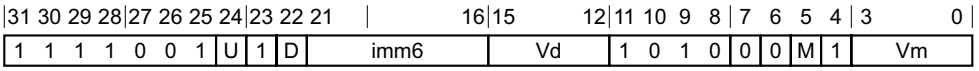
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit untyped integers, maximum shift only.

The result elements are twice the length of the operand elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *A1 variant*

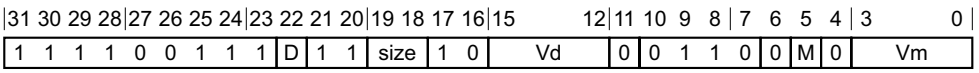
Applies when `imm6 != 000xxx`.

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

#### *Decode for this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
case imm6 of
    when "001xxx"  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when "01xxxx"  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when "1xxxxx"  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE VMOVL;
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);
```

### A2




#### *A2 variant*

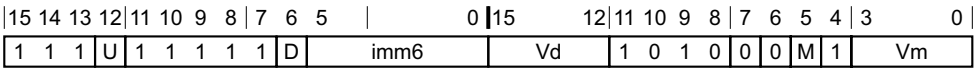
```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

#### *Decode for this encoding*

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  shift_amount = esize;
unsigned = FALSE;  // Or TRUE without change of functionality
d = UInt(D:Vd);  m = UInt(M:Vm);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *T1 variant*

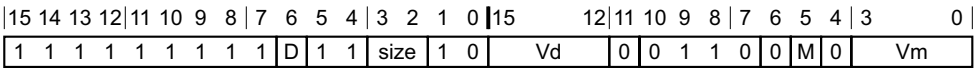
Applies when `imm6 != 000xxx`.

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

#### *Decode for this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
case imm6 of
    when "001xxx"  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when "01xxxx"  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when "1xxxxx"  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE VMOVL;
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);
```

### T2




#### *T2 variant*

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

#### *Decode for this encoding*

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  shift_amount = esize;
unsigned = FALSE;  // Or TRUE without change of functionality
d = UInt(D:Vd);  m = UInt(M:Vm);
```

## Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VSHLL` instruction must be unconditional. ARM strongly recommends that a T32 `VSHLL` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<type>` The data type for the elements of the operand. It must be one of:

- `S` Signed. In encoding T1/A1, encoded as U = 0.
- `U` Unsigned. In encoding T1/A1, encoded as U = 1.
- `I` Untyped integer, Available only in encoding T2/A2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<size> The data size for the elements of the operand. The following table shows the permitted values and their encodings:

| <size> | Encoding T1/A1 | Encoding T2/A2 |
|---|---|---|
| 8 | Encoded as imm6<5:3> = 0b001 | Encoded as size = 0b00 |
| 16 | Encoded as imm6<5:4> = 0b01 | Encoded as size = 0b01 |
| 32 | Encoded as imm6<5> = 1 | Encoded as size = 0b10 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> The immediate value. <imm> must lie in the range 1 to <size>, and:

- If <size> == <imm>, the encoding is T2/A2.
- Otherwise, the encoding is T1/A1, and:
  - If <size> == 8, <imm> is encoded in imm6<2:0>.
  - If <size> == 16, <imm> is encoded in imm6<3:0>.
  - If <size> == 32, <imm> is encoded in imm6<4:0>.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[Din[m],e,esize], unsigned) << shift_amount;
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.205 VSHR

Vector Shift Right takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see VRSHR.

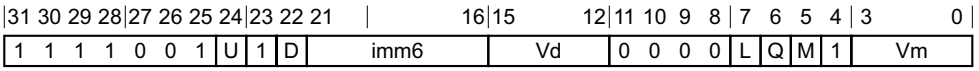
The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

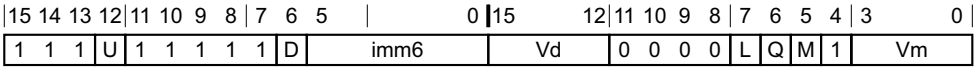
Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.

```
VSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

### Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VSHR instruction must be unconditional. ARM strongly recommends that a T32 VSHR instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

- S when U = 0
- U when U = 1

<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

- 8 when L = 0, imm6<5:3> = 001
- 16 when L = 0, imm6<5:3> = 01x
- 32 when L = 0, imm6<5:3> = 1xx
- 64 when L = 1, imm6<5:3> = xxx

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

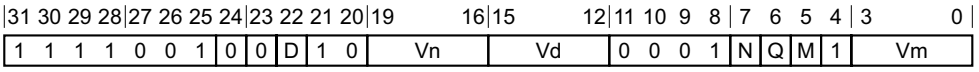
## F7.1.206 VSHR (zero)

Vector Shift Right copies the contents of one SIMD register to another

This instruction is an alias of the VORR (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (register).
- The description of VORR (register) gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

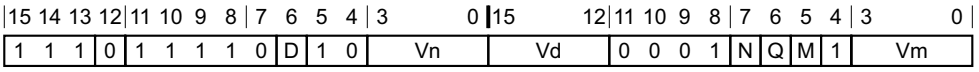
```
VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is never the preferred disassembly.

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is never the preferred disassembly.

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

is equivalent to

`VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>`

and is never the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VMOV` (register) instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VMOV` (register) instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

### Operation for all encodings

The description of VORR (register) gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.207 VSHRN

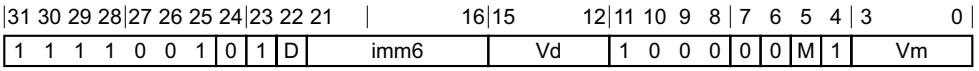
Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see VRSHRN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *A1 variant*

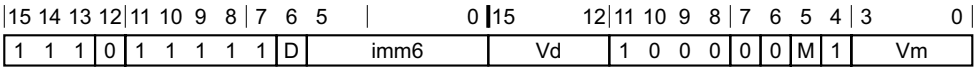
Applies when `imm6 != 000xxx`.

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

#### *Decode for this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when "001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "01xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "1xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

Applies when `imm6 != 000xxx`.

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

#### *Decode for this encoding*

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when "001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "01xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "1xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VSHRN instruction must be unconditional. ARM strongly recommends that a T32 VSHRN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:

- 16 when imm6<5:3> = 001
- 32 when imm6<5:3> = 01x
- 64 when imm6<5:3> = 1xx

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = LSR(Elem[Qin[m>>1],e,2*esize], shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

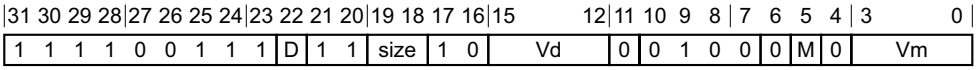
## F7.1.208 VSHRN (zero)

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector

This instruction is an alias of the VMOVN instruction. This means that:

- The encodings in this description are named to match the encodings of VMOVN.
- The description of VMOVN gives the operational pseudocode for this instruction.

### A1




#### *A1 variant*

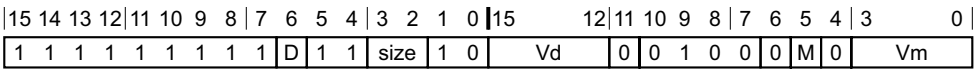
```
VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

### T1




#### *T1 variant*

```
VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

and is never the preferred disassembly.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VMOVN instruction must be unconditional. ARM strongly recommends that a T32 VMOVN instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

- I16 when size = 00
- I32 when size = 01
- I64 when size = 10

The encoding size = 11 is reserved.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation for all encodings

The description of VMOVN gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.209 VSLI

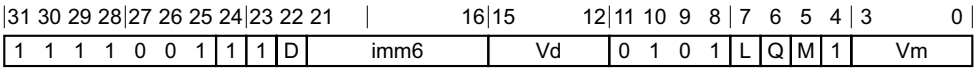
Vector Shift Left and Insert takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VSLI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

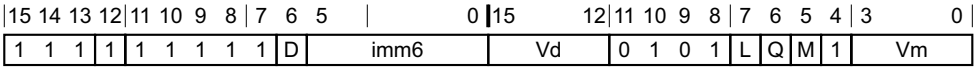
Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.

```
VSLI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VSLI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.

```
VSLI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VSLI instruction must be unconditional. ARM strongly recommends that a T32 VSLI instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

- 8 when L = 0, imm6<5:3> = 001
- 16 when L = 0, imm6<5:3> = 01x
- 32 when L = 0, imm6<5:3> = 1xx
- 64 when L = 1, imm6<5:3> = xxx

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    mask = LSL(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSL(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

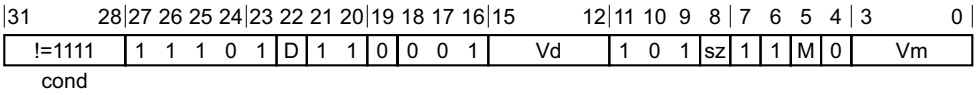

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.210 VSQRT

Square Root calculates the square root of the value in a floating-point register and writes the result to another floating-point register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

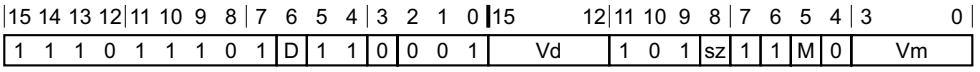
Applies when `sz = 1`.

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T1




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446.

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        D[d] = FPSqrt(D[m], FPSCR);
    else
        S[d] = FPSqrt(S[m], FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.211 VSRA

Vector Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the truncated results into the destination vector. For rounded results, see VRSRA.

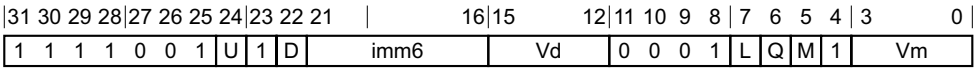
The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

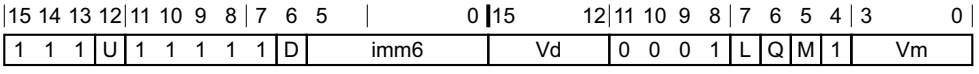
Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.

```
VSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

### Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VSRA instruction must be unconditional. ARM strongly recommends that a T32 VSRA instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<type> Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

- S when U = 0
- U when U = 1

<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

- 8 when L = 0, imm6<5:3> = 001
- 16 when L = 0, imm6<5:3> = 01x
- 32 when L = 0, imm6<5:3> = 1xx
- 64 when L = 1, imm6<5:3> = xxx

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.212 VSRI

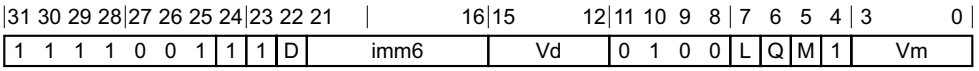
Vector Shift Right and Insert takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VSRI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

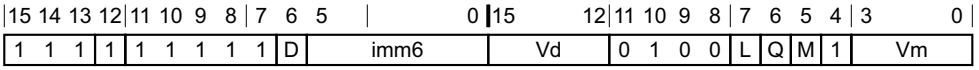
Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.

```
VSRI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```

#### *Decode for all variants of this encoding*

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 0`.

```
VSRI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

#### *128-bit SIMD vector variant*

Applies when `!(imm6 == 000xxx && L == 0) && Q = 1`.

```
VSRI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when "0001xxx"  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when "001xxxx"  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when "01xxxxx"  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when "1xxxxxx"  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Notes for all encodings

Related encodings: *One register and a modified immediate value* on page F5-2562.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VSRI instruction must be unconditional. ARM strongly recommends that a T32 VSRI instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

- 8 when L = 0, imm6<5:3> = 001
- 16 when L = 0, imm6<5:3> = 01x
- 32 when L = 0, imm6<5:3> = 1xx
- 64 when L = 1, imm6<5:3> = xxx

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    mask = LSR(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSR(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

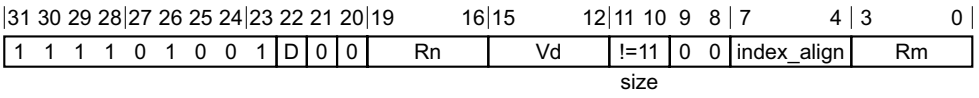
## F7.1.213 VST1 (single element from one lane)

Store single element from one lane of one register stores one element to memory from one element of a register. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

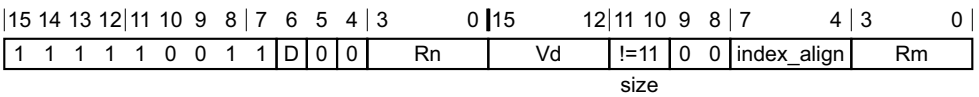
Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
    when '01'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 2;  index = UInt(index_align<3:2>);
        alignment = if index_align<0> == '0' then 1 else 2;
    when '10'
        if index_align<2> != '0' then UNDEFINED;
        if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when Rm = 1111.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when Rm = 1101.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when Rm != 11x1.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
    when '01'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 2;  index = UInt(index_align<3:2>);
        alignment = if index_align<0> == '0' then 1 else 2;
    when '10'
        if index_align<2> != '0' then UNDEFINED;
        if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> Is the data size, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| 8 | when size = 00 |
| 16 | when size = 01 |
| 32 | when size = 10 |

<list> Is a list containing the single 64-bit name of the SIMD&FP register holding the element. The list must be { <Dd>[<index>] }. The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

| | |
|---|---|
| <size> == 8 | <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field. |
| <size> == 16 | <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field. |
| <size> == 32 | <index> is 0 or 1, encoded in the "index_align<3>" field. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

`<align>` When `<size>` == 8, `<align>` must be omitted, otherwise it is the optional alignment. Whenever `<align>` is omitted, the standard alignment is used, see , and the encoding depends on `<size>`:

- `<size> == 8` Encoded in the "index_align<0>" field as `0`.
- `<size> == 16` Encoded in the "index_align<1:0>" field as `0b00`.
- `<size> == 32` Encoded in the "index_align<2:0>" field as `0b000`.

Whenever `<align>` is present, the permitted values and encoding depend on `<size>`:

- `<size> == 16` `<align>` is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as `0b01`.
- `<size> == 32` `<align>` is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as `0b011`.

`:` is the preferred separator before the `<align>` value, but the alignment can be specified as `@<align>`, see *Advanced SIMD addressing mode* on page F5-2571.

`<Rm>` Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
    MemU[address,ebytes] = Elem[D[d],index];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

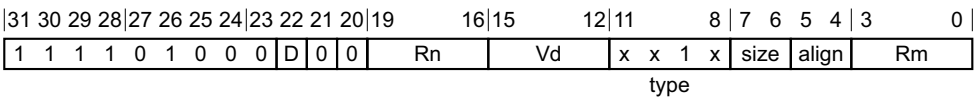
### F7.1.214 VST1 (multiple single elements)

Store multiple single elements from one, two, three, or four registers stores elements to memory from one, two, three, or four registers, without interleaving. Every element of each register is stored. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

#### A1




##### *Offset variant*

Applies when `Rm = 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

##### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

##### *Post-indexed variant*

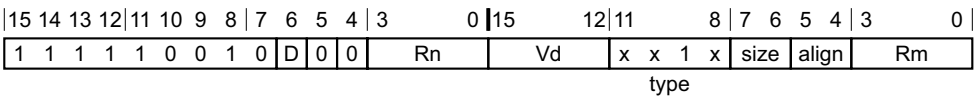
Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

##### *Decode for all variants of this encoding*

```
case type of
    when '0111'
        regs = 1;  if align<1> == '1' then UNDEFINED;
    when '1010'
        regs = 2;  if align == '11' then UNDEFINED;
    when '0110'
        regs = 3;  if align<1> == '1' then UNDEFINED;
    when '0010'
        regs = 4;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

#### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
case type of
    when '0111'
        regs = 1;  if align<1> == '1' then UNDEFINED;
    when '1010'
        regs = 2;  if align == '11' then UNDEFINED;
    when '0110'
        regs = 3;  if align<1> == '1' then UNDEFINED;
    when '0010'
        regs = 4;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VST1 (multiple single elements)* on page K1-5492.

Related encodings: Advanced SIMD element or structure load/store instructions.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| `8` | when `size = 00` |
| `16` | when `size = 01` |
| `32` | when `size = 10` |
| `64` | when `size = 11` |

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

| | |
|---|---|
| `{ <Dd> }` | Encoded in the "type" field as `0b0111`. |
| `{ <Dd>, <Dd+1> }` | Encoded in the "type" field as `0b1010`. |
| `{ <Dd>, <Dd+1>, <Dd+2> }` | Encoded in the "type" field as `0b0110`. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }Encoded in the "type" field as 0b0010.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as 0b01.

128 128-bit alignment, encoded in the "align" field as 0b10. Available only if <list> contains two or four registers.

256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 8*regs;
    for r = 0 to regs-1
        for e = 0 to elements-1
            if ebytes != 8 then
                MemU[address,ebytes] = Elem[D[d+r],e];
            else
                bits(64) data = Elem[D[d+r],e];
                MemU[address,4] = if BigEndian() then data<63:32> else data<31:0>;
                MemU[address+4,4] = if BigEndian() then data<31:0> else data<63:32>;
            address = address + ebytes;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

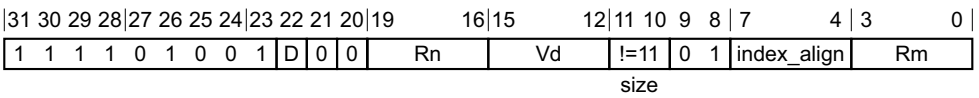
### F7.1.215 VST2 (single 2-element structure from one lane)

Store single 2-element structure from one lane of two registers stores one 2-element structure to memory from corresponding elements of two registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

#### A1




##### *Offset variant*

Applies when `Rm = 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

##### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

##### *Post-indexed variant*

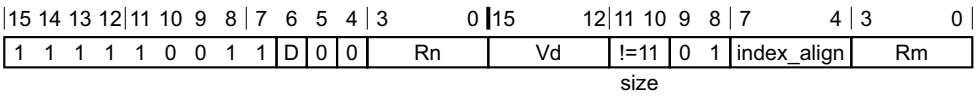
Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

##### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 2;
    when '01'
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '10'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

#### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 2;
    when '01'
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '10'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VST2 (single 2-element structure from one lane)* on page K1-5493.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

`<list>` Is a list containing the 64-bit names of the two SIMD&FP registers holding the element. The list must be one of:

- `{ <Dd>[<index>], <Dd+1>[<index>] }` Single-spaced registers, encoded as "spacing" = `0`.
- `{ <Dd>[<index>], <Dd+2>[<index>] }` Double-spaced registers, encoded as "spacing" = `1`. Not permitted when `<size>` == 8.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding of "spacing" depends on <size>:

<size> == 16 "spacing" is encoded in the "index_align<1>" field.

<size> == 32 "spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8 <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16 <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32 <index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and the encoding depends on <size>:

<size> == 8 Encoded in the "index_align<0>" field as 0.

<size> == 16 Encoded in the "index_align<0>" field as 0.

<size> == 32 Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8 <align> is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16 <align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32 <align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
    MemU[address,       ebytes] = Elem[D[d], index];
    MemU[address+ebytes,ebytes] = Elem[D[d2],index];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

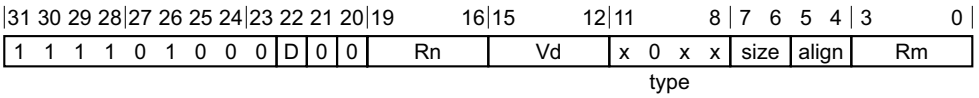
### F7.1.216 VST2 (multiple 2-element structures)

Store multiple 2-element structures from two or four registers stores multiple 2-element structures from two or four registers to memory, with interleaving. For more information, see *Element and structure load/store instructions* on page F1-2428. Every element of each register is saved. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

#### A1




##### *Offset variant*

Applies when `Rm = 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

##### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

##### *Post-indexed variant*

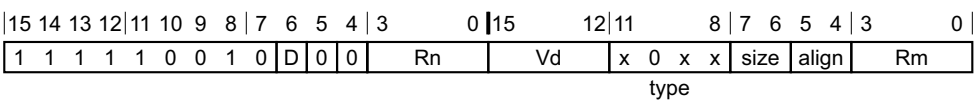
Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

##### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case type of
    when '1000'
        regs = 1;  inc = 1;  if align == '11' then UNDEFINED;
    when '1001'
        regs = 1;  inc = 2;  if align == '11' then UNDEFINED;
    when '0011'
        regs = 2;  inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```

#### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case type of
    when '1000'
        regs = 1;  inc = 1;  if align == '11' then UNDEFINED;
    when '1001'
        regs = 1;  inc = 2;  if align == '11' then UNDEFINED;
    when '0011'
        regs = 2;  inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VST2 (multiple 2-element structures)* on page K1-5493.

Related encodings: Advanced SIMD element or structure load/store instructions.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

- `{ <Dd>, <Dd+1> }` Single-spaced registers, encoded in the "type" field as `0b1000`.
- `{ <Dd>, <Dd+2> }` Double-spaced registers, encoded in the "type" field as `0b1001`.
- `{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Single-spaced registers, encoded in the "type" field as `0b0011`.

The register `<Dd>` is encoded in the "D:Vd" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as 0b01.

128 128-bit alignment, encoded in the "align" field as 0b10.

256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 16*regs;
    for r = 0 to regs-1
        for e = 0 to elements-1
            MemU[address,       ebytes] = Elem[D[d+r], e];
            MemU[address+ebytes,ebytes] = Elem[D[d2+r],e];
            address = address + 2*ebytes;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

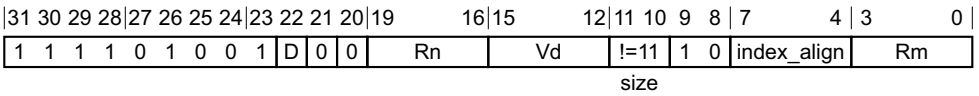
## F7.1.217 VST3 (single 3-element structure from one lane)

Store single 3-element structure from one lane of three registers stores one 3-element structure to memory from corresponding elements of three registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

#### *Post-indexed variant*

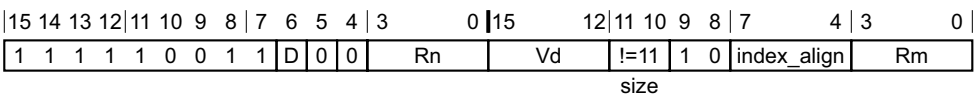
Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
    when '01'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
    when '10'
        if index_align<1:0> != '00' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
    when '01'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
    when '10'
        if index_align<1:0> != '00' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VST3 (single 3-element structure from one lane)* on page K1-5494.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

`<list>` Is a list containing the 64-bit names of the three SIMD&FP registers holding the element. The list must be one of:

- `{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] }` Single-spaced registers, encoded as "spacing" = `0`.
- `{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] }` Double-spaced registers, encoded as "spacing" = 1. Not permitted when `<size>` == 8.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding of "spacing" depends on <size>:

<size> == 8 — "spacing" is encoded in the "index_align<0>" field.

<size> == 16 — "spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.

<size> == 32 — "spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8 — <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16 — <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32 — <index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Alignment

Standard alignment rules apply, see *Alignment support* on page B2-75.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
    MemU[address,         ebytes] = Elem[D[d], index];
    MemU[address+ebytes,  ebytes] = Elem[D[d2],index];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

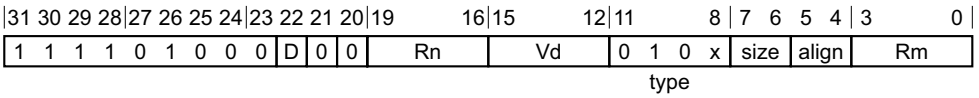
## F7.1.218 VST3 (multiple 3-element structures)

Store multiple 3-element structures from three registers stores multiple 3-element structures to memory from three registers, with interleaving. For more information, see *Element and structure load/store instructions* on page F1-2428. Every element of each register is saved. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

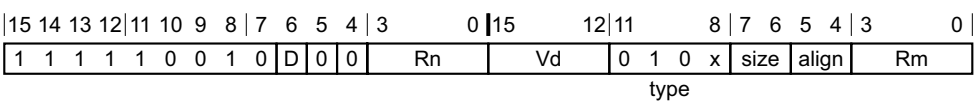
Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || align<1> == '1' then UNDEFINED;
case type of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || align<1> == '1' then UNDEFINED;
case type of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VST3 (multiple 3-element structures)* on page K1-5494.

Related encodings: Advanced SIMD element or structure load/store instructions.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| `8` | when `size = 00` |
| `16` | when `size = 01` |
| `32` | when `size = 10` |

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2> }` Single-spaced registers, encoded in the "type" field as `0b0100`.

`{ <Dd>, <Dd+2>, <Dd+4> }` Double-spaced registers, encoded in the "type" field as `0b0101`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and is encoded in the "align" field as 0b00. Whenever <align> is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as 0b01. : is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
    for e = 0 to elements-1
        MemU[address,         ebytes] = Elem[D[d], e];
        MemU[address+ebytes,  ebytes] = Elem[D[d2],e];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e];
        address = address + 3*ebytes;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

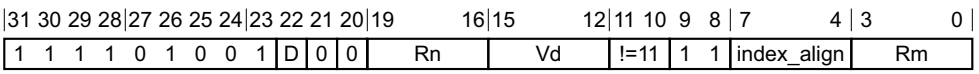
## F7.1.219 VST4 (single 4-element structure from one lane)

Store single 4-element structure from one lane of four registers stores one 4-element structure to memory from corresponding elements of four registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

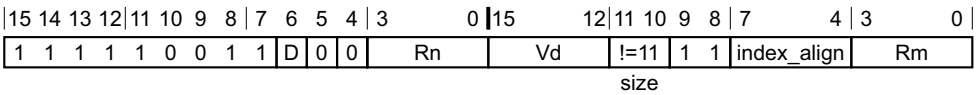
Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '01'
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
    when '10'
        if index_align<1:0> == '11' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '01'
        ebytes = 2;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
    when '10'
        if index_align<1:0> == '11' then UNDEFINED;
        ebytes = 4;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VST4 (single 4-element structure from one lane)* on page K1-5495.

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

`<list>` Is a list containing the 64-bit names of the four SIMD&FP registers holding the element. The list must be one of:

- `{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] }` Single-spaced registers, encoded as "spacing" = `0`.
- `{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] }` Double-spaced registers, encoded as "spacing" = 1. Not permitted when `<size>` == 8.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The encoding of "spacing" depends on <size>:

<size> == 16 "spacing" is encoded in the "index_align<1>" field.

<size> == 32 "spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8 <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16 <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32 <index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and the encoding depends on <size>:

<size> == 8 Encoded in the "index_align<0>" field as 0.

<size> == 16 Encoded in the "index_align<0>" field as 0.

<size> == 32 Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8 <align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16 <align> is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32 <align> can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as 0b01, and 128-bit alignment is encoded in the "index_align<1:0>" field as 0b10.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
    MemU[address,         ebytes] = Elem[D[d], index];
    MemU[address+ebytes,  ebytes] = Elem[D[d2],index];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index];
    MemU[address+3*ebytes,ebytes] = Elem[D[d4],index];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

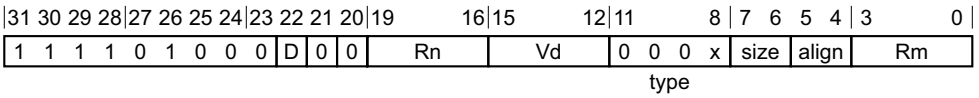
## F7.1.220 VST4 (multiple 4-element structures)

Store multiple 4-element structures from four registers stores multiple 4-element structures to memory from four registers, with interleaving. For more information, see *Element and structure load/store instructions* on page F1-2428. Every element of each register is saved. For details of the addressing mode see *Advanced SIMD addressing mode* on page F5-2571.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *Offset variant*

Applies when `Rm = 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

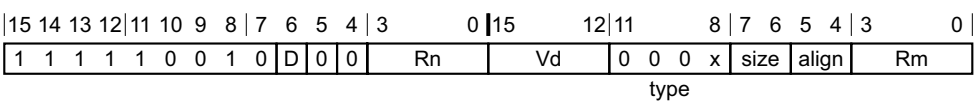
Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case type of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### T1





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Offset variant*

Applies when `Rm = 1111`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

#### *Post-indexed variant*

Applies when `Rm = 1101`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

#### *Post-indexed variant*

Applies when `Rm != 11x1`.

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
case type of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VST4 (multiple 4-element structures)* on page K1-5495.

Related encodings: Advanced SIMD element or structure load/store instructions.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. This instruction must be unconditional in A32. ARM strongly recommends that this instruction is unconditional in T32, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` Is the data size, encoded in the "size" field. It can have the following values:

- `8` when `size = 00`
- `16` when `size = 01`
- `32` when `size = 10`

The encoding `size = 11` is reserved.

`<list>` Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

`{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }` Single-spaced registers, encoded in the "type" field as `0b0000`.

`{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> }` Double-spaced registers, encoded in the "type" field as `0b0001`.

The register `<Dd>` is encoded in the "D:Vd" field.

`<Rn>` Is the general-purpose base register, encoded in the "Rn" field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see , and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as 0b01.

128 128-bit alignment, encoded in the "align" field as 0b10.

256 256-bit alignment, encoded in the "align" field as 0b11.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode* on page F5-2571.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode* on page F5-2571.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 32;
    for e = 0 to elements-1
        MemU[address,         ebytes] = Elem[D[d], e];
        MemU[address+ebytes,  ebytes] = Elem[D[d2],e];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e];
        MemU[address+3*ebytes,ebytes] = Elem[D[d4],e];
        address = address + 4*ebytes;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

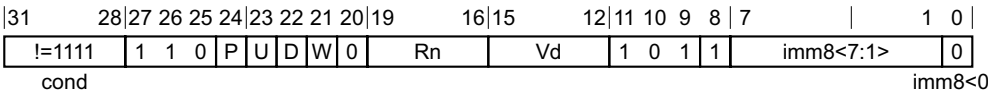
## F7.1.221 VSTM, VSTMDB, VSTMIA

Store multiple SIMD&FP registers stores multiple registers from the Advanced SIMD and floating-point register file to consecutive memory locations using an address from a general-purpose register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

This instruction is used by the alias VPUSH. See the *Alias conditions* on page F7-3713 table for details of when each alias is preferred.

### A1




#### *Decrement Before variant*

Applies when `P = 1 && U = 0 && W = 1`.

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

#### *Increment After variant*

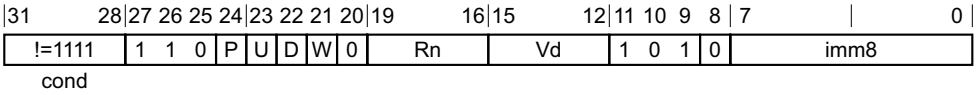
Applies when `P = 0 && U = 1`.

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

#### *Decode for all variants of this encoding*

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VSTR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

### A2




#### *Decrement Before variant*

Applies when `P = 1 && U = 0 && W = 1`.

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

#### *Increment After variant*

Applies when `P = 0 && U = 1`.

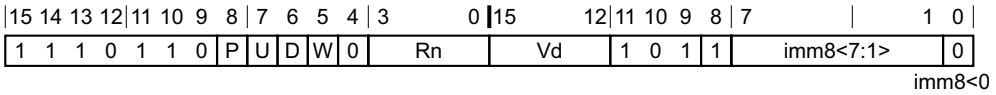
```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VSTR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

## T1




### Decrement Before variant

Applies when P = 1 && U = 0 && W = 1.

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

### Increment After variant

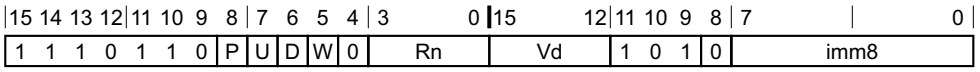
Applies when P = 0 && U = 1.

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

### Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VSTR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

## T2




### Decrement Before variant

Applies when P = 1 && U = 0 && W = 1.

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

### Increment After variant

Applies when P = 0 && U = 1.

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VSTR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VSTM* on page K1-5495.

Related encodings: 64-bit transfers between ARM core and extension registers.

## Alias conditions

| Alias | is preferred when |
|---|---|
| VPUSH | `P == '1' && U == '0' && W == '1' && Rn == '1101'` |

## Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<size>` An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

`<Rn>` For encoding A1 and A2: is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, ARM deprecates use of the PC.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

`!` Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.

`<sreglist>` Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

`<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r];  address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian() then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian() then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
```

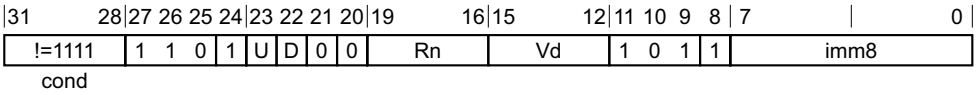

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.222 VSTR

Store SIMD&FP register stores a single register from the Advanced SIMD and floating-point register file to memory, using an address from a general-purpose register, with an optional offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

### A1



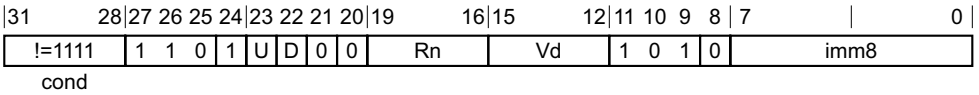

#### *A1 variant*

```
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #{+/-}<imm>}]
```

#### *Decode for this encoding*

```
single_reg = FALSE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(D:Vd);  n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
```

### A2



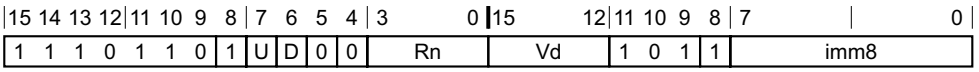

#### *A2 variant*

```
VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #{+/-}<imm>}]
```

#### *Decode for this encoding*

```
single_reg = TRUE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(Vd:D);  n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
```

### T1




#### *T1 variant*

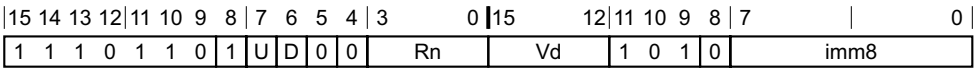
```
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #{+/-}<imm>}]
```

#### *Decode for this encoding*

```
single_reg = FALSE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(D:Vd);  n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### T2




#### *T2 variant*

```
VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #{+/-}<imm>}]
```

#### *Decode for this encoding*

```
single_reg = TRUE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(Vd:D);  n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
```

## Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

## Assembler symbols

| | |
|---|---|
| <c> | See *Standard assembler syntax fields* on page F2-2446. |
| <q> | See *Standard assembler syntax fields* on page F2-2446. |
| .64 | Optional data size specifiers. |
| <Dd> | The source register for a doubleword store. |
| .32 | Optional data size specifiers. |
| <Sd> | The source register for a singleword store. |
| <Rn> | For encoding A1 and A2: is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated.<br>For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | The immediate offset used for forming the address. Values are multiples of 4 in the range 0-1020. <imm> can be omitted, meaning an offset of +0. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    if single_reg then
        MemA[address,4] = S[d];
    else
        // Store as two word-aligned words in the correct order for current endianness.
        MemA[address,4] = if BigEndian() then D[d]<63:32> else D[d]<31:0>;
        MemA[address+4,4] = if BigEndian() then D[d]<31:0> else D[d]<63:32>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

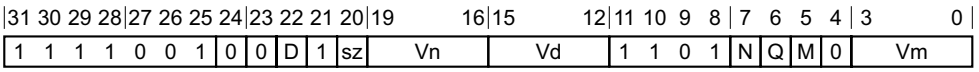
## F7.1.223 VSUB (floating-point)

Vector Subtract (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VSUB{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

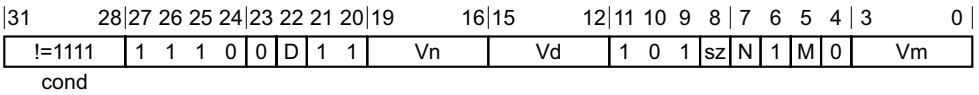
Applies when `Q = 1`.

```
VSUB{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### A2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VSUB{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

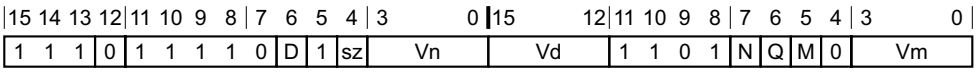
```
VSUB{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VSUB{<c>}{<q>}.F32 {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

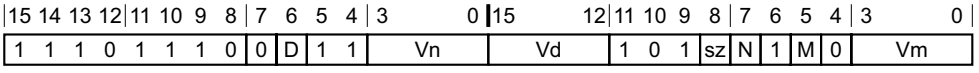
Applies when `Q = 1`.

```
VSUB{<c>}{<q>}.F32 {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T2




#### *Single-precision scalar variant*

Applies when `sz = 0`.

```
VSUB{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

#### *Double-precision scalar variant*

Applies when `sz = 1`.

```
VSUB{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

#### *Decode for all variants of this encoding*

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## Assembler symbols

<c> For encoding A1 and T1: see *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VSUB` instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VSUB` instruction is unconditional, see *Conditional execution* on page F2-2447.

For encoding A2: see *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD `VADD` instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD `VADD` instruction is unconditional, see *Conditional execution* on page F2-2447

For encoding T2: see *Standard assembler syntax fields* on page F2-2446.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
 if ConditionPassed() then
     EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
     if advsimd then  // Advanced SIMD instruction
         for r = 0 to regs-1
             for e = 0 to elements-1
                 Elem[D[d+r],e,esize] = FPSub(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize],
StandardFPSCRValue());
     else             // VFP instruction
         if dp_operation then
             D[d] = FPSub(D[n], D[m], FPSCR);
         else
             S[d] = FPSub(S[n], S[m], FPSCR);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

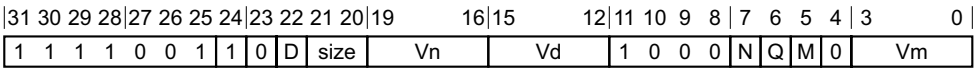
## F7.1.224 VSUB (integer)

Vector Subtract (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

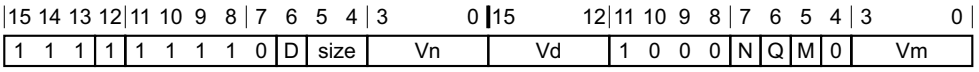
Applies when `Q = 1`.

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 Advanced SIMD VSUB instruction must be unconditional. ARM strongly recommends that a T32 Advanced SIMD VSUB instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

| | |
|---|---|
| I8 | when size = 00 |
| I16 | when size = 01 |
| I32 | when size = 10 |
| I64 | when size = 11 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] - Elem[D[m+r],e,esize];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.225 VSUBHN

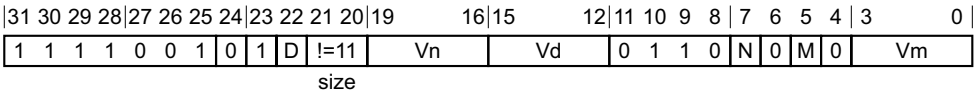
Vector Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are truncated. For rounded results, see VRSUBHN.

There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



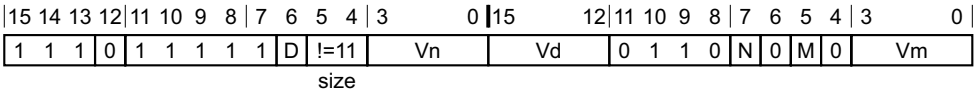

#### *A1 variant*

```
VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VSUBHN instruction must be unconditional. ARM strongly recommends that a T32 VSUBHN instruction is unconditional, see *Conditional execution* on page F2-2447.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

I16 when size = 00

I32 when size = 01

I64 when size = 10

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

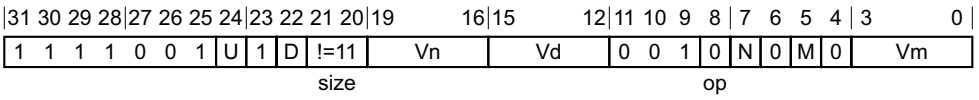
## F7.1.226 VSUBL

Vector Subtract Long subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in a quadword vector. Before subtracting, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



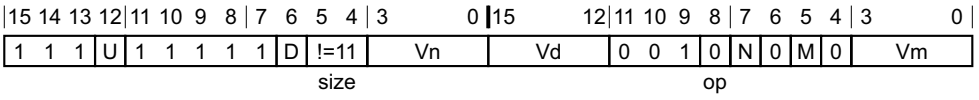

#### *A1 variant*

```
VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VSUBL or VSUBW instruction must be unconditional. ARM strongly recommends that a T32 VSUBL or VSUBW instruction is unconditional, see *Conditional execution* on page F2-2447.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vsubw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 - Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

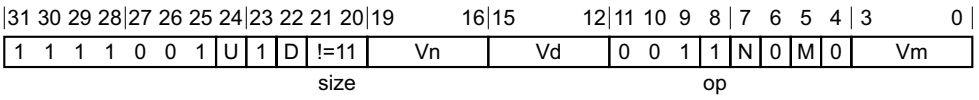
## F7.1.227 VSUBW

Vector Subtract Wide subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in another quadword vector. Before subtracting, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1



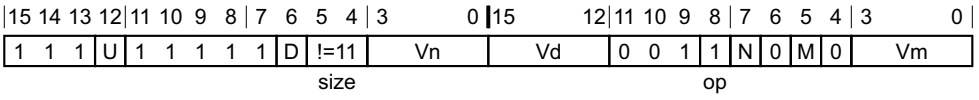

#### *A1 variant*

```
VSUBW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### T1




#### *T1 variant*

```
VSUBW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>
```

#### *Decode for this encoding*

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: Advanced SIMD data-processing instructions.

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VSUBL or VSUBW instruction must be unconditional. ARM strongly recommends that a T32 VSUBL or VSUBW instruction is unconditional, see *Conditional execution* on page F2-2447.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

| | |
|---|---|
| S8 | when U = 0, size = 00 |
| S16 | when U = 0, size = 01 |
| S32 | when U = 0, size = 10 |
| U8 | when U = 1, size = 00 |
| U16 | when U = 1, size = 01 |
| U32 | when U = 1, size = 10 |

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vsubw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 - Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

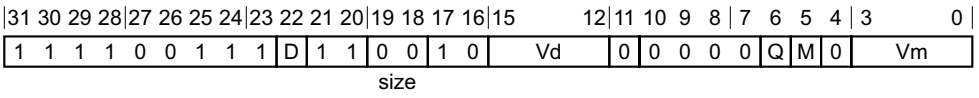
## F7.1.228 VSWP

Vector Swap exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

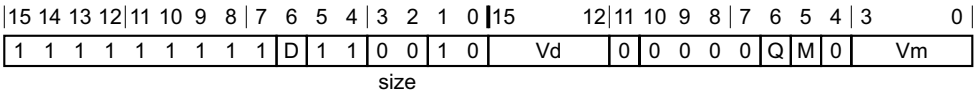
Applies when `Q = 1`.

```
VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VSWP` instruction must be unconditional. ARM strongly recommends that a T32 `VSWP` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

`<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            D[d+r] = Din[m+r];
            D[m+r] = Din[d+r];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.229 VTBL, VTBX

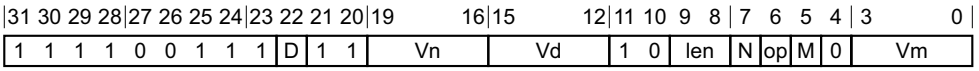
Vector Table Lookup uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.

Vector Table Extension works in the same way, except that indexes out of range leave the destination element unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *VTBL variant*

Applies when `op = 0`.

```
VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

#### *VTBX variant*

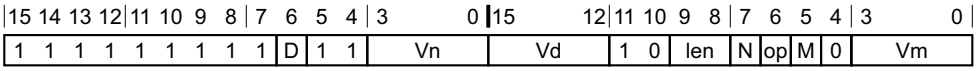
Applies when `op = 1`.

```
VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

#### *Decode for all variants of this encoding*

```
is_vtbl = (op == '0');  length = UInt(len)+1;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;
```

### T1




#### *VTBL variant*

Applies when `op = 0`.

```
VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

#### *VTBX variant*

Applies when `op = 1`.

```
VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

#### *Decode for all variants of this encoding*

```
is_vtbl = (op == '0');  length = UInt(len)+1;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly *VTBL, VTBX* on page K1-5496.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VTBL` or `VTBX` instruction must be unconditional. ARM strongly recommends that a T32 `VTBL` or `VTBX` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<list>` The vectors containing the table. It must be one of:

`{<Dn>}` Encoded as len = `0b00`.

`{<Dn>, <Dn+1>}` Encoded as len = `0b01`.

`{<Dn>, <Dn+1>, <Dn+2>}` Encoded as len = `0b10`.

`{<Dn>, <Dn+1>, <Dn+2>, <Dn+3>}` Encoded as len = `0b11`.

`<Dm>` Is the 64-bit name of the SIMD&FP source register holding the indices, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
    table3 = if length == 4 then D[n+3] else Zeros(64);
    table2 = if length >= 3 then D[n+2] else Zeros(64);
    table1 = if length >= 2 then D[n+1] else Zeros(64);
    table = table3 : table2 : table1 : D[n];

    for i = 0 to 7
        index = UInt(Elem[D[m],i,8]);
        if index < 8*length then
            Elem[D[d],i,8] = Elem[table,index,8];
        else
            if is_vtbl then
                Elem[D[d],i,8] = Zeros(8);
            // else Elem[D[d],i,8] unchanged
```

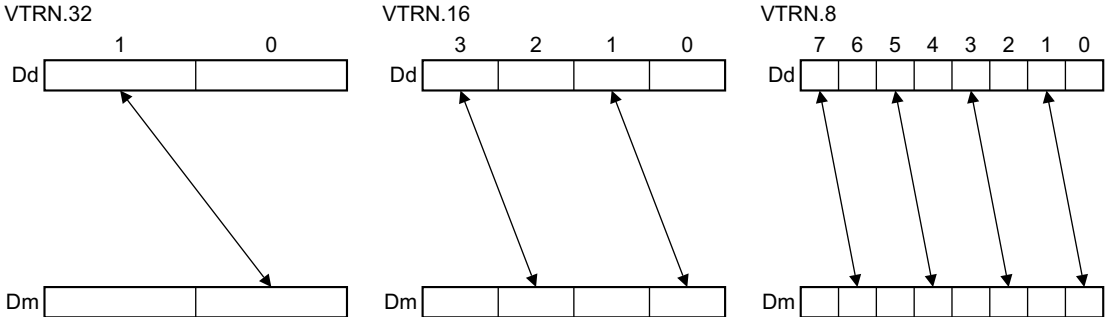

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.230 VTRN

Vector Transpose treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows an example of the operation of VTRN doubleword operations.




Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

This instruction is used by the aliases VUZP (alias) and VZIP (alias). See the *Alias conditions* on page F7-3732 table for details of when each alias is preferred.

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | D | 1 | 1 | size | | 1 | 0 | Vd | | | | 0 | 0 | 0 | 0 | 1 | Q | M | 0 | Vm | | | |

#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | | size | | 1 | 0 | Vd | | | | 0 | 0 | 0 | 0 | 1 | Q | M | 0 | Vm | | | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### 64-bit SIMD vector variant

Applies when `Q = 0`.

```
VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### 128-bit SIMD vector variant

Applies when `Q = 1`.

```
VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### Alias conditions

| Alias | is preferred when |
|---|---|
| VUZP (alias) | Never |
| VZIP (alias) | Never |

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 `VTRN` instruction must be unconditional. ARM strongly recommends that a T32 `VTRN` instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

The encoding `size = 11` is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements DIV 2;

    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
for e = 0 to h-1
    Elem[D[d+r],2*e+1,esize] = Elem[Din[m+r],2*e,esize];
    Elem[D[m+r],2*e,esize] = Elem[Din[d+r],2*e+1,esize];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.231 VTST

Vector Test Bits takes each element in a vector, and bitwise ANDs it with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

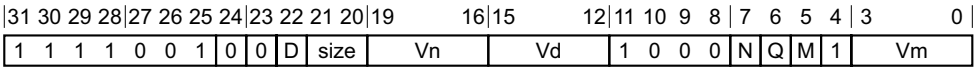
- 8-bit, 16-bit, or 32-bit fields.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

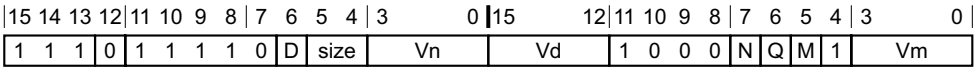
Applies when `Q = 1`.

```
VTST{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

### T1




#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VTST{<c>}{<q>}.<size> {<Dd>,} <Dn>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VTST{<c>}{<q>}.<size> {<Qd>,} <Qn>, <Qm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Decode for all variants of this encoding*

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VTST instruction must be unconditional. ARM strongly recommends that a T32 VTST instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<size> The data size for the elements of the operands. It must be one of:

- 8 Encoded as size = 0b00.
- 16 Encoded as size = 0b01.
- 32 Encoded as size = 0b10.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

- 8 when size = 00
- 16 when size = 01
- 32 when size = 10

The encoding size = 11 is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !IsZero(Elem[D[n+r],e,esize] AND Elem[D[m+r],e,esize]) then
                Elem[D[d+r],e,esize] = Ones(esize);
            else
                Elem[D[d+r],e,esize] = Zeros(esize);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.232 VUZP

Vector Unzip de-interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows an example of the operation of VUZP doubleword operation for data type 8.

VUZP.8, doubleword

| | Register state before operation | | | | | | | | Register state after operation | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dd | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | B6 | B4 | B2 | B0 | A6 | A4 | A2 | A0 |
| Dm | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 | B7 | B5 | B3 | B1 | A7 | A5 | A3 | A1 |

The following figure shows an example of the operation of VUZP quadword operation for data type 32.

VUZP.32, quadword

| | Register state before operation | | | | Register state after operation | | | |
|---|---|---|---|---|---|---|---|---|
| Qd | A3 | A2 | A1 | A0 | B2 | B0 | A2 | A0 |
| Qm | B3 | B2 | B1 | B0 | B3 | B1 | A3 | A1 |

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | D | 1 | 1 | size | | 1 | 0 | Vd | | | | 0 | 0 | 0 | 1 | 0 | Q | M | 0 | Vm | | | |

#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | | size | | 1 | 0 | Vd | | | | 0 | 0 | 0 | 1 | 0 | Q | M | 0 | Vm | | | |

#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

### *Decode for all variants of this encoding*

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

## Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 `VUZP` instruction must be unconditional. ARM strongly recommends that a T32 `VUZP` instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

The encoding `size = 1x` is reserved.

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

`8` when `size = 00`

`16` when `size = 01`

`32` when `size = 10`

The encoding `size = 11` is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;  Q[m>>1] = bits(128) UNKNOWN;
        else
            zipped_q = Q[m>>1]:Q[d>>1];
            for e = 0 to (128 DIV esize) - 1
                Elem[Q[d>>1],e,esize] = Elem[zipped_q,2*e,esize];
                Elem[Q[m>>1],e,esize] = Elem[zipped_q,2*e+1,esize];
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;  D[m] = bits(64) UNKNOWN;
        else
            zipped_d = D[m]:D[d];
            for e = 0 to (64 DIV esize) - 1
                Elem[D[d],e,esize] = Elem[zipped_d,2*e,esize];
                Elem[D[m],e,esize] = Elem[zipped_d,2*e+1,esize];
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

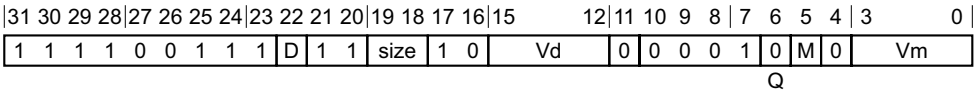
## F7.1.233 VUZP (alias)

Vector Unzip de-interleaves the elements of two vectors.

This instruction is an alias of the VTRN instruction. This means that:

- The encodings in this description are named to match the encodings of VTRN.
- The description of VTRN gives the operational pseudocode for this instruction.

### A1




#### *64-bit SIMD vector variant*

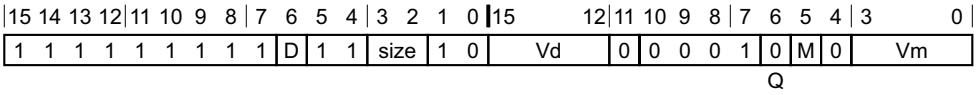
`VUZP{<c>}{<q>}.32 <Dd>, <Dm>`

is equivalent to

`VTRN{<c>}{<q>}.32 <Dd>, <Dm>`

and is never the preferred disassembly.

### T1




#### *64-bit SIMD vector variant*

`VUZP{<c>}{<q>}.32 <Dd>, <Dm>`

is equivalent to

`VTRN{<c>}{<q>}.32 <Dd>, <Dm>`

and is never the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VUZP` instruction must be unconditional. ARM strongly recommends that a T32 `VUZP` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

The description of VTRN gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## F7.1.234 VZIP

Vector Zip interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows an example of the operation of VZIP doubleword operation for data type 8.

VZIP.8, doubleword

| | Register state before operation | | | | | | | | Register state after operation | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dd | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | B3 | A3 | B2 | A2 | B1 | A1 | B0 | A0 |
| Dm | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 | B7 | A7 | B6 | A6 | B5 | A5 | B4 | A4 |

The following figure shows an example of the operation of VZIP quadword operation for data type 32.

VZIP.32, quadword

| | Register state before operation | | | | Register state after operation | | | |
|---|---|---|---|---|---|---|---|---|
| Qd | A3 | A2 | A1 | A0 | B1 | A1 | B0 | A0 |
| Qm | B3 | B2 | B1 | B0 | B3 | A3 | B2 | A2 |

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-3841.

ARM deprecates the conditional execution of any Advanced SIMD instruction encoding that is not also available as a floating-point instruction encoding, see *Conditional execution* on page F2-2447.

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | D | 1 | 1 | size | | 1 | 0 | Vd | | | | 0 | 0 | 0 | 1 | 1 | Q | M | 0 | Vm | | | |

#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

#### *128-bit SIMD vector variant*

Applies when `Q = 1`.

```
VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### *Decode for all variants of this encoding*

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | | size | | 1 | 0 | Vd | | | | 0 | 0 | 0 | 1 | 1 | Q | M | 0 | Vm | | | |

#### *64-bit SIMD vector variant*

Applies when `Q = 0`.

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### 128-bit SIMD vector variant

Applies when Q = 1.

```
VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

#### Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

### Assembler symbols

<c> See *Standard assembler syntax fields* on page F2-2446. An A32 VZIP instruction must be unconditional. ARM strongly recommends that a T32 VZIP instruction is unconditional, see *Conditional execution* on page F2-2447.

<q> See *Standard assembler syntax fields* on page F2-2446.

<dt> For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

- 8 when size = 00
- 16 when size = 01

The encoding size = 1x is reserved.

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

- 8 when size = 00
- 16 when size = 01
- 32 when size = 10

The encoding size = 11 is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;  Q[m>>1] = bits(128) UNKNOWN;
        else
            bits(256) zipped_q;
            for e = 0 to (128 DIV esize) - 1
                Elem[zipped_q,2*e,esize] = Elem[Q[d>>1],e,esize];
                Elem[zipped_q,2*e+1,esize] = Elem[Q[m>>1],e,esize];
            Q[d>>1] = zipped_q<127:0>;  Q[m>>1] = zipped_q<255:128>;
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;  D[m] = bits(64) UNKNOWN;
        else
            bits(128) zipped_d;
            for e = 0 to (64 DIV esize) - 1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Elem[zipped_d,2*e,esize] = Elem[D[d],e,esize];
        Elem[zipped_d,2*e+1,esize] = Elem[D[m],e,esize];
    D[d] = zipped_d<63:0>;  D[m] = zipped_d<127:64>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

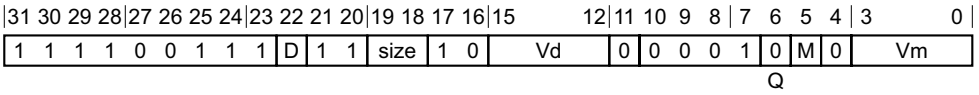
### F7.1.235 VZIP (alias)

Vector Zip interleaves the elements of two vectors.

This instruction is an alias of the VTRN instruction. This means that:

- The encodings in this description are named to match the encodings of VTRN.
- The description of VTRN gives the operational pseudocode for this instruction.

#### A1




##### *64-bit SIMD vector variant*

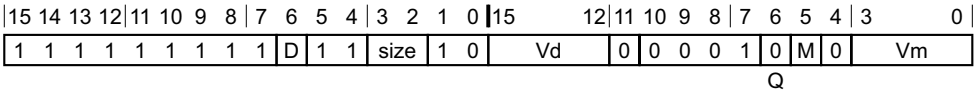
`VZIP{<c>}{<q>}.32 <Dd>, <Dm>`

is equivalent to

`VTRN{<c>}{<q>}.32 <Dd>, <Dm>`

and is never the preferred disassembly.

#### T1




##### *64-bit SIMD vector variant*

`VZIP{<c>}{<q>}.32 <Dd>, <Dm>`

is equivalent to

`VTRN{<c>}{<q>}.32 <Dd>, <Dm>`

and is never the preferred disassembly.

### Assembler symbols

`<c>` See *Standard assembler syntax fields* on page F2-2446. An A32 `VZIP` instruction must be unconditional. ARM strongly recommends that a T32 `VZIP` instruction is unconditional, see *Conditional execution* on page F2-2447.

`<q>` See *Standard assembler syntax fields* on page F2-2446.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation for all encodings

The description of VTRN gives the operational pseudocode for this instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Part G
## The AArch32 System Level Architecture

# Chapter G1
# The AArch32 System Level Programmers' Model

This chapter gives a system level description of the programmers' model for execution in AArch32 state. It contains the following sections:


- *About the AArch32 System level programmers' model* on page G1-3746.
- *Exception levels* on page G1-3747.
- *Exception terminology* on page G1-3748.
- *Execution state* on page G1-3750.
- *Instruction Set state* on page G1-3752.
- *Security state* on page G1-3753.
- *Virtualization* on page G1-3756.
- *AArch32 PE modes, and general-purpose and Special-purpose registers* on page G1-3758.
- *Process state, PSTATE* on page G1-3767.
- *Instruction set states* on page G1-3772.
- *Handling exceptions that are taken to an Exception level using AArch32* on page G1-3774.
- *Exception return to an Exception level using AArch32* on page G1-3796.
- *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-3801.
- *AArch32 state exception descriptions* on page G1-3811.
- *Reset into AArch32 state* on page G1-3830.
- *Mechanisms for entering a low-power state* on page G1-3834.
- *The conceptual coprocessor interface and system control* on page G1-3839.
- *Advanced SIMD and floating-point support* on page G1-3841.
- *Configurable instruction enables and disables, and trap controls* on page G1-3846.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G1.1 About the AArch32 System level programmers' model

An application programmer has only a restricted view of the system. The System level programmers' model supports this application level view of the system, and includes features that are required for one or both of an operating system (OS) and a hypervisor to provide the programming environment seen by an application. This chapter describes the System level programmers' model when executing at EL1 or higher in an Exception level that is using AArch32.

The system level programmers' model includes all of the system features required to support operating systems and to handle hardware events.

The following sections give a system level introduction to the basic concepts of the ARM architecture AArch32 state, and the terminology that is used for describing the architecture when executing in this state:

- *Exception levels* on page G1-3747.
- *Exception terminology* on page G1-3748.
- *Execution state* on page G1-3750.
- *Instruction Set state* on page G1-3752.
- *Security state* on page G1-3753.
- *Virtualization* on page G1-3756.

The rest of this chapter describes the system level programmers' model when executing in AArch32 state.

The other chapters in this part describe:

- The memory system architecture, as seen when executing in an Exception level that is using AArch32:
  - Chapter G3 *The AArch32 System Level Memory Model* describes the general features of the ARMv8 memory model, when executing in AArch32 state, that are not visible at the application level.

    **Note**

    Chapter E2 *The AArch32 Application Level Memory Model* describes the application level view of the memory model.

  - Chapter G4 *The AArch32 Virtual Memory System Architecture* describes the Virtual Memory System Architecture (VMSA) used in AArch32 state.
- The AArch32 System Registers, see Chapter G6 *AArch32 System Register Descriptions*.

**Note**

The T32 and A32 instruction sets include instructions that provide system level functionality, such as returning from an exception. See for example, *ERET* on page F6-2683.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.2 Exception levels

The ARMv8-A architecture defines a set of Exception levels, EL0 to EL3, where:

- If ELn is the Exception level, increased values of *n* indicate increased software execution privilege.
- Execution at EL0 is called *unprivileged execution*.
- EL2 provides support for virtualization of Non-secure operation.
- EL3 provides support for switching between two Security states, Secure state and Non-secure state.

An implementation might not include all of the Exception levels. All implementations must include EL0 and EL1. EL2 and EL3 are optional.

**Note**

A PE is not required to implement a contiguous set of Exception levels. For example, it is permissible for an implementation to include only EL0, EL1, and EL3.

*The effect of implementation choices on the programmers' model* on page D1-1631 provides information on implementations.

When executing in AArch32 state, execution can move between Exception levels only on taking an exception or on returning from an exception:

- On taking an exception, the Exception level can only increase or remain the same.
- On returning from an exception, the Exception level can only decrease or remain the same.

The Exception level that execution changes to or remains in on taking an exception is called the *target Exception level* of the exception.

Each exception type has a target Exception level that is either:

- Implicit in the nature of the exception.
- Defined by configuration bits in the System registers.

An exception cannot target EL0.

Exception levels exist within *Security states*. *The ARMv8-A security model* on page G1-3753 describes this. When executing at an Exception level, the PE can access both of the following:

- The resources that are available for the combination of the current Exception level and the current Security state.
- The resources that are available at all lower Exception levels, provided that those resources are available to the current Security state.

This means that if the implementation includes EL3, then because EL3 is only implemented in Secure state, execution at EL3 can access all resources available at all Exception levels, for both Security states.

Each Exception level other than EL0 has its own translation regime and associated control registers. For information on the translation regimes, see Chapter G4 *The AArch32 Virtual Memory System Architecture*.

### G1.2.1 Typical Exception level usage model

The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

**EL0** Applications.

**EL1** OS kernel and associated functions that are typically described as *privileged*.

**EL2** Hypervisor.

**EL3** Secure monitor.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.3 Exception terminology

The following subsections define the terms that are used when describing exceptions:

- *Terminology for taking an exception*.
- *Terminology for returning from an exception*.
- *Exception levels*.
- *Definition of a precise exception*.
- *Definitions of synchronous and asynchronous exceptions* on page G1-3749.

### G1.3.1 Terminology for taking an exception

An exception is *generated* when the PE first responds to an exceptional condition. The PE state at this time is the state that the exception is *taken from*. The PE state immediately after taking the exception is the state that the exception is *taken to*.

### G1.3.2 Terminology for returning from an exception

To return from an exception, the PE must execute an exception return instruction. The PE state when an exception return instruction is committed for execution is the state the exception *returns from*. The PE state immediately after the execution of that instruction is the state that the exception *returns to*.

### G1.3.3 Exception levels

An Exception level, EL*n*, with a larger value of *n* than another Exception level, is described as being a *higher* Exception level than the other Exception level. For example, EL3 is a higher Exception level than EL1.

An Exception level with a smaller value of *n* than another Exception level is described as being a *lower* Exception level than the other Exception level. For example, EL0 is a lower Exception level than EL1.

An Exception level is described as:

- *Using AArch64* when execution in that Exception level is in the AArch64 Execution state.
- *Using AArch32* when execution in that Exception level is in the AArch32 Execution state.

### G1.3.4 Definition of a precise exception

An exception is described as *precise* when the exception handler receives the PE state and memory system state that is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken, and none afterwards.

Other than the *Asynchronous Data Abort*, sometimes referred to as an *external interrupt*, all exceptions that are taken to AArch32 state are required to be precise. For each occurrence of an Asynchronous Data Abort, whether the interrupt is precise or imprecise is IMPLEMENTATION DEFINED.

Where a synchronous exception that is taken to AArch32 state is generated as part of an instruction that performs more than one single-copy atomic memory access, the definition of precise permits that the values in registers or memory affected by those instructions can be UNKNOWN, provided that:

- The accesses affecting those registers or memory locations do not, themselves, generate exceptions.
- The registers are not involved in the calculation of the memory address that is used by the instruction.

In AArch32 state, examples of instructions that perform more than one single-copy atomic memory access are the `LDM` and `STM` instructions.

**Note**

- For the definition of a single-copy atomic access, see *Properties of single-copy atomic accesses* on page E2-2369.
- Asynchronous Data Aborts are known as SError interrupts in AArch64 state.
- By definition, all synchronous aborts are precise.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G1.3.5 Definitions of synchronous and asynchronous exceptions

An exception is described as *synchronous* if all of the following apply:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The return address presented to the exception handler is guaranteed to indicate the instruction that caused the exception.
- The exception is precise.

An exception is described as *asynchronous* if any of the following apply:

- The exception is not generated as a result of direct execution or attempted execution of the instruction stream.
- The return address presented to the exception handler is not guaranteed to indicate the instruction that caused the exception.
- The exception is imprecise.

For more information about exceptions, see *Handling exceptions that are taken to an Exception level using AArch32* on page G1-3774.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.4 Execution state

The Execution states are:

**AArch64** The 64-bit Execution state.

**AArch32** The 32-bit Execution state. Operation in this state is compatible with ARMv7-A operation.

*Execution state* on page A1-33 gives more information about them.

Exception levels *use* Execution states. For example, EL0, EL1 and EL2 might all be using AArch32, under EL3 using AArch64.

This means that:

- Different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.
- The PE can change Execution states only either:
  - At reset.
  - On a change of Exception level.

**Note**

- *Typical Exception level usage model* on page G1-3747 shows which Exception levels different software layers might typically use.
- *The effect of implementation choices on the programmers' model* on page D1-1631 gives information on supported configurations of Exception levels and Execution states.

The interaction between the AArch64 and AArch32 Execution states is called *interprocessing*. *Interprocessing* on page D1-1619 describes this.

### G1.4.1 About the AArch32 PE modes

AArch32 state provides a set of *PE modes* that support normal software execution and handle exceptions. The current mode determines the set of registers that are available, as described in *AArch32 general-purpose registers, the PC, and the Special-purpose registers* on page G1-3763.

The AArch32 modes are:

- Monitor mode. This mode always executes at Secure EL3.
- Hyp mode. This mode always executes at Non-secure EL2.
- System, Supervisor, Abort, Undefined, IRQ and FIQ modes. The Exception level these modes execute at depends on the Security state, as described in *Security state* on page G1-3753.
- User mode. This mode always executes at EL0.

**Note**

AArch64 state does not support modes. Modes are a concept that is specific to AArch32 state. Modes that execute at a particular Exception level are only implemented if that Exception level supports using AArch32.

For more information on modes see *AArch32 PE mode descriptions* on page G1-3758.

The mode in use immediately before an exception is taken is described as the mode the exception is *taken from*. The mode that is used on taking the exception is described as the mode the exception is *taken to*.

All of the following define the mode that an exception is taken to:

- The type of exception.
- The mode the exception is taken from.
- Configuration settings defined at EL2 and EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Monitor mode and Hyp mode can create system traps that cause exceptions to EL3 or EL2 respectively. There is an architected hierarchy where EL2 and EL3 configuration settings affect a common condition, for example interrupt routing. When no traps are enabled for a particular condition, the AArch32 mode an exception is taken to is called the *default mode* for that exception.

In AArch32 state, a number of different modes can exist at the same *Privilege level* (PL). All modes at a particular privilege level have the same access rights for accesses to memory and to System registers. The mapping of PE modes to Exception levels depends on the Security state, as described in *Security state* on page G1-3753.

*Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990 gives more information about the PE modes, their associated Privilege levels, and how these map onto the Exception levels.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.5 Instruction Set state

In AArch32 state, the *Instruction Set state* determines the instruction set that the PE is executing. In an implementation that follows the ARM recommendations, the available Instruction Set states are:

**T32 state** The PE is executing T32 instructions.

**A32 state** The PE is executing A32 instructions.

**Note**

In previous versions of the ARM architecture:

- The T32 instruction set was called the Thumb instruction set.
- The A32 instruction set was called the ARM instruction set.

For more information, see *Process state, PSTATE* on page E1-2319.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.6 Security state

The ARMv8-A architecture provides two Security states, each with an associated physical memory address space, as follows:

**Secure state** When in this state, the PE can access both the Secure physical address space and the Non-secure physical address space.

**Non-secure state** When in this state, the PE:
- Can access only the Non-secure physical address space.
- Cannot access the Secure system control resources.

For information on how virtual addresses translate onto Secure physical and Non-secure addresses, see *About VMSAv8-32* on page G4-3992.

### G1.6.1 The ARMv8-A security model

The general principles of the ARMv8-A security model are:

- If the implementation includes EL3 then it has two Security states, Secure and Non-secure, and:
  - EL3 exists only in Secure state.
  - A change from Non-secure state to Secure state can only occur on taking an exception to EL3.
  - A change from Secure state to Non-secure state can only occur on an exception return from EL3.
  - If EL2 is implemented, it exists only in Non-secure state.
- If the implementation does not include EL3 it has one Security state, that is:
  - IMPLEMENTATION DEFINED, if the implementation does not include EL2.
  - Non-secure state if the implementation includes EL2.

#### The AArch32 security model, and execution privilege

The Exception level hierarchy of four Exception levels, EL0, EL1, EL2, and EL3, applies to execution in both Execution states. This section describes the mapping between Exception levels, AArch32 modes, and Privilege levels.

The AArch32 modes Monitor, System, Supervisor, Abort, Undefined, IRQ, and FIQ all have the same privilege. In the AArch32 Privilege model this is PL1.

In Secure state:

- Monitor mode executes only at EL3, and is accessible only when EL3 is using AArch32.
- System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode all:
  - Execute at EL1 when EL3 is using AArch64.
  - Execute at EL3 when EL3 is using AArch32.

This means that there is a difference in the Secure state hierarchy that the PE is using, depending on which Execution state EL3 is using:

- If EL3 is using AArch64:
  - There is no support for Monitor mode.
  - If EL1 is using AArch32, System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode execute at Secure EL1.
- If EL3 is using AArch32:
  - Monitor mode is supported, and executes at Secure EL3
  - System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode execute at Secure EL3.
  - There is no support for a Secure EL1 Exception level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See *Security behavior in Exception levels using AArch32 when EL3 is using AArch64* on page G1-3786 for more information about operation in a Secure EL1 mode when EL3 is using AArch64.

In Non-secure state, the PL1 modes System, Supervisor, Abort, Undefined, IRQ, and FIQ always execute at EL1.

User mode always executes at EL0 and has Privilege level PL0. Hyp mode always executes at EL2 and has Privilege level PL2. See *About the AArch32 PE modes* on page G1-3750.

**Note**

For more information about the Privilege level terminology, see *Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990.

Figure G1-1 shows the security model when EL3 is using AArch32, and shows the expected use of the different Exception levels, and which modes execute at which Exception levels.

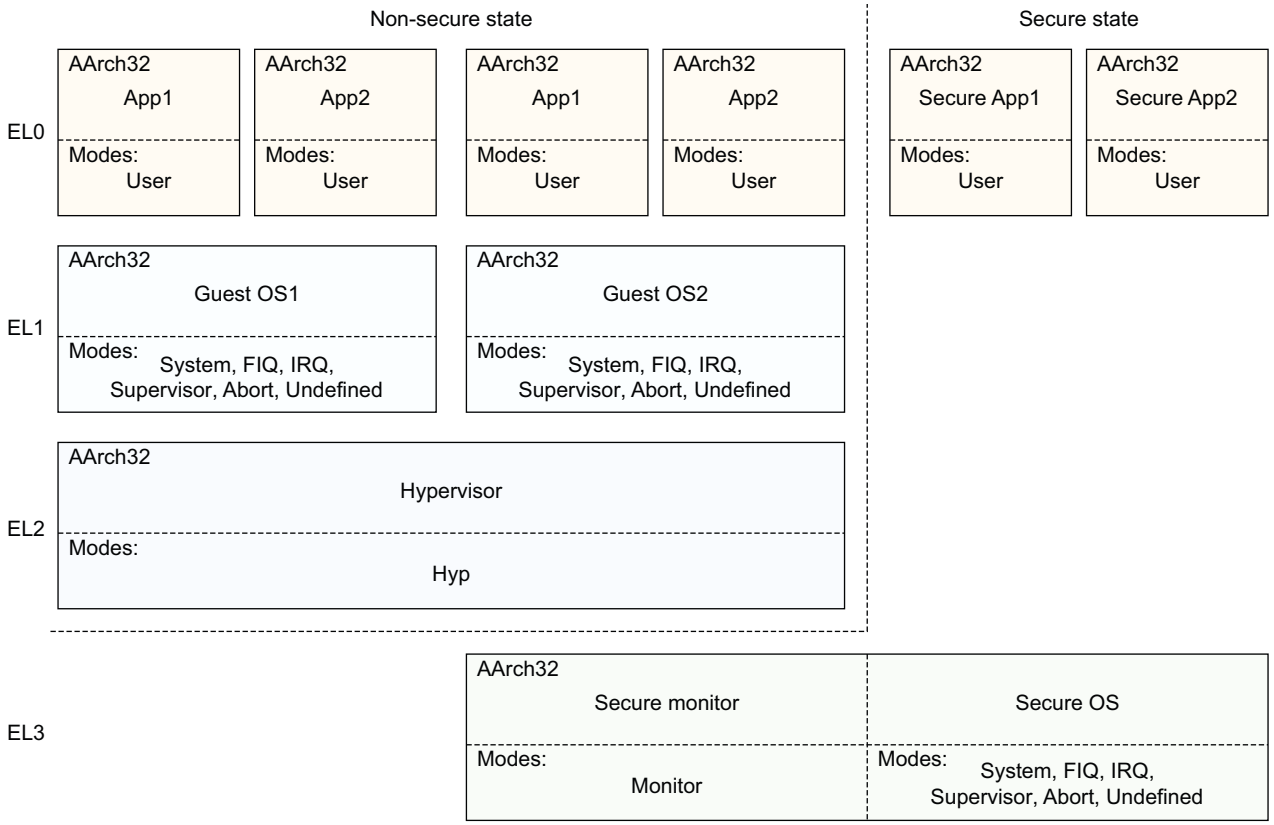



**Figure G1-1 ARMv8-A Security model when EL3 is using AArch32**

**Note**

For an overview of the Security models when EL3 is using AArch64:

- See Figure G1-2 on page G1-3761 for the case where EL2, EL1, and EL0 are all using AArch32. This figure shows the implementation of the PE modes.
- See Figure D1-1 on page D1-1519 for an overview of the set of possible implementations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Figure G1-1 on page G1-3754 shows that when EL3 is using AArch32, the Exception levels and modes available in each Security state are as follows:

**Secure state**

**EL0** User mode.

**EL3** Any mode that is available in Secure state, other than User mode.

**Non-secure state**

**EL0** User mode.

**EL1** Any mode that is available in Non-secure state, other than Hyp mode and User mode.

**EL2** Hyp mode.

Execution at EL0 is described as *unprivileged execution*.

A mode associated with a particular Exception level, EL*n*, is described as an EL*n* mode.

**Note**

The Exception level defines the ability to access resources in the current Security state, and does not imply anything about the ability to access resources in the other Security state.

When EL3 is using AArch32, many AArch32 System registers accessible at PL1 are *banked* between the Secure and Non-secure states.

When EL3 is using AArch64 and Secure EL1 is using AArch32, System registers accessible at PL1 are not banked between the Non-secure and Secure states. Software running at EL3 is expected to switch the content of the PL1 accessible System registers between the Secure and Non-secure context, in a similar manner to switching the contents of general purpose registers. For information on the relationship between AArch64 and AArch32 System registers in an interprocessing environment, see *Mapping of the System registers between the Execution states* on page D1-1622.

For more information on the System registers, see *The conceptual coprocessor interface and system control* on page G1-3839.

The *Secure Monitor Call* (`SMC`) instruction provides software with a system call to EL3. When executing at a privileged Exception level, `SMC` instructions generates exceptions. For more information, see *Secure Monitor Call (SMC) exception* on page G1-3816 and *SMC* on page F6-2979.

**Note**

For more information about the Privilege level terminology, see *Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990.

### Changing from Secure state to Non-secure state

Monitor mode is provided to support switching between Secure and Non-secure states. When executing in an Exception level that is using AArch32, except in Monitor mode and Hyp mode, the Security state is controlled:

- By the SCR.NS bit, when EL3 is using AArch32.
- By the SCR_EL3.NS bit, when EL3 is using AArch64.

The mapping of AArch32 privileged modes to the exception hierarchy means that it is possible when EL3 is using AArch32 to change from EL3 to Non-secure EL1 without an exception return. This can occur in one of the following ways:

- Using an `MSR` or `CPS` instruction to switch from Monitor mode to another privileged mode while SCR.NS is 1.
- Using an `MCR` instruction that writes SCR.NS to change from Secure to Non-secure state when in a privileged mode other than Monitor mode.

ARM strongly recommends that software executing at EL3 using AArch32 does not use either of these mechanisms to change from EL3 to Non-secure EL1 without an exception return. The use of both of these mechanisms is deprecated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.7 Virtualization

The support for virtualization described in this section applies only to an implementation that includes EL2. A PE is in *Hyp mode* when it is executing at EL2 in the AArch32 state. An exception return from Hyp mode to software running at EL1 or EL0 is performed using the `ERET` instruction.

EL2 provides a set of features that support virtualizing the Non-secure state of an ARMv8-A implementation. The basic model of a virtualized system involves:

- A hypervisor, running in EL2, that is responsible for switching between *virtual machines*. A virtual machine is comprised of Non-secure EL1 and Non-secure EL0.
- A number of Guest operating systems, that each run in Non-secure EL1, on a virtual machine.
- For each Guest operating system, applications, that usually run in Non-secure EL0, on a virtual machine.

**Note**

In some systems, a Guest OS is unaware that it is running on a virtual machine, and is unaware of any other Guest OS. In other systems, a hypervisor makes the Guest OS aware of these facts. The ARMv8-A architecture supports both of these models.

The hypervisor assigns a *virtual machine identifier* (VMID) to each virtual machine.

EL2 is implemented only in Non-secure state, to support Guest OS management. EL2 provides controls to:

- Provide virtual values for the contents of a small number of identification registers. A read of one of these registers by a Guest OS or the applications for a Guest OS returns the virtual value.
- *Trap* various operations, including memory management operations and accesses to many other registers. A trapped operation generates an exception that is taken to EL2.
- Route interrupts to the appropriate one of:
  - The current Guest OS.
  - A Guest OS that is not currently running.
  - The hypervisor.

In Non-secure state:

- The implementation provides an independent *translation regime* for memory accesses from EL2.
- For the PL1&0 translation regime, address translation occurs in two stages:
  - Stage 1 maps the *virtual address* (VA) to an *intermediate physical address* (IPA). This is managed at EL1, usually by a Guest OS. The Guest OS believes that the IPA is the *physical address* (PA).
  - Stage 2 maps the IPA to the PA. This is managed at EL2. The Guest OS might be completely unaware of this stage.

For more information on the translation regimes, see Chapter G4 *The AArch32 Virtual Memory System Architecture*.

### G1.7.1 The effect of implementing EL2 on the Exception model

An implementation that includes EL2 implements the following exceptions:

- Hypervisor Call (HVC) exception.
- Traps to EL2. *EL2 configurable controls* on page G1-3855, describes these.
- All of the virtual interrupts:
  - Virtual External interrupt.
  - Virtual IRQ.
  - Virtual FIQ.

HVC exceptions are always taken to EL2. All virtual interrupts are always taken to EL1, and can only be taken from Non-secure EL1 or EL0.

Each of the virtual interrupts can be independently enabled using controls at EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Each of the virtual interrupts has a corresponding physical interrupt. See *Virtual interrupts*.

When a virtual interrupt is enabled, its corresponding physical exception is taken to EL2, unless EL3 has configured that physical exception to be taken to EL3. For more information, see *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-3801.

An implementation that includes EL2 also:

- Provides controls that can be used to route some synchronous exceptions, taken from Non-secure state, to EL2. For more information see:
  - *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.
  - *Routing debug exceptions to EL2* on page G1-3795.
- Provides mechanisms to trap PE operations to EL2. See *EL2 configurable controls* on page G1-3855.

  When an operation is trapped to EL2, the hypervisor typically either:
  - Emulates the required operation. The application running in the Guest OS is unaware of the trap.
  - Returns an error to the Guest OS.

## Virtual interrupts

The virtual interrupts have names that correspond to the physical interrupts, as shown in Table G1-1.

**Table G1-1 The virtual interrupts**

| Physical interrupt | Corresponding virtual interrupt |
|---|---|
| External abort | Virtual External Abort |
| IRQ | Virtual IRQ |
| FIQ | Virtual FIQ |

Software executing at EL2 can use virtual interrupts to signal physical interrupts to Non-secure EL1 and Non-secure EL0. Example G1-1 shows a usage model for virtual interrupts.

**Example G1-1 Virtual interrupt usage model**

A usage model is as follows:

1. Software executing at EL2 routes a physical interrupt to EL2.
2. When a physical interrupt of that type occurs, the exception handler executing in EL2 determines whether the interrupt can be handled in EL2 or requires routing to a Guest OS in EL1. If the interrupt requires routing to a Guest OS:
   - If the Guest OS is currently running, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.
   - If the Guest OS is not currently running, the physical interrupt is marked as pending for the guest OS. When the hypervisor next switches to the virtual machine that is running that Guest OS, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.

Non-secure EL1 and Non-secure EL0 modes cannot distinguish a virtual interrupt from the corresponding physical interrupt.

For more information see *Virtual exceptions when an implementation includes EL2* on page G1-3801.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.8 AArch32 PE modes, and general-purpose and Special-purpose registers

The following sections describe the AArch32 PE modes and the general-purpose registers and the PC:

- *AArch32 PE mode descriptions*.
- *AArch32 general-purpose registers, the PC, and the Special-purpose registers* on page G1-3763.
- *Saved Program Status Registers (SPSRs)* on page G1-3765.
- *ELR_hyp* on page G1-3766.

**Note**

The PC is included in the scope of this section because, in AArch32 state, it is defined as being part of the same register file as the general-purpose registers. That is, the AArch32 register file R0-R15 comprises:

- The general-purpose registers R0-R14.
- The PC, that can be described as R15.

### G1.8.1 AArch32 PE mode descriptions

Table G1-2 shows the PE modes defined by the ARM architecture, for execution in AArch32 state. In this table:

- The *PE mode* column gives the name of each mode and the abbreviation used, for example, in the general-purpose register name suffixes used in *AArch32 general-purpose registers, the PC, and the Special-purpose registers* on page G1-3763.
- The *Encoding* column gives the corresponding PSTATE.M field.
- The *Exception level* column gives the Exception level at which the mode is implemented, including dependencies on the current Security state and on whether EL3 is using AArch32, see *Exception levels* on page G1-3747.

**Table G1-2 AArch32 PE modes**

| PE mode | | Encoding | Security state | Exception level | Implemented |
|---|---|---|---|---|---|
| User | usr | 10000 | Both | EL0 | Always |
| FIQ | fiq | 10001 | Non-secure<br>Secure | EL1<br>EL1 or EL3[a] | Always |
| IRQ | irq | 10010 | Non-secure<br>Secure | EL1<br>EL1 or EL3[a] | Always |
| Supervisor | svc | 10011 | Non-secure<br>Secure | EL1<br>EL1 or EL3[a] | Always |
| Monitor | mon | 10110 | Secure | EL3 | If EL3 implemented and using AArch32 |
| Abort | abc | 10111 | Non-secure<br>Secure | EL1<br>EL1 or EL3[a] | Always |
| Hyp | hyp | 11010 | Non-secure | EL2 | If EL2 implemented and using AArch32 |
| Undefined | und | 11011 | Non-secure<br>Secure | EL1<br>EL1 or EL3[a] | Always |
| System | sys | 11111 | Non-secure<br>Secure | EL1<br>EL1 or EL3[a] | Always |

a. EL3 if EL3 is using AArch32. EL1 if EL3 is using AArch64 and EL1 is using AArch32.

Mode changes can be made under software control, or can be caused by an external or internal exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Notes on the AArch32 PE modes

PE modes are defined only in AArch32. Because each mode is implemented as part of a particular Exception level that is using AArch32, the set of available modes depends on which Exception levels are implemented and using AArch32, as described in *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-3761.

This section gives more information about each of the modes, when it is implemented.

**User mode** Software executing in User mode executes at EL0. Execution in User mode is sometimes described as unprivileged execution. Application programs normally execute in User mode, and any program executed in User mode:

- Makes only unprivileged accesses to system resources, meaning it cannot access protected system resources.
- Makes only unprivileged access to memory.
- Cannot change mode except by causing an exception, see *Handling exceptions that are taken to an Exception level using AArch32* on page G1-3774.

**System mode** System mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-3761.

System mode has the same registers available as User mode, and is not entered by any exception.

**Supervisor mode**

Supervisor mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-3761.

Supervisor mode is the default mode to which a Supervisor Call exception is taken. Executing a SVC (Supervisor Call) instruction generates a Supervisor Call exception.

In an implementation where the highest implemented Exception level is using AArch32, if that Exception level is EL3 or EL1, a PE enters Supervisor mode on Reset.

**Abort mode** Abort mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-3761.

Abort mode is the default mode to which a Data Abort exception or Prefetch Abort exception is taken.

**Undefined mode**

Undefined mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-3761.

Undefined mode is the default mode to which an instruction-related exception, including any attempt to execute an UNDEFINED instruction, is taken.

**FIQ mode** FIQ mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-3761.

FIQ mode is the default mode to which an FIQ interrupt is taken.

**IRQ mode** IRQ mode is implemented at EL1 or EL3, see *Effect of the EL3 Execution state on the PE modes and Exception levels* on page G1-3761.

IRQ mode is the default mode to which an IRQ interrupt is taken.

**Hyp mode** Hyp mode is the Non-secure EL2 mode.

Hyp mode is entered on taking an exception from Non-secure state that must be taken to EL2.

In an implementation where the highest implemented Exception level is EL2 and EL2 uses AArch32 on reset, a PE enters Hyp mode on Reset.

The Hypervisor Call exception and Hyp Trap exception are implemented as part of EL2 and are always taken to Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

This means that Hypervisor Call and Hyp Trap exceptions cannot be taken from Secure state.

When the value of the Hypervisor Call enable bit, SCR.HCE, is 1, executing an HVC (Hypervisor Call) instruction in a Non-secure EL1 mode generates a Hypervisor Call exception.

For more information, see *Hyp mode* on page G1-3762.

**Monitor mode**

Monitor mode is the Secure EL3 mode. This means it is always in the Secure state, regardless of the value of the SCR.NS bit.

Monitor mode is the mode to which a Secure Monitor Call exception is taken. In a Non-secure EL1 mode, or a Secure EL3 mode, executing an SMC (Secure Monitor Call) instruction generates a Secure Monitor Call exception.

When EL3 is using AArch32, some exceptions that are taken to a different mode by default can be configured to be taken to EL3, see *PE mode for taking exceptions* on page G1-3784.

When EL3 is using AArch32, software executing in Monitor mode:

- Has access to both the Secure and Non-secure copies of System registers.
- Can perform an exception return to Secure state, or to Non-secure state.

This means that, when EL3 is using AArch32, Monitor mode provides the only recommended method of changing between the Secure and Non-secure Security states.

**Secure and Non-secure modes**

In an implementation that includes EL3, the names of most implemented modes can be qualified as Secure or Non-secure, to indicate whether the PE is also in Secure state or Non-secure state. For example:

- If a PE is in Supervisor mode and Secure state, it is in *Secure Supervisor mode*.
- If a PE is in User mode and Non-secure state, it is in *Non-secure User mode*.

**Note**

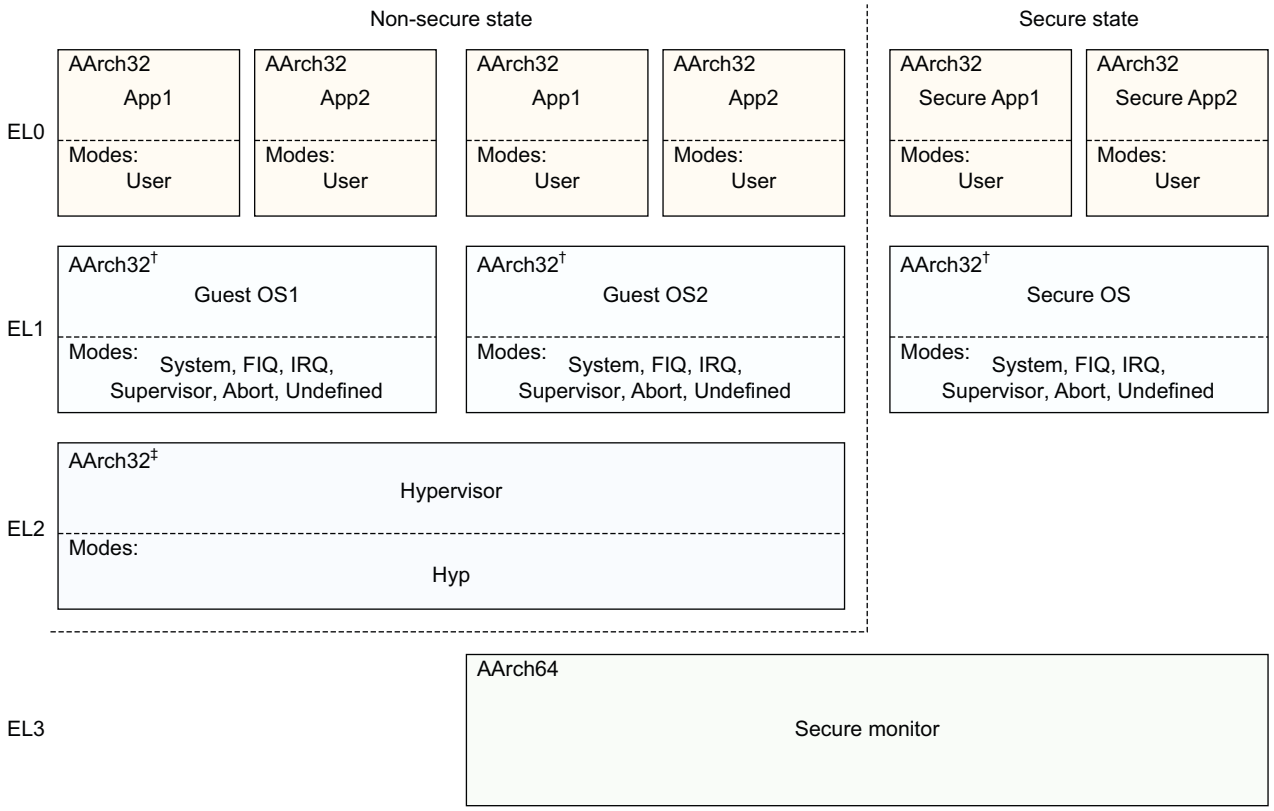
As indicated in the appropriate Mode descriptions:

- Monitor mode is a Secure mode, meaning it is always in the Secure state.
- Hyp mode is a Non-secure mode, meaning it is accessible only in Non-secure state.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### *Effect of the EL3 Execution state on the PE modes and Exception levels*

Figure G1-1 on page G1-3754 shows the PE modes, Exception levels, and Security states, for an implementation that includes all of the Exception levels, when EL3 is using AArch32. Figure G1-2 shows how the implemented modes change when EL3 is using AArch64.




† When EL1 is using AArch64, System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are not implemented
‡ When EL2 is using AArch64, Hyp mode is not implemented

**Figure G1-2 ARMv8 Exception levels, and PE modes, when EL3 is using AArch64**

Comparing Figure G1-1 on page G1-3754 and Figure G1-2 shows how, in Secure state only, the implementation of System, FIQ, IRQ, Supervisor, Abort, and Undefined mode depends on the Execution state that EL3 is using. That is, these modes are implemented as follows:

**Non-secure state**

If Non-secure EL1 is using AArch32 then System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL1. Otherwise, these modes are not implemented in Non-secure state.

**Secure state** The implementation of these modes depends on the Execution state that EL3 is using, as follows:

**EL3 using AArch64** If Secure EL1 is using AArch32 then System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL1. Otherwise, these modes are not implemented in Secure state.

**EL3 using AArch32** In Secure state, System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL3, see Figure G1-1 on page G1-3754.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Hyp mode

Hyp mode is the Non-secure EL2 mode. When EL2 is using AArch32, it provides the usual method of controlling the virtualization of Non-secure execution at EL1 and EL0.

**Note**

The alternative method of controlling this functionality is by accessing the EL2 controls from EL3 with the SCR_EL3.NS or SCR.NS bit set to 1.

This section summarizes how Hyp mode differs from the other modes, and references where this part of the manual describes the features of Hyp mode in more detail:

- Software executing in Hyp mode executes at EL2, see Figure G1-1 on page G1-3754.
- Hyp mode is accessible only in Non-secure state. In Secure state, an attempt by a `CPS` or an `MSR` instruction to change PSTATE.M to Hyp mode is an illegal change to PSTATE.M, as described in *Illegal changes to PSTATE.M* on page G1-3771.
- In Non-debug state, the only mechanisms for changing to Hyp mode are:
  - An exception taken from a Non-secure EL1 or EL0 mode.
  - When EL3 is using AArch32, an exception return from Secure Monitor mode.
  - When EL3 is using AArch64, an exception return from EL3.
- In Hyp mode, the only exception return is execution of an `ERET` instruction, see *ERET* on page F6-2683.
- In Hyp mode, the CPACR has no effect on the execution of coprocessor, floating-point, or Advanced SIMD instructions. The HCPTR controls execution of these instructions in Hyp mode.
- If software running in Hyp mode executes an `SVC` instruction, the Supervisor Call exception generated by the instruction is taken to Hyp mode, see *SVC* on page F6-3103.
- An exception return with restored PSTATE specifying Hyp mode is an *illegal return event*, as described in *Illegal return events from AArch32 state* on page G1-3797, if any of the following applies:
  - EL3 is using AArch64 and the value of SCR_EL3.NS is 0.
  - EL3 is using AArch32 and the value of SCR.NS is 0.
  - The return is from a Non-secure EL1 mode.
- The instructions described in the following sections are UNDEFINED if executed in Hyp mode:
  - SRS. See *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F6-3013.
  - RFE. See *RFE, RFEDA, RFEDB, RFEIA, RFEIB* on page F6-2915.
  - *LDM (exception return)* on page F6-2714.
  - *LDM (User registers)* on page F6-2716.
  - *STM (User registers)* on page F6-3043.
  - The `SUBS PC, LR` forms of the instructions described in *SUB, SUBS (immediate)* on page F6-3088.

    **Note**

    In T32 state, `ERET` is encoded as `SUBS PC, LR, #0`, and therefore this is a valid instruction.

  - The exception return form of the instructions described in *MOV, MOVS (register)* on page F6-2813.

  In addition, deprecated forms of the A32 `ADCS`, `ADDS`, `ANDS`, `BICS`, `EORS`, `MOVS`, `MVNS`, `ORRS`, `RSBS`, `RSCS`, `SBCS`, and `SUBS` instructions with the PC as the destination register are UNDEFINED if executed in Hyp mode. The instruction descriptions identify these UNDEFINED cases.
- The unprivileged Load unprivileged and Store unprivileged instructions `LDRT`, `LDRSHT`, `LDRHT`, `LDRBT`, `STRT`, `STRHT`, and `STRBT`, are UNPREDICTABLE if executed in Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In an implementation that includes EL3, from reset, the HVC instruction is UNDEFINED in Non-secure EL1 modes, meaning entry to Hyp mode is disabled by default. To permit entry to Hyp mode using the Hypervisor Call exception, Secure software must enable use of the HVC instruction:

- By setting the SCR_EL3.HCE bit to1, if EL3 is using AArch64.
- By setting the SCR.HCE bit to 1, if EL3 is using AArch32.

When the HVC instruction is UNDEFINED in Non-secure EL1 modes because of the value of the SCR_EL3.HCE or SCR.HCE bit, HVC is UNPREDICTABLE in Hyp mode.

### Pseudocode description of mode operations

The BadMode() function tests whether a 5-bit mode number corresponds to one of the permitted modes.

The BadMode() function is defined in Chapter J1 *ARMv8 Pseudocode*.

## G1.8.2 AArch32 general-purpose registers, the PC, and the Special-purpose registers

*The general-purpose registers, and the PC, in AArch32 state* on page E1-2316 describes the application level view of the general-purpose registers, and the PC. This view provides:

- The general-purpose registers R0-R14, of which:
  - The preferred name for R13 is SP (*stack pointer*).
  - The preferred name for R14 is LR (*link register*).
- The PC (*program counter*), that can be described as R15.

These registers are selected from a larger set of registers, that includes *Banked* copies of some registers, with the current register selected by the execution mode. The implementation and banking of the general-purpose registers depends on whether or not the implementation includes EL2 and EL3, and whether those exception levels are using AArch32. Figure G1-3 on page G1-3764 shows the full set of Banked general-purpose registers, and the Special-purpose registers:

- The Program Status Registers CPSR and SPSR.
- ELR_hyp.

**Note**

The architecture uses system level register names, such as R0_usr, R8_usr, and R8_fiq, when it must identify a specific register. The application level names refer to the registers for the current mode, and usually are sufficient to identify a register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Application level view | System level view | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | User | System | Hyp † | Supervisor | Abort | Undefined | Monitor ‡ | IRQ | FIQ |
| R0 | R0_usr | | | | | | | | |
| R1 | R1_usr | | | | | | | | |
| R2 | R2_usr | | | | | | | | |
| R3 | R3_usr | | | | | | | | |
| R4 | R4_usr | | | | | | | | |
| R5 | R5_usr | | | | | | | | |
| R6 | R6_usr | | | | | | | | |
| R7 | R7_usr | | | | | | | | |
| R8 | R8_usr | | | | | | | | R8_fiq |
| R9 | R9_usr | | | | | | | | R9_fiq |
| R10 | R10_usr | | | | | | | | R10_fiq |
| R11 | R11_usr | | | | | | | | R11_fiq |
| R12 | R12_usr | | | | | | | | R12_fiq |
| SP | SP_usr | | SP_hyp | SP_svc | SP_abt | SP_und | SP_mon | SP_irq | SP_fiq |
| LR | LR_usr | | | LR_svc | LR_abt | LR_und | LR_mon | LR_irq | LR_fiq |
| PC | PC | | | | | | | | |
| APSR | CPSR | | | | | | | | |
| | | | SPSR_hyp | SPSR_svc | SPSR_abt | SPSR_und | SPSR_mon | SPSR_irq | SPSR_fiq |
| | | | ELR_hyp | | | | | | |

‡ Part of EL3. Exists only in Secure state, and only when EL3 is using AArch32.
† Part of EL2. Exists only in Non-secure state, and only when EL2 is using AArch32.
Cells with no entry indicate that the User mode register is used.

**Figure G1-3 AArch32 general-purpose registers, PC, and Special-purpose registers, showing banking**

As described in *PE mode for taking exceptions* on page G1-3784, on taking an exception the PE changes mode, unless it is already in the mode to which it must take the exception. Each mode that the PE might enter in this way has:

- A Banked copy of the stack pointer, for example SP_irq and SP_hyp.
- A register that holds a preferred return address for the exception. This is:
  - For the EL2 mode, Hyp mode, the Special-purpose register ELR_hyp.
  - For the other privileged modes to which exceptions can be taken, a Banked copy of the link register, for example LR_und and LR_mon.
- A saved copy of PSTATE, made on exception entry, for example SPSR_irq and SPSR_hyp.

In addition FIQ mode has Banked copies of the general-purpose registers R8 to R12.

User mode and System mode share the same general-purpose registers.

User mode, System mode, and Hyp mode share the same LR.

For more information about the application level view of the SP, LR, and PC, and the alternative descriptions of them as R13, R14 and R15, see *The general-purpose registers, and the PC, in AArch32 state* on page E1-2316.

## AArch32 Special-purpose registers

In AArch32 state, the Special-purpose registers are:

- The CPSR and its view as the APSR.
- The SPSR, including the banked copies SPSR_abt, SPSR_fiq, SPSR_hyp, SPSR_irq, SPSR_mon, SPSR_svc, and SPSR_und.
- The ELR_hyp.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Pseudocode description of general-purpose register and PC operations

The following pseudocode gives access to the general-purpose registers and the PC. These registers are an array, `_R[]`, indexed by parameter `n`. This array is common to AArch32 and AArch64 operation and therefore contains 31 64-bit registers. `_PC` is the program counter, and its definition is common to AArch32 and AArch64 operation and therefore its size is 64-bit.

`LookUpRIndex()` looks up the index value, `n`, for the specified register number and PE mode, using `RBankSelect()` to evaluates the result.

`R[]` accesses the specified general-purpose register in the current PE mode, using `Rmode[]` to accesses the register, accessing `_R[]` if necessary. `SP` accesses the stack pointer, `LR` accesses the link register, and `PC` accesses the program counter. Each function has a non-assignment form for register reads and an assignment form for register writes, other than `PC`, which has only a non-assignment form.

`BranchTo()` performs a branch to the specified address.

The `_R[]`, `_PC`, `LR`, `SP`, `LookUpRIndex()`, `RBankSelect()`, `Rmode[]`, and `BranchTo()` functions are defined in Chapter J1 *ARMv8 Pseudocode*.

## G1.8.3 Saved Program Status Registers (SPSRs)

The Saved Program Status Registers (SPSRs) are used to save PE state on taking exceptions. In AArch32 state, there is an SPSR for every mode that an exception can be taken to, as shown in Figure G1-3 on page G1-3764. For example, the SPSR for Monitor mode is called SPSR_mon.

**Note**

Exceptions cannot be taken to EL0.

When the PE takes an exception, PE state is saved from PSTATE in the SPSR for the mode the exception is taken to. For example, if the PE takes an exception to Monitor mode, PE state is saved in SPSR_mon. For more information on PSTATE, see *Process state, PSTATE* on page G1-3767.

**Note**

All PSTATE fields are saved, including those which have no direct read and write access.

Saving the PSTATE fields means the exception handler can:

- On return from the exception, restore the PE state to the values it had immediately before the exception was taken. When the PE returns from an exception, PE state is restored to the state stored in the SPSR of the mode the exception is returning from, if the exception return is made using one of:
  - ERET.
  - LDM.
  - The Exception return form of the instruction described in *MOV, MOVS (register)* on page F6-2813.
  - The Exception return form of the instruction described in *SUB, SUBS (immediate)* on page F6-3088.

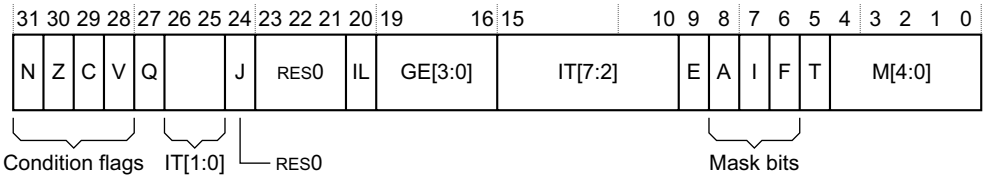
  For example, on returning from Monitor mode, PE state is restored to the state stored in SPSR_mon. If the exception return is made using the RFE instruction, the PE restores the PE state from an SPSR valued read from memory.
- Examine the value that PSTATE had when the exception was taken, for example to determine the instruction set state and privilege level in which the instruction that caused an Undefined Instruction exception was executed.

The SPSRs are UNKNOWN on reset. Any operation in a Non-secure EL1 or EL0 mode makes SPSR_hyp UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### SPSR format for exceptions taken to AArch32 state

The SPSR bit assignments are:




**N, Z, C, V, Q, GE[3:0], bits[31:27, 19:16]**
Shows the value of the PSTATE.{N, Z, C, V, Q, GE} flags immediately before the exception was taken.

**IT[1:0], J, IL, IT[7:2], E, T, bits[26:24, 20, 15:9, 5]**
Shows the values of the PSTATE.{IT[7:0], J, IL, E, T} PE state controls immediately before the exception was taken. The J bit is RES0.

**Bits[23:21]** Reserved, RES0.

**A, I, F, M, bits[8:6, 4:0]**
Shows the values of the PSTATE.{A, I, F} exception mask and PSTATE.M mode bits immediately before the exception was taken.

Bits[23:21] of an SPSR are ignored on an exception return from AArch32 state.

### Pseudocode description of SPSR operations

The following pseudocode gives access to the SPSRs.

The `SPSR[]` function accesses the current SPSR and is common to AArch32 and AArch64 operation.

The `SPSRWriteByInstr()` function is used by the MSR (register) and MSR (immediate) instructions to update the current SPSR.

The `SPSR[]` and `SPSRWriteByInstr()` functions are defined in Chapter J1 *ARMv8 Pseudocode*.

## G1.8.4 ELR_hyp

Hyp mode does not provide its own Banked copy of LR. Instead, on taking an exception to Hyp mode, the preferred return address is stored in ELR_hyp, a 32-bit Special-purpose register implemented for this purpose.

ELR_hyp can be accessed explicitly only by executing:

- An `MRS` or `MSR` instruction that targets ELR_hyp, see:
  - *MRS (Banked register)* on page F6-2832.
  - *MSR (Banked register)* on page F6-2835.

The `ERET` instruction uses the value in ELR_hyp as the return address for the exception. For more information, see *ERET* on page F6-2683.

Software execution in any Non-secure EL1 or EL0 mode makes ELR_hyp UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.9 Process state, PSTATE

In the ARMv8-A architecture, Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

PSTATE includes all of the following:

- Fields that are meaningful only in AArch32 state.
- Fields that are meaningful only in AArch64 state.
- Fields that are meaningful in both Execution states.

PSTATE is defined in pseudocode as the `PSTATE` structure, of type `ProcState`. `ProcState` is defined in Chapter J1 *ARMv8 Pseudocode*.

The PSTATE fields that are meaningful in AArch32 state are:

**The condition flags**

**N** Negative Condition flag.

**Z** Zero Condition flag.

**C** Carry Condition flag.

**V** Overflow Condition flag.

*Process state, PSTATE* on page E1-2319 gives more information about these.

**The overflow or saturation flag**

**Q** See *Process state, PSTATE* on page E1-2319.

**The greater than or equal flags**

**GE[3:0]** See *Process state, PSTATE* on page E1-2319.

**The PE state controls**

**J, T** Instruction set state. See *Process state, PSTATE* on page E1-2319. J is RES0. On a reset to AArch32 state, T is set to an IMPLEMENTATION DEFINED value. On taking an exception to:

- A PL1 mode using AArch32, T is set to SCTLR.TE.
- EL2 using AArch32, T is set to HSCTLR.TE.

**IT[7:0]** IT block state bits. See *Process state, PSTATE* on page E1-2319. On a reset or taking an exception to AArch32 state, these bits are set to 0.

**E** Endianness of data accesses. See *Process state, PSTATE* on page E1-2319. If an implementation provides both Big-endian and Little-endian support, then:

- On a reset to AArch32 state this bit is set to the IMPLEMENTATION DEFINED reset value of:
  - SCTLR.EE if the highest implemented Exception level is not EL2.
  - HSCTLR.EE if the highest implemented Exception level is EL2.
- On taking an exception to:
  - A PL1 mode using AArch32, this bit is set to SCTLR.EE.
  - EL2 using AArch32, this bit is set to HSCTLR.EE

**IL** Illegal Execution state bit. See *The Illegal Execution state exception* on page G1-3799. On a reset or taking an exception to AArch32 state, this bit is set to 0.

For information on how the J, T, IT[7:0], E, and IL fields can be accessed, see *Accessing the PE state controls and the Execution state bit* on page G1-3770.

**The asynchronous exception mask bits**

**A** Asynchronous abort mask bit.

**I** IRQ interrupt mask bit.

**F** FIQ interrupt mask bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For each bit, the values are:

**0** Exception not masked.

**1** Exception masked.

On a reset to AArch32 state, these bits are set to 1.

On taking an exception to AArch32 state, one or more of these bits are set to 1.

For more information, see both:

- *Asynchronous exception masking controls* on page G1-3804.
- *PE state on exception entry* on page G1-3788.

**The mode bits**

**M[4:0]** Current mode of the PE. Table G1-2 on page G1-3758 lists the permitted values of this field. All other values are reserved. *Illegal changes to PSTATE.M* on page G1-3771 describes the effect of setting M[4:0] to a reserved value.

M[4] is:

**M[4], Execution state**

The current Execution state:

**0** AArch64 state.

**1** AArch32 state.

**Note**

This is consistent with the use of M[4:0] in previous versions of the architecture.

On a reset to AArch32 state, M[4:0] is set to:

- `0b10011`, meaning Supervisor mode, if the highest implemented Exception level is not EL2.
- `0b11010`, meaning Hyp mode, if the highest implemented Exception level is EL2.

On taking an exception to AArch32 state, M[4:0] is set to the target mode for the exception type.

For more information about the PE modes, see:

- *AArch32 PE mode descriptions* on page G1-3758.
- *PE state on exception entry* on page G1-3788.

## G1.9.1 Accessing PSTATE fields

The PSTATE fields can be accessed as described in the following subsections:

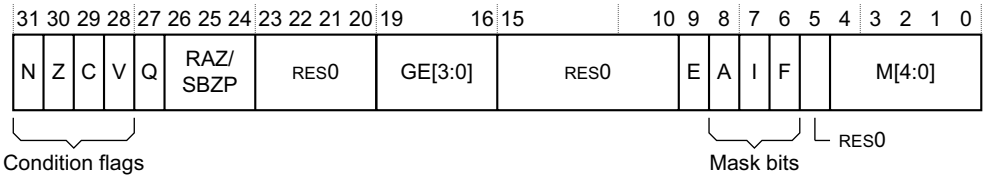
- *The Current Program Status Register, CPSR*.
- *Accessing the PE state controls and the Execution state bit* on page G1-3770.
- *The CPS instruction* on page G1-3770.
- *The SETEND instruction* on page G1-3770.

### The Current Program Status Register, CPSR

Some PSTATE fields can be accessed using the Special-purpose *Current Program Status Register* (CPSR). The CPSR can be directly read using the MRS instruction, and directly written using the MSR (register) and MSR (immediate) instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The CPSR bit assignments are:




**N, Z, C, V, bits [31:28]**

The PSTATE condition flags.

**Q, bit [27]** The PSTATE overflow or saturation flag.

**Bits[26:24]** Reserved, RAZ/SBZP. Software can use `MSR` instructions that write the top byte of the CPSR without using a read-modify-write sequence. If it does this, it must write zeros to bits[26:24].

**Bits[23:20, 15:10, 5]**

Reserved, RES0.

**GE[3:0], bits [19:16]**

The PSTATE greater than or equal flags.

**E, bit [9]** The PSTATE endianness bit.

**A, I, F, bits [8:6]**

The PSTATE asynchronous exception mask bits.

**M[4:0], bits [4:0]**

The PSTATE mode bits.

The other PSTATE fields cannot be accessed by using the CPSR. For information on how to access them, see *Accessing the PE state controls and the Execution state bit* on page G1-3770.

The application level alias for the CPSR is the APSR. The APSR is a subset of the CPSR. See *The Application Program Status Register, APSR* on page E1-2322.

Writes to the CPSR have side-effects on various aspects of PE operation. All of these side-effects, except side-effects on memory accesses associated with fetching instructions, are synchronous to the CPSR write. This means that they are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

The privilege level and address space of memory accesses associated with fetching instructions depend on the current Exception level and Security state. Writes to PSTATE.M can change one or both of the Exception level and Security state. The effect, on memory accesses associated with fetching instructions, of a change of Exception level or Security state is:

- Synchronous to the change of Exception level or Security state, if that change is caused by an exception entry or exception return.
- Guaranteed not to be visible to any memory access caused by fetching an earlier instruction in the execution stream.
- Guaranteed to be visible to any memory access caused by fetching any instruction after the next Context synchronization operation in the execution stream.
- Might or might not affect memory accesses caused by fetching instructions between the mode change instruction and the point where the mode change is guaranteed to be visible.

See *Exception return to an Exception level using AArch32* on page G1-3796 for the definition of exception return instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PE state controls and the Execution state bit

The PE state controls are the PSTATE.{IL, IT[7:0], J, E, T} fields. Software can read or write these in an SPSR.

In the CPSR:

- The PE state controls, other than PSTATE.E, are RAZ when read by an MRS instruction.
- Writes to the PE state controls, other than PSTATE.E, by MSR (register) or MSR (immediate), are ignored in all modes.

Instructions other than MRS, MSR (register), or MSR (immediate) that access the PE state controls can read and write them in any mode.

Unlike the other PSTATE PE state controls, PSTATE.E can be read by an MRS instruction and might be written by MSR (register) or MSR (immediate). However, ARM deprecates PSTATE.E having a different value from the equivalent System register EE bit, see *Mixed-endian support* on page G3-3950.

**Note**

To determine the current endianness, software can use an `LDR` instruction to load a word from memory with a known value that differs if the endianness is reversed. For example, using an `LDR` instruction to load a word whose four bytes are `0x01`, `0x00`, `0x00`, and `0x00` in ascending order of memory address loads the destination register with:

- `0x00000001` if the current endianness is little-endian.
- `0x01000000` if the current endianness is big-endian.

The PSTATE.M[4] bit is the Execution state bit. When read by an MRS instruction in AArch32 state, this bit always reads as 1. When written by an MSR (register) instruction or MSR (immediate) instruction, writing a value other than 1 is an illegal change to the PSTATE.M field. See *Illegal changes to PSTATE.M* on page G1-3771.

### The CPS instruction

The A32 and T32 instruction sets both include an instruction to manipulate PSTATE.{A, I, F} and PSTATE.M:

`CPSIE <iflags> {, #<mode>}`
: Sets the specified PSTATE. {A, I, F} exception masks to 0, enabling the exception, and optionally changes to the specified mode.

`CPSID <iflags> {, #<mode>}`
: Sets the specified PSTATE.{A, I, F} exception masks to 1, disabling the exception, and optionally changes to the specified mode.

`CPS #<mode>`
: Changes to the specified mode without affecting the PSTATE.{A, I, F} exception masks.

The `CPS` instruction is unconditional. For more information, see *CPS, CPSID, CPSIE* on page F6-2662.

### The SETEND instruction

The A32 and T32 instruction sets both include an instruction to manipulate PSTATE.E:

`SETEND BE`
: Sets PSTATE.E to 1, for big-endian operation.

`SETEND LE`
: Sets PSTATE.E to 0, for little-endian operation.

The `SETEND` instruction is unconditional. For more information, see *SETEND* on page F6-2962. ARM deprecates use of the `SETEND` instruction.

## G1.9.2 The Saved Program Status Registers (SPSRs)

On taking an exception, PSTATE is preserved in the SPSR of the mode to which the exception is taken. The SPSRs are described in *Saved Program Status Registers (SPSRs)* on page G1-3765.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G1.9.3 Illegal changes to PSTATE.M

In AArch32 PE modes other than User mode, MSR and CPS instructions can explicitly change PSTATE.M. The following changes to PSTATE.M by MSR or CPS instructions are illegal:

- A change to an encoding that Table G1-2 on page G1-3758 does not show.
- A change to a mode that is not implemented.
- A change to a mode that is not accessible from the context the MRS or CPS instruction is executed in, as follows:
  - A change to a mode that would cause entry to a higher Exception level.
  - When executing in Non-secure state, a change to Monitor mode.
  - When executing in Secure EL1, a change to Monitor mode when EL3 is using AArch64.
  - A change to Hyp mode from any other mode.
  - A change from Hyp mode to any other mode.
  - When the value of HCR.TGE is 1, attempting to change from Monitor mode to a Non-secure PL1 mode, see *Trapping of general exceptions to Hyp mode* on page K1-5499.

On executing an instruction that attempts an illegal change to PSTATE.M:

- PSTATE.M is unchanged, and the current mode remains unchanged.
- PSTATE.IL is set to 1.
- All other PSTATE fields are written to as normal.

**Note**

For the PSTATE fields that MSR and CPS instructions update, see the instruction descriptions:

- *MSR (register)* on page F6-2840.
- *MSR (immediate)* on page F6-2838.
- *CPS, CPSID, CPSIE* on page F6-2662.

When the value of PSTATE.IL is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception* on page G1-3799.

**Note**

- The PE ignores writes to PSTATE.M when executing at PL0.
- In ARMv7, an instruction that attempts to make an illegal change to PSTATE.M is UNPREDICTABLE.

### G1.9.4 Pseudocode description of PSTATE operations

The `CPSRWriteByInstr()` function is used by the MSR (register) and MSR (immediate) instructions to update PSTATE.

The `SetPSTATEFromPSR()` function updates PSTATE from a CPSR or SPSR.

Chapter J1 *ARMv8 Pseudocode* defines of these functions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.10 Instruction set states

The instruction set states are described in Chapter E2 *The AArch32 Application Level Memory Model* and application level operations on them are described there. This section supplies more information about how they interact with system level functionality, in the sections:

- *Exceptions and instruction set state*.
- *Unimplemented instruction sets*.

### G1.10.1 Exceptions and instruction set state

If an exception is taken to an EL1 mode, the SCTLR.TE bit for the Security state the exception is taken to determines the instruction set state that handles the exception, and if necessary, the PE changes to this instruction set state on exception entry.

If the exception is taken to Hyp mode, the HSCTLR.TE bit determines the instruction set state that handles the exception, and if necessary, the PE changes to this instruction set state on exception entry.

On coming out of reset, if the highest implemented Exception level is using AArch32:

- If the highest implemented Exception level is EL2, the PE starts execution in Hyp mode, in the instruction set state determined by the reset value of HSCTLR.TE.
- Otherwise, the PE starts execution in Supervisor mode, in the instruction set state determined by the reset value of SCTLR.TE. If the implementation includes EL3, this execution is in Secure Supervisor mode.

For more information about exception entry, see *Overview of exception entry* on page G1-3781.

### G1.10.2 Unimplemented instruction sets

The PSTATE.T bit defines the current instruction set state, see *Process state, PSTATE* on page E1-2319.

In the ARMv8 architecture there is no support for the hardware acceleration of Java bytecodes, and the Jazelle Instruction set state is obsolete. Every AArch32 implementation must support the Trivial Jazelle implementation described in *Trivial implementation of the Jazelle extension*.

**Note**

In previous versions of the ARM architecture, the PSTATE.{J, T} bits determined the Instruction set state. In ARMv8, PSTATE.J is RES0.

#### Trivial implementation of the Jazelle extension

ARMv8 requires that the implementation of AArch32 state includes the trivial Jazelle implementation.

In a trivial implementation of the Jazelle extension:

- At EL1, EL2, or EL3, if the Exception level is using AArch32:
  - The JMCR and JOSCR are RAZ/WI.
  - The JIDR is a RAZ read-only register.
- At EL0 when EL0 is using AArch32:
  - It is IMPLEMENTATION DEFINED whether the JMCR and JOSCR are RAZ/WI or UNDEFINED.
  - It is IMPLEMENTATION DEFINED whether JIDR is RAZ or UNDEFINED.
- The `BXJ` instruction behaves identically to the `BX` instruction in all circumstances.

  **Note**

  This is consistent with the JMCR.JE bit being RAZ, and means that the A32 and T32 instruction sets do not provide any mechanism for attempting to enter Jazelle state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Jazelle state, as defined in previous versions of the ARM architecture, is an unimplemented instruction set state.

These requirements ensure that operating systems that support an EJVM execute correctly.

A trivial implementation is not required to extend the PC to 32 bits, that is, it can implement PC[0] as RAZ/WI.

**Note**

This is because the only way that PC[0] is visible in A32 or T32 state is as a result of an exception occurring during Jazelle state execution, and Jazelle state execution cannot occur on a trivial implementation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.11 Handling exceptions that are taken to an Exception level using AArch32

An exception causes the PE to suspend program execution to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. Exceptions can be generated by internal and external sources.

Normally, when an exception is taken the PE state is preserved immediately, before handling the exception. This means that, when the event has been handled, the original state can be restored and program execution resumed from the point where the exception was taken.

More than one exception might be generated at the same time, and a new exception can be generated while the PE is handling an exception.

The following sections describe exception handling:

- *Exception vectors and the exception base address*.
- *Exception prioritization for exceptions taken to AArch32 state* on page G1-3778.
- *Overview of exception entry* on page G1-3781.
- *PE mode for taking exceptions* on page G1-3784.
- *PE state on exception entry* on page G1-3788.
- *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.
- *Routing debug exceptions to EL2* on page G1-3795.
- *Exception return to an Exception level using AArch32* on page G1-3796.

*Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-3801 gives a full description of asynchronous exception handling, for exceptions taken asynchronously from AArch32 state.

**Note**

Because of the common model for handling exceptions, the current section requires some understanding of the asynchronous exception behaviors described in *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-3801.

*AArch32 state exception descriptions* on page G1-3811 then describes each exception.

### G1.11.1 Exception vectors and the exception base address

When an exception is taken, PE execution is forced to an address that corresponds to the type of exception. This address is called the *exception vector* for that exception. The vectors for the different types of exception form a *vector table*.

**Note**

There are significant differences in the sets of exception vectors for exceptions taken to an Exception level that is using AArch32 and for exceptions taken to an Exception level that is using AArch64. This part of this manual describes only how exceptions are taken to an Exception level that is using AArch32. So, for example, when executing at EL1 or EL0, an exception might be generated that must be taken to EL3. In this case:

- If EL3 is using AArch32 then the exception is taken as described in this chapter, using the exception vectors described in this section.
- If EL3 is using AArch64 then the exception is taken as described in Chapter D1 *The AArch64 System Level Programmers' Model* using the exception vectors described in *Exception vectors* on page D1-1538.

AArch32 state defines exception vector tables for exceptions taken to EL2 and EL3 when those Exception levels are using AArch32. Those vector tables are not used when the corresponding Exception levels are using AArch64.

A set of exception vectors for an Exception level that is using AArch32 comprises eight consecutive word-aligned memory addresses, starting at an *exception base address*. These eight vectors form an AArch32 *vector table*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The number of possible exception base addresses, and therefore the number of vector tables, depends on the implemented Exception levels, as follows:

**Implementation that does not include EL3**

Any implementation that does not include EL3 must include the following AArch32 vector table if EL1 can use AArch32:

- An exception table for exceptions taken to EL1 modes other than System mode. This is the EL1 vector table, and is in the address space of the PL1&0 translation regime.

  **Note**

  Exceptions cannot be taken to System mode.

  For this vector table, the VBAR holds the exception base address.

**Implementation that includes EL2**

Any implementation that includes EL2 must include the following additional AArch32 vector table if EL2 can use AArch32:

- An exception table for exceptions taken to Hyp mode. This is the Hyp vector table, and is in the address space of the Non-secure PL2 translation regime.

  For this vector table, HVBAR holds the exception base address.

**Implementation that includes EL3**

Any implementation that includes EL3 must include the following AArch32 vector tables:

- If EL3 can use AArch32, a vector table for exceptions taken to Secure Monitor mode. This is the Monitor vector table, and is in the address space of the Secure PL1&0 translation regime.

  For this vector table, MVBAR holds the exception base address.

- If Secure EL1 can use AArch32, a vector table for exceptions taken to Secure privileged modes other than Monitor mode and System mode. This is the Secure vector table, and is in the address space of the Secure PL1&0 translation regime.

  For this vector table, the Secure VBAR holds the exception base address.

- If Non-secure EL1 can use AArch32, a vector table for exceptions taken to Non-secure PL1 modes. This is the Non-secure vector table, and is in the address space of the Non-secure PL1&0 translation regime.

  For this vector table, the Non-secure VBAR holds the exception base address.

The following subsections give more information:

- *The vector tables and exception offsets*.
- *Pseudocode determination of the exception base address* on page G1-3777.

## The vector tables and exception offsets

Table G1-3 on page G1-3776 defines the AArch32 vector table entries. In this table:

- The *Hyp mode* column defines the vector table entries for exceptions taken to Hyp mode.
- The *Monitor mode* column defines the vector table entries for exceptions taken to Monitor mode.
- The *Secure* and *Non-secure* columns define the Secure and Non-secure vector table entries, that are used for exceptions taken to modes other than Monitor mode, Hyp mode, System mode, and User mode. Table G1-4 on page G1-3776 shows the mode to which each of these exceptions is taken. Each of these modes is described as the *default* mode for taking the corresponding exception.

  **Note**

  Exceptions cannot be taken to System mode or User mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about determining the mode to which an exception is taken, see *PE mode for taking exceptions* on page G1-3784.

When EL2 is using AArch32, it provides a number of additional exceptions, some of which are not shown explicitly in the vector tables. For more information, see *Offsets of AArch32 exceptions provided by EL2* on page G1-3777.

**Table G1-3 The AArch32 vector tables**

| Offset | Vector tables | | | |
|---|---|---|---|---|
| | Hyp[a] | Monitor[b] | Secure[c] | Non-secure[c] |
| 0x00 | Not used | Not used | Not used[d] | Not used |
| 0x04 | Undefined Instruction, from Hyp mode | Monitor Trap | Undefined Instruction | Undefined Instruction |
| 0x08 | Hypervisor Call, from Hyp mode | Secure Monitor Call | Supervisor Call | Supervisor Call |
| 0x0C | Prefetch Abort, from Hyp mode | Prefetch Abort | Prefetch Abort | Prefetch Abort |
| 0x10 | Data Abort, from Hyp mode | Data Abort | Data Abort | Data Abort |
| 0x14 | Hyp Trap, or Hyp mode entry[e] | Not used | Not used | Not used |
| 0x18 | IRQ interrupt | IRQ interrupt | IRQ interrupt | IRQ interrupt |
| 0x1C | FIQ interrupt | FIQ interrupt | FIQ interrupt | FIQ interrupt |

a. Non-secure state only. Implemented only if the implementation includes EL2 and EL2 can use AArch32.
b. Secure state only. Implemented only if the implementation includes EL3 and EL3 can use AArch32.
c. If the implementation does not include EL3 then there is a single vector table for exceptions taken to EL1 when EL1 is using AArch32. That table holds the vectors shown in the Secure column of this table
d. In previous versions of the architecture, this entry has been used for the Reset vector, meaning the address at which execution starts on coming out of reset. In ARMv8, the AArch32 Reset vector is IMPLEMENTATION DEFINED. An implementation might use this vector table entry to hold the Reset vector.
e. See *Use of offset 0x14 in the Hyp vector table* on page G1-3777.

**Table G1-4 Modes for taking the exceptions shown in the Secure or Non-secure vector table**

| Exception | Mode taken to |
|---|---|
| Undefined Instruction | Undefined |
| Supervisor Call | Supervisor |
| Prefetch Abort | Abort |
| Data Abort | Abort |
| IRQ interrupt | IRQ |
| FIQ interrupt | FIQ |

For more information about use of the vector tables see *Overview of exception entry* on page G1-3781.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Offsets of AArch32 exceptions provided by EL2

EL2 provides the following exceptions. When EL2 is using AArch32, these exceptions are taken to Hyp mode, and the PE enters the handlers for these exceptions using the following vector table entries shown in Table G1-3 on page G1-3776:

**Hypervisor Call**
: If taken from Hyp mode, shown explicitly in the Hyp mode vector table. Otherwise, see *Use of offset 0x14 in the Hyp vector table*.

**Hyp Trap**
: Shown explicitly in the Hyp mode vector table.

**Virtual Abort**
: Entered through the Data Abort vector in the Non-secure vector table.

**Virtual IRQ**
: Entered through the IRQ vector in the Non-secure vector table.

**Virtual FIQ**
: Entered through the FIQ vector in the Non-secure vector table.

**Note**

*Virtual exceptions when an implementation includes EL2* on page G1-3801 gives more information about the virtual exceptions.

### Use of offset `0x14` in the Hyp vector table

The vector at offset `0x14` in the Hyp vector table is used for exceptions that cause entry to Hyp mode. This means it is:

- Always used for the Hyp Trap exception.
- Used for any Hypervisor Call exception that is taken from a mode other than Hyp mode.
- Used for any Supervisor Call exception that is taken from Non-secure User mode when the value of HCR.TGE is 1.
- Used for any Undefined Instruction that is taken from Non-secure User mode when the value of HCR.TGE is 1.
- Used for any Prefetch Abort exception that is:
  - Taken from Non-secure User mode when the value of HCR.TGE is 1.
  - Generated by a Debug exception from Non-secure state when the value of HDCR.TDE is 1.
  - Generated by a stage 2 abort on an address translation instruction.
- Used for any Data Abort exception that is:
  - Taken from Non-secure User mode when the value of HCR.TGE is 1.
  - Generated by an asynchronous External abort from Non-secure state when the value of HCR.AMO is 1.
  - Generated by a Watchpoint exception from Non-secure state when the value of HDCR.TDE is 1.
  - Generated by a stage 2 abort on an address translation operation.

**Note**

Offset `0x14` is never used for IRQ exceptions, Virtual IRQ exceptions, FIQ exceptions, or Virtual FIQ exceptions.

For more information, see *PE mode for taking exceptions* on page G1-3784.

### Pseudocode determination of the exception base address

For an exception taken to a PL1 mode, the `ExcVectorBase()` function determines the exception base address.

The `ExcVectorBase()` function is defined in Chapter J1 *ARMv8 Pseudocode*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The PL1 modes to which exceptions can be taken are Supervisor mode, Undefined mode, Abort mode, IRQ mode, and FIQ mode. In Non-secure state, and in Secure state when EL3 is using AArch64, these are EL1 modes. However, in Secure state when EL3 is using AArch32, these are EL3 modes. For more information see *Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990.

### G1.11.2 Exception prioritization for exceptions taken to AArch32 state

The following sections describe the ARMv8 requirements for the prioritization of synchronous exceptions, and the limits on when asynchronous exceptions can be taken:

- *Synchronous exception prioritization for exceptions taken to AArch32 state*.
- *Architectural requirements for taking asynchronous exceptions* on page G1-3780.

See also:

- *AArch32 prioritization of synchronous aborts from a single stage of address translation* on page G4-4088, for information about:
  - The prioritization of aborts on a single memory access in a VMSA implementation.
  - The prioritization of exceptions generated during address translation

  .
- *Debug state entry and debug event prioritization* on page H2-4823 for information about the relative prioritization of exceptions and the debug events that cause entry to Debug state.

#### Synchronous exception prioritization for exceptions taken to AArch32 state

In principle, any single instruction can generate a number of different synchronous exceptions, between the fetching of the instruction, its decode, and eventual execution. This section describes the prioritization of such exceptions when they are taken to an Exception level that is using AArch32.

**Note**

- An exception that is taken to an Exception level that is using AArch32 must have been taken from an Exception level that is using AArch32.
- The priority numbering in this list only shows the relative priorities of exceptions taken to an Exception level that is using AArch32. This numbering has no global significance and, for example, does not correlate with the equivalent AArch32 list in *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

For an exception that is taken to an Exception level that is using AArch32, exceptions are prioritized as follows, where 1 is the highest priority.

1. Misaligned PC exceptions. A Misaligned PC exception can only be taken to an Exception level that is using AArch32 as a result of:
   - The CONSTRAINED UNPREDICTABLE handling of a branch to an unaligned address, see *Branching to an unaligned PC* on page K1-5424.
   - Exiting from Debug state to AArch32 specifying an unaligned PC value, see *Exiting Debug state* on page H2-4858.

   A Misaligned PC exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception, see *Prefetch Abort exception reporting a Misaligned PC exception* on page G1-3819.
2. Prefetch Abort exceptions. See *Prefetch Abort exception* on page G1-3818 and *AArch32 prioritization of synchronous aborts from a single stage of address translation* on page G4-4088.
3. Breakpoint exceptions or Address Matching Vector Catch exceptions. See:
   - *Breakpoint exceptions* on page G2-3898.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Vector Catch exceptions* on page G2-3937.

**Note**

An Exception Trapping Vector Catch exception is generated on exception entry for an exception that has been prioritized as described in this section. This means that it does not have its own entry in this list.

4. Illegal Execution state exceptions. See *The Illegal Execution state exception* on page G1-3799.
5. Exceptions taken from EL1 to EL2 because of one of the following configuration settings:
   - HSTR.T*n*.
   - HCR.TIDCP.
6. Undefined Instruction exceptions that occur as a result of one or more of the following:
   - An attempt to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation.
   - An attempt to execute an instruction that is defined never to be accessible at the current Exception level regardless of any enables or traps.
   - Debug state execution of an instruction encoding that is unallocated in Debug state.
   - Non-debug state execution of an instruction encoding that is unallocated in Non-debug state.
   - Execution of an `HVC` instruction, when `HVC` instructions are disabled by SCR.HCE or HCR.HCD.
   - Execution of an `HLT` instruction when `HLT` instructions are disabled by EDSCR.HDE.
   - In Debug state:
     — Execution of a `DCPS1` instruction in Non-secure EL0 when HCR.TGE is 1.
     — Execution of a `DCPS2` instruction in EL1 or EL0 when SCR.NS is 0 or when EL2 is not implemented.
     — Execution of a `DCPS3` instruction when EDSCR.SDD is 1 or when EL3 is not implemented.
     — When the value of EDSCR.SDD is 1, execution in EL2, EL1, or EL0 of an instruction that is trapped to EL3.
   - Execution of an instruction that is UNDEFINED as a result of any of:
     — Being in an IT block when SCTLR.ITD is 1, or when HSCTLR.ITD is 1.
     — Executing a `SETEND` instruction when SCTLR.SED is 1, or when HSCTLR.SED is 1.
     — Executing a CP15DMB, CP15DSB, or CP15ISB barrier instruction when SCTLR.CP15BEN is 0, or when HSCTLR.CP15BEN is 0.

     See *Disabling or enabling PL0 and PL1 use of AArch32 deprecated functionality* on page G1-3849 and *Disabling or enabling EL2 use of AArch32 deprecated functionality* on page G1-3857.
   - Execution of an instruction that is UNDEFINED because at least one of FPSCR.{Stride, Len} is nonzero, when programming these bits to nonzero values is supported. See *Floating-point exception traps* on page G1-3844.
7. Exceptions taken to EL1, or taken to EL2 because the value of HCR.TGE is 1, that are generated because of configurable access to instructions, and that are not covered by any of priorities 1-6.
8. Exceptions taken from EL0 to EL2 because of one of the following configuration settings:
   - HSTR.T*n*.
   - HCR.TIDCP.
9. Exceptions taken to EL2 because of configuration settings in the HCPTR.
10. Exceptions taken to EL2 because of one of the following configuration settings:
    - Any setting in HCR, other than the TIDCP bit.
    - Any setting in CNTHCTL.
    - Any setting in HDCR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

11. Exceptions taken to EL2 because of configurable access to instructions, and that are not covered by any of priorities 1-10.

12. Exceptions caused by the SMC instruction being UNDEFINED because the value of SCR.SCD is 1.

13. Exceptions caused by the execution of an Exception generating instruction, SVC, HVC, SMC, or BKPT.

14. Exceptions taken to EL3 because of configuration settings in the SDCR. These might be taken from EL0, EL1, or EL2.

15. Exceptions taken to EL3 because of configurable access to instructions, and that are not covered by any of priorities 1-14.

16. Trapped floating-point exceptions, if supported. See *Floating-point exception traps* on page G1-3844.

17. Data Abort exceptions other than a Data Abort exception generated by a Synchronous external abort that was not generated by a translation table walk. That is, any Data Abort exception that is not covered by item 23. See *Data Abort exception* on page G1-3821 and *AArch32 prioritization of synchronous aborts from a single stage of address translation* on page G4-4088. It is IMPLEMENTATION DEFINED whether Synchronous external aborts are prioritized here or as item 19.

18. Watchpoint exceptions. See *Watchpoint exceptions* on page G2-3923.

19. Data Abort exception generated by a Synchronous external abort that was not generated by a translation table walk, see *External aborts* on page G3-3976. It is IMPLEMENTATION DEFINED whether Synchronous external aborts are prioritized here or as item 17.

For items 17-19, if an instruction results in more than one single-copy atomic memory access, the prioritization between synchronous exceptions generated on each of those different memory accesses is not defined by the architecture.

**Note**

Exceptions generated by a translation table walk are reported and prioritized as either a Prefetch Abort exception, priority 2 in this list, or a Data Abort exception, priority 17 in this list. See also *AArch32 prioritization of synchronous aborts from a single stage of address translation* on page G4-4088.

### Architectural requirements for taking asynchronous exceptions

The ARM architecture does not define when asynchronous exceptions are taken, but sets the following limits on when they are taken:

- An asynchronous exception that is pending before one of the following context synchronizing events is taken before the first instruction after the context synchronizing event completes its execution, provided that the pending asynchronous event is not masked after the context synchronizing event. The context synchronizing events are:
  - Execution of an ISB instruction.
  - Taking an exception.
  - Return from an exception.
  - Exit from Debug state.

  The ISR identifies any pending asynchronous exceptions.

  **Note**

  If the first instruction after the context synchronizing event generates a synchronous exception, then the architecture does not define the order in which that synchronous exception and the asynchronous exception are taken.

- In the absence of a specific requirement to take an asynchronous exception because of a context synchronizing event, the only requirement of the architecture is that an unmasked asynchronous exception is taken in finite time.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The taking of an unmasked asynchronous exception in finite time must occur with all code sequences, including with a sequence that consists of unconditional loops.

Within these limits, the prioritization of asynchronous exceptions relative to other exceptions, both synchronous and asynchronous, is IMPLEMENTATION DEFINED.

PSTATE includes a mask bit for each type of asynchronous exception. Setting one of these bits to 1 can prevent the corresponding asynchronous exception from being taken, although when the PE is in Non-secure state other controls can modify the effect of these bits. For more information, see *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-3801.

Taking an exception sets an exception-dependent subset of these mask bits.

**Note**

In some contexts, the PSTATE.{A, I, F} bits mask the taking of asynchronous exceptions. The way these are set on exception entry, described in *PSTATE.{A, I, F, M} values on exception entry* on page G1-3789, can prevent an exception handler being interrupted by an asynchronous exception.

### G1.11.3 Overview of exception entry

There are some significant differences between the handling of exceptions taken to Hyp mode and exceptions taken to other modes. Because Hyp mode is the EL2 mode, this means that the following descriptions sometimes distinguish between the *EL2 mode* and the *non-EL2 modes*.

On taking an exception to an Exception level that is using AArch32:

1. The hardware determines the mode to which the exception must be taken, see *PE mode for taking exceptions* on page G1-3784.
2. A link value, indicating the *preferred return address* for the exception, is saved. This is a possible return address for the exception handler, and depends on:
   - The exception type.
   - Whether the exception is taken to the EL2 mode or to a non-EL2 mode.
   - For some exceptions taken to non-EL2 modes, the instruction set state when the exception was taken.

   Where the link value is saved depends on whether the exception is taken to the EL2 mode.

   For more information see *Link values saved on exception entry* on page G1-3783.
3. The value of PSTATE is saved in the SPSR for the mode to which the exception must be taken. The value saved in SPSR.IT[7:0] is always correct for the preferred return address.
4. In an implementation that includes EL3, when EL3 is using AArch32:
   - If the exception is taken from Monitor mode, SCR.NS is cleared to 0.
   - Otherwise, taking the exception leaves SCR.NS unchanged.

   When EL3 is using AArch64, Monitor mode is not available.
5. PSTATE is updated with new context information for the exception handler. This includes:
   - Setting PSTATE.M to the PE mode to which the exception is taken.
   - Setting the appropriate PSTATE mask bits. This can disable the corresponding exceptions, preventing uncontrolled nesting of exception handlers.
   - Setting the instruction set state to the state required for exception entry.
   - Setting the endianness to the required value for exception entry.
   - Clearing the PSTATE.IT[7:0] bits to 0.

   For more information, see *PE state on exception entry* on page G1-3788.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

6. The appropriate exception vector is loaded into the PC, see *Exception vectors and the exception base address* on page G1-3774.

7. Execution continues from the address held in the PC.

For an exception taken to a non-EL2 mode, on exception entry, the exception handler can use the SRS instruction to store the return state onto the stack of any mode at the same Exception level and in the same Security state, and can use the CPS instruction to change mode. For more information about the instructions, see *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F6-3013 and *CPS, CPSID, CPSIE* on page F6-2662.

Later sections of this chapter describe each of the possible exceptions, and each of these descriptions includes a pseudocode description of the PE state changes on taking that exception. Table G1-5 gives an index to these descriptions:

**Table G1-5 Pseudocode descriptions of exception entry for exceptions taken to AArch32 state**

| Exception | Description of exception entry |
|---|---|
| Reset | *Pseudocode descriptions of reset* on page G1-3832 |
| Undefined Instruction | *Pseudocode description of taking the Undefined Instruction exception* on page G1-3813 |
| Hyp Trap | *Pseudocode description of taking the Hyp Trap exception* on page G1-3814 |
| Monitor Trap | *Pseudocode description of taking the Monitor Trap exception* on page G1-3814 |
| Supervisor Call | *Pseudocode description of taking the Supervisor Call exception* on page G1-3816 |
| Secure Monitor Call | *Pseudocode description of taking the Secure Monitor Call exception* on page G1-3817 |
| Hypervisor Call | *Pseudocode description of taking the Hypervisor Call exception* on page G1-3818 |
| Prefetch Abort | *Pseudocode description of taking the Prefetch Abort exception* on page G1-3820 |
| Data Abort | *Pseudocode description of taking the Data Abort exception* on page G1-3823 |
| Virtual Abort | *Pseudocode description of taking the Virtual Asynchronous Abort exception* on page G1-3824 |
| IRQ | *Pseudocode description of taking the IRQ exception* on page G1-3826 |
| Virtual IRQ | *Pseudocode description of taking the Virtual IRQ exception* on page G1-3827 |
| FIQ | *Pseudocode description of taking the FIQ exception* on page G1-3828 |
| Virtual FIQ | *Pseudocode description of taking the Virtual FIQ exception* on page G1-3829 |

The following sections give more information about the PE state changes, for different architecture implementations. However, you must refer to the pseudocode for a full description of the state changes:

- *PE mode for taking exceptions* on page G1-3784.
- *PE state on exception entry* on page G1-3788.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Link values saved on exception entry

On exception entry, a link value for use on return from the exception, is saved. This link value is based on the *preferred return address for the exception*, as shown in Table G1-6:

**Table G1-6 Exception return addresses for exceptions taken to AArch32 state**

| Exception | Preferred return address | Taken to a mode at |
|---|---|---|
| Undefined Instruction | Address of the UNDEFINED instruction | Non-EL2[a], or EL2[c] |
| Hyp Trap | Address of the trapped instruction | EL2 only[c] |
| Monitor Trap | Address of the trapped instruction | EL3 only |
| Supervisor Call | Address of the instruction after the SVC instruction | Non-EL2[a] or EL2[c] |
| Secure Monitor Call | Address of the instruction after the SMC instruction | EL3[b], and only in Secure state |
| Hypervisor Call | Address of the instruction after the HVC instruction | EL2 only[c] |
| Prefetch Abort | Address of aborted instruction fetch | Non-EL2[a] or EL2[c] |
| Data Abort | Address of instruction that generated the abort | Non-EL2[a] or EL2[c] |
| Virtual Abort | Address of next instruction to execute | EL1, and only in Non-secure state |
| IRQ or FIQ | Address of next instruction to execute | Non-EL2[a] or EL2[c] |
| Virtual IRQ or Virtual FIQ | Address of next instruction to execute | EL1, and only in Non-secure state |

a. EL1 if the exception is taken to a Non-secure mode, or is taken to a Secure mode when EL3 is using AArch64. EL3 if the exception is taken to a Secure mode when EL3 is using AArch64.

b. A Secure Monitor Call exception is taken to EL3, and therefore is taken to AArch32 state only if EL3 is using AArch32, in which case it is taken to Monitor mode.

c. EL2 is implemented only in Non-secure state. Therefore, an exception can be taken to EL2 mode only if it is taken from Non-secure state.

**Note**

- Although Reset is described as an exception, it differs significantly from other exceptions. The architecture has no concept of a return from a Reset and therefore it is not listed in this section.
- For each exception, the preferred return address is not affected by the Exception level from which the exception was taken.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The link value saved, and where it is saved, depend on whether the exception is taken to a non-EL2 mode, or to an EL2 mode, as follows:

**Exception taken to a non-EL2 mode**

The link value is saved in the LR for the mode to which the exception is taken.

The saved link value is the preferred return address for the exception, plus an offset that depends on the instruction set state when the exception was taken, as Table G1-7 shows:

**Table G1-7 Offsets applied to Link value for exceptions taken to non-EL2 modes**

| Exception | Offset, for PE state of: | |
|---|---|---|
| | A32 | T32 |
| Undefined Instruction | +4 | +2 |
| Monitor Trap | +4 | +2 |
| Supervisor Call | None | None |
| Secure Monitor Call | None | None |
| Prefetch Abort | +4 | +4 |
| Data Abort | +8 | +8 |
| Virtual Abort | +8 | +8 |
| IRQ or FIQ | +4 | +4 |
| Virtual IRQ or Virtual FIQ | +4 | +4 |

**Exception taken to an EL2 mode**

The link value is saved in the ELR_hyp Special-purpose register.

The saved link value is the preferred return address for the exception, as shown in Table G1-6 on page G1-3783, with no offset.

## G1.11.4 PE mode for taking exceptions

The following principles determine the Exception level to which an exception is taken, and if that Exception level is using AArch32, the PE mode to which the exception is taken:

- An exception cannot be taken to the EL0 mode.
- An exception is taken either:
  - To the Exception level at which the PE was executing when it took the exception.
  - To a higher Exception level.

  This means that, in Secure state:
  - When EL3 is using AArch32, an exception is always taken to an EL3 mode.
  - When EL3 is using AArch64, an exception that is taken to AArch32 state is taken to an EL1 mode.
- Configuration options and other features provided by EL2 and EL3 can determine the mode to which some exceptions are taken, as follows:

  **In an implementation that does not include EL2 or EL3**

  An exception is always taken to the default mode for that exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**In an implementation that includes EL3**

A Secure Monitor Call exception is always taken to EL3. This means:

- If EL3 is using AArch32 the exception is taken to Secure Monitor mode.
- If EL3 is using AArch64 then executing the instruction generates an exception that is taken to EL3, see *Execution of an SMC instruction from a privileged Exception level that is using AArch32* on page G1-3786.

IRQ, FIQ, and External abort exceptions can be configured to be taken to EL3. Therefore, if EL3 is using AArch32 the exceptions are taken to Secure Monitor mode.

When EL3 is using AArch32, a Monitor Trap exception is taken to Secure Monitor mode.

Any exception taken from Secure state that is not taken to Secure Monitor mode is taken to Secure state in the default mode for that exception. As described in *Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990, this means it is taken to:

- An EL3 mode other than Monitor mode if EL3 is using AArch32.
- An EL1 mode if EL3 is using AArch64.

If the implementation does not include EL2, any exception taken from Non-secure state that is not taken to Secure Monitor mode is taken to Non-secure state to the default mode for that exception. The default mode will be an EL1 mode.

**In an implementation that includes EL2**

An exception taken from Non-secure state that is not taken to Secure Monitor mode is taken to Non-secure state and:

- If the exception is taken from Hyp mode then it is taken to Hyp mode.
- Otherwise, the exception is either taken to Hyp mode, as described in *Exceptions taken to Hyp mode* on page G1-3786, or taken to the default mode for the exception.

**Note**

- Hyp mode is the EL2 mode. The other modes to which an exception can be taken in Non-secure state are EL1 modes.
- EL2 has no effect on the handling of exceptions taken from Secure state.

Table G1-4 on page G1-3776 shows the default mode to which each exception is taken.

*Asynchronous exception routing controls* on page G1-3803 describes the exception routing controls provided by EL2 and EL3.

*Routing of aborts taken to AArch32 state* on page G4-4078 gives more information about the modes to which memory aborts are taken.

*The possible modes for taking each exception* on page G1-3787 shows all modes to which each exception might be taken, in any implementation. That is, it applies to implementations:

- That include neither EL2 nor EL3.
- That include EL2 but not EL3
- That do not include EL2 but include EL3
- That include both EL2 and EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Exceptions taken to Hyp mode

In an implementation that includes EL2 and EL3, when EL2 is using AArch32:

- Any exception taken from Hyp mode, that is not routed to EL3 by the controls described in *Asynchronous exception routing controls* on page G1-3803, is taken to Hyp mode.
- The following exceptions, if taken from Non-secure state, are taken to Hyp mode:
  - An abort that *Routing of aborts taken to AArch32 state* on page G4-4078 identifies as taken to Hyp mode.
  - A Hyp Trap exception, see *EL2 configurable controls* on page G1-3855.
  - A Hypervisor Call exception. This is generated by executing an `HVC` instruction in a Non-secure mode.
  - An asynchronous abort, IRQ exception or FIQ exception that is not routed to EL3 but is explicitly routed to Hyp mode, as described in *Asynchronous exception routing controls* on page G1-3803.
  - A synchronous external abort, Alignment fault, Undefined Instruction exception, or Supervisor Call exception taken from the Non-secure EL0 mode and explicitly routed to Hyp mode, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.

    **Note**

    A synchronous external abort can be routed to Hyp mode only if it is not routed to EL3.

  - A debug exception that is explicitly routed to Hyp mode as described in *Routing debug exceptions to EL2* on page G1-3795.

**Note**

The virtual exceptions cannot be taken to Hyp mode. They are always taken to a Non-secure EL1 mode.

### Security behavior in Exception levels using AArch32 when EL3 is using AArch64

As described in *The ARMv8-A security model* on page G1-3753, when EL3 is using AArch64, lower Exception levels, in either Security state, can be using AArch32. This means software executing in those Exception levels might try to access AArch32 security features that are not available. The following subsections describe the associated behaviors:

- *Execution of an SMC instruction from a privileged Exception level that is using AArch32*
- *Non-secure reads of the NSACR*
- *Secure EL1 operations when Secure EL1 is using AArch32* on page G1-3787

#### *Execution of an SMC instruction from a privileged Exception level that is using AArch32*

When EL3 is using AArch64, an `SMC` instruction executed from Secure or Non-secure EL1 using AArch32, or from Non-secure EL2 using AArch32 when the value of HCR.TSC is 0, generates an exception that is taken to EL3. The exception syndrome is reported with an EC value of `0x13`, `SMC` instruction executed in AArch32 state, see *Exception from SMC instruction execution in AArch32 state* on page D1-1547.

#### *Non-secure reads of the NSACR*

The NSACR is defined as being RO from Non-secure PE modes other than User mode. When EL3 is using AArch64, a read of the NSACR returns a fixed value of `0x00000C00` in the following cases:

- If the read is from a Non-secure EL1 mode when EL1 is using AArch32.
- If the read is from Hyp mode when EL2 is using AArch32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Secure EL1 operations when Secure EL1 is using AArch32*

When Secure EL1 is using AArch32 and EL3 is using AArch64:

- Any of the following operations performed in a Secure EL1 mode is trapped to Secure EL3:
  - A read or write of any of the SCR, NSACR, MVBAR, and SDCR.
  - Executing any of the ATS12NSO** instructions described in *ATS12NSOxx, Address translation stages 1 and 2, Non-secure state only* on page G4-4111.
  - Executing an `SRS` instruction that would use SP_mon, see *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F6-3013.
  - Executing an `MRS` (Banked register) or `MSR` (Banked register) instruction that would access SPSR_mon, SP_mon, or LR_mon, see *MRS (Banked register)* on page F6-2832 and *MSR (Banked register)* on page F6-2835.

  For more information about these traps, including the associated exception syndromes, see *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32* on page D1-1602.
- Writes to the CNTFRQ register are UNDEFINED.
- Any attempt to move into Monitor mode, either by an exception return or by executing a `CPS` or `MSR` instruction, is treated as an illegal operation and is handled as described in *Illegal return events from AArch32 state* on page G1-3797.

**Note**

This functionality supports a usage model where:

- EL3 uses AArch64.
- Secure software executed in Secure EL1 using AArch32 and Secure EL0 using AArch32.
- The Non-secure state uses AArch64.

### The possible modes for taking each exception

Each of the exception descriptions in *AArch32 state exception descriptions* on page G1-3811 includes a subsection that describes the modes to which each exception can be taken. Those subsections are:

- *The PE mode to which the Undefined Instruction exception is taken* on page G1-3812.
- *The PE mode to which the Hyp Trap exception is taken* on page G1-3814.
- *The PE mode to which the Monitor Trap exception is taken* on page G1-3814.
- *The PE mode to which the Supervisor Call exception is taken* on page G1-3815.
- *The PE mode to which the Secure Monitor Call exception is taken* on page G1-3817.
- *The PE mode to which the Hypervisor Call exception is taken* on page G1-3817.
- *The PE mode to which the Prefetch Abort exception is taken* on page G1-3820.
- *The PE mode to which the Data Abort exception is taken* on page G1-3822.
- *The PE mode to which the Virtual Abort exception is taken* on page G1-3824.
- *The PE mode to which the physical IRQ exception is taken* on page G1-3825.
- *The PE mode to which the Virtual IRQ exception is taken* on page G1-3827.
- *The PE mode to which the physical FIQ exception is taken* on page G1-3828.
- *The PE mode to which the Virtual FIQ exception is taken* on page G1-3829.

These descriptions also show the vector offset for the exception entry for each mode. These descriptions assume that all Exception levels are using AArch32, meaning:

- HCR, rather than HCR_EL2, controls the routing of exceptions to EL2.
- SCR, rather than SCR_EL3, controls the routing of exceptions to EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about:

- Vector offsets, see *Exception vectors and the exception base address* on page G1-3774.
- The routing of external aborts, IRQ and FIQ exceptions, and the virtual exceptions, see *Asynchronous exception routing controls* on page G1-3803.

#### *UNPREDICTABLE cases when the value of HCR.TGE is 1*

When the value of HCR.TGE is 1, exceptions that would otherwise be taken to EL1 are, instead, routed to EL2, see *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790. Related to this, when the value of HCR.TGE is 1, execution in a Non-secure EL1 mode is UNPREDICTABLE. ARMv8 does not constrain this UNPREDICTABLE behavior, but in ARMv8 software that follows the ARM recommendations cannot get to this state. When following the ARM recommendations, any attempt to move to a Non-secure EL1 mode when the value of HCR.TGE is 1 is either:

- An illegal exception return, see *Illegal return events from AArch32 state* on page G1-3797.
- An illegal PE mode change, see *Illegal changes to PSTATE.M* on page G1-3771.

### G1.11.5 PE state on exception entry

The description of each exception includes a pseudocode description of entry to that exception, as Table G1-5 on page G1-3782 shows. The following sections describe the PE state changes on entering an exception, for different implementations and operating states. However, you must always see the exception entry pseudocode for a full description of the state changes on exception entry:

- *Instruction set state on exception entry*.
- *PSTATE.E value on exception entry* on page G1-3789.
- *PSTATE.{A, I, F, M} values on exception entry* on page G1-3789.

**Note**

The descriptions in these sections assume that EL2 and EL3, that control some aspects of the routing of exceptions taken from EL1 or EL0, are both using AArch32. If this is not the case:

- If EL2 is using AArch64:
  - Controls shown as provided by the HSCTLR are provided by the SCTLR_EL2.
  - Controls shown as provided by the HCR are provided by the HCR_EL2.
- If EL3 is using AArch64, controls shown as provided by the SCR are provided by the SCR_EL3.

#### Instruction set state on exception entry

Exception handlers can execute in either T32 state or A32 state. On exception entry, PSTATE.T is set to the required value, as determined by SCTLR.TE or HSCTLR.TE, depending on the mode the exception is taken to. Table G1-8 shows this:

**Table G1-8 PSTATE.T bit value on exception entry**

| Exception mode | HSCTLR.TE | SCTLR.TE | PSTATE.T | Exception handler state |
|---|---|---|---|---|
| Not Hyp | x | 0 | 0 | A32 |
| | | 1 | 1 | T32 |
| Hyp | 0 | x | 0 | A32 |
| | 1 | x | 1 | T32 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When an implementation includes EL3 and EL3 is using AArch32, SCTLR is Banked for Secure and Non-secure states, and therefore the TE bit value might be different for Secure and Non-secure states. For an exception taken to a Secure or Non-secure non-Hyp mode, the SCTLR.TE bit for the Security state to which the exception is taken determines the instruction set state for the exception handler. This means the non-Hyp mode exception handlers might run in different instruction set states, depending on the Security state.

### PSTATE.E value on exception entry

PSTATE.E controls the load and store endianness for data handling. Table G1-9 show the value to which this bit is set on exception entry:

**Table G1-9 PSTATE.E value on exception entry**

| Exception mode | HSCTLR.EE | SCTLR.EE | Endianness for data loads and stores | PSTATE.E |
|---|---|---|---|---|
| Secure or Non-secure EL1 | x | 0 | Little-endian | 0 |
| | | 1 | Big-endian | 1 |
| Hyp | 0 | x | Little-endian | 0 |
| | 1 | x | Big-endian | 1 |

For more information, see the bit description in *SPSR format for exceptions taken to AArch32 state* on page G1-3766.

### PSTATE.{A, I, F, M} values on exception entry

On exception entry, PSTATE.M is set to the value for the mode to which the exception is taken, as described in *PE mode for taking exceptions* on page G1-3784.

Table G1-10 on page G1-3790 shows the cases where PSTATE.{A, I, F} bits are set to 1 on an exception entry, and how this depends on the mode and Security state to which an exception is taken. If the table entry for a particular mode and Security state does not define a value for a PSTATE.{A, I, F} bit then that bit is unchanged by the exception entry. In this table:

- The *Exception mode* column is the mode to which the exception is taken.
- The *Non-secure, EL2 not implemented* column applies to exceptions taken to Non-secure state in an implementation that includes EL3 but does not include EL2.
- The *All others* column applies to:
  - Exceptions taken to Secure state.
  - Implementations that do not include the EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Exceptions taken to Non-secure state in an implementation that includes EL2.

**Table G1-10 PSTATE.{A, I, F} values on exception entry**

| PE mode exception is taken to | Security state | |
|---|---|---|
| | Non-secure | Secure |
| Hyp | If SCR.EA==0 then PSTATE.A is set to 1<br>If SCR.IRQ==0 then PSTATE.I is set to 1<br>If SCR.FIQ==0 then PSTATE.F is set to 1 | - |
| Monitor | - | PSTATE.A is set to 1<br>PSTATE.I is set to 1<br>PSTATE.F is set to 1 |
| FIQ | PSTATE.A is set to 1<br>PSTATE.I is set to 1<br>PSTATE.F is set to 1 | PSTATE.A is set to 1<br>PSTATE.I is set to 1<br>PSTATE.F is set to 1 |
| IRQ, Abort | PSTATE.A is set to 1<br>PSTATE.I is set to 1 | PSTATE.A is set to 1<br>PSTATE.I is set to 1 |
| Undefined, Supervisor | PSTATE.I is set to 1 | PSTATE.I is set to 1 |

*Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-3801 describes how, in some situations, the PSTATE.{A, I, F} bits mask the taking of asynchronous aborts, IRQ interrupts, and FIQ interrupts.

### G1.11.6 Routing exceptions from Non-secure EL0 to EL2

**Note**

The routing control described in this section permits a Non-secure state usage model where applications execute in User mode under a hypervisor, that executes in Hyp mode, without a Guest OS running at Non-secure EL1. This control applies when either:

- EL2 is using AArch32 and the value of HCR.TGE is 1.
- EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

If the PE is in Non-secure User mode, any exception that would otherwise be taken to Non-secure EL1 is taken to EL2 if either:

- EL2 is using AArch32 and the value of HCR.TGE is 1.

  In this case the exception is taken to Hyp mode, instead of to the default Non-secure mode for handling the exception. For more information see *Exception reporting when HCR.TGE routes an exception to EL2 using AArch32* on page G1-3791.

- EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

Any exception that is routed to Secure Monitor mode or to EL3 using AArch64 is unaffected by the value of HCR.TGE or HCR_EL2.TGE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When TGE routing from Non-secure EL0 applies:

- When EL2 is using AArch32:
  - The SCTLR.M bit is treated as 0 for all purposes other than a direct read of the SCTLR register.
  - Each of the HCR.{FMO, IMO, AMO} bits is treated as 1 for all purposes other than a direct read of the HCR register
  - Each of the HDCR.{TDE, TDA, TDRA, TDOSA} bits is treated as 1 for all purposes other than a direct read of the HDCR register.

  **Note**

  When EL2 is using AArch64:
  - The SCTLR_EL2.M bit is treated as 0 for all purposes other than a direct read of the register.
  - The HCR_EL2.{FMO, IMO, AMO} bits are treated as 1 for all purposes other than a direct read of the register.
  - The MDCR_EL2.{TDE, TDA, TDRA, TDOSA} bits are treated as 1 for all purposes other than a direct read of the register.

- An exception return to EL1 is treated as an illegal exception return, see *Illegal return events from AArch32 state* on page G1-3797.
- All virtual interrupts, including any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts, are disabled.

## Exception reporting when HCR.TGE routes an exception to EL2 using AArch32

The following sections give more information about the behavior of synchronous exceptions that are routed to Hyp mode because the value of HCR.TGE is 1:

- *Undefined Instruction exception, when the value of HCR.TGE is 1*.
- *Supervisor Call exception, when the value of HCR.TGE is 1*.
- *Abort exceptions, when the value of HCR.TGE is 1* on page G1-3792.
- *Reporting of exceptions routed to EL2 using AArch32 because the value of HCR.TGE is 1* on page G1-3792.

### *Undefined Instruction exception, when the value of HCR.TGE is 1*

When HCR.TGE is set to 1, if the PE is executing in Non-secure User mode and attempts to execute an UNDEFINED instruction, it takes the Hyp Trap exception, instead of an Undefined Instruction exception. On taking the Hyp Trap exception, the HSR reports an unknown reason for the exception, using the EC value `0x00`. For more information see *Use of the HSR* on page G4-4105.

### *Supervisor Call exception, when the value of HCR.TGE is 1*

When HCR.TGE is set to 1, if the PE executes an `SVC` instruction in Non-secure User mode, the Supervisor Call exception generated by the instruction is taken to Hyp mode.

The HSR reports that entry to Hyp mode was because of a Supervisor Call exception, and:

- If the `SVC` is unconditional, takes for the `imm16` value in the HSR:
  - A zero-extended 8-bit immediate value for the T32 `SVC` instruction.

    **Note**

    The only T32 encoding for `SVC` is a 16-bit instruction encoding.

  - The bottom16 bits of the immediate value for the A32 `SVC` instruction.
- If the `SVC` is conditional, the `imm16` value in the HSR is UNKNOWN.

If the `SVC` is conditional, the PE takes the exception only if the instruction passes its condition code check.

The HSR reports the exception as a Supervisor Call exception taken to Hyp mode, using the EC value `0x11`. For more information, see *Use of the HSR* on page G4-4105.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The effect of setting HCR.TGE to 1 is to route the Supervisor Call exception to Hyp mode, not to trap the execution of the SVC instruction. This means that the preferred return address for the exception, when routed to Hyp mode in this way, is the instruction after the SVC instruction.

#### *Abort exceptions, when the value of HCR.TGE is 1*

When the value of HCR.TGE is 1, if the PE is executing in Non-secure User mode then any abort exception that is not routed to Secure Monitor mode or to EL3 using AArch64 generates an exception that is taken as a Hyp Trap exception. Where an attempt to execute an instruction causes an abort, on taking the Hyp Trap exception, the HSR indicates whether a Data Abort exception or a Prefetch Abort exception caused the Hyp Trap exception entry, and presents a valid syndrome in the HSR.

**Note**

- When SCR.EA is set to 1, external aborts are routed to Secure Monitor mode, and this takes priority over the HCR.TGE routing. For more information, see *Asynchronous exception routing controls* on page G1-3803. The SCR.EA control described in that section applies to both synchronous and asynchronous external aborts.
- Any asynchronous external abort generates a Data Abort exception. Therefore, if an asynchronous external abort is routed to Hyp mode because the value of HCR.TGE is 1 the exception is reported as a Data Abort exception routed to Hyp mode.

The HSR reports the exception either:

- As a Prefetch Abort exception routed to Hyp mode, using the EC value 0x20.
- As a Data Abort exception routed to Hyp mode, using the EC value 0x24.

For more information about the exception reporting, see *Use of the HSR* on page G4-4105.

#### *Reporting of exceptions routed to EL2 using AArch32 because the value of HCR.TGE is 1*

*PL1 configurable controls* on page G1-3847 describes controls that, when the value of HCR.TGE is 0, can generate exceptions that are taken from Non-secure EL0 to EL1. When EL2 is using AArch32 and the value of HCR.TGE is 1, the exceptions generated by these controls are routed to Hyp mode. Table G1-11 shows how the exception is then reported in the HSR.

**Table G1-11 Syndrome reporting in HSR from HCR.TGE routing of traps, disables, and enables**

| Control provided by PL1 | Control type[a] | Syndrome reporting in HSR |
|---|---|---|
| SCTLR.{nTWE, nTWI} | T | Trapped WFI or WFE instruction, using EC value 0x01 |
| SCTLR.{SED, ITD} | D | Exception for an unknown reason, using EC value 0x00 |
| SCTLR.CP15BEN | E | Exception for an unknown reason, using EC value 0x00 |
| CPACR.TRCDIS | T | For accesses using:<br>• MCR or MRC instructions, trapped MCR or MRC CP14 access, using EC value 0x05.<br>• MCRR or MRRC instructions, trapped MCRR or MRRC CP14 access, using EC value 0x0C. |
| CPACR.{cp11, cp10} | E | Exception for an unknown reason, using EC value 0x00 |
| FPEXC.EN | E | Exception for an unknown reason, using EC value 0x00 |
| CPACR.ASEDIS | D | Exception for an unknown reason, using EC value 0x00 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G1-11 Syndrome reporting in HSR from HCR.TGE routing of traps, disables, and enables (continued)**

| Control provided by PL1 | Control type[a] | Syndrome reporting in HSR |
|---|---|---|
| DBGDSCRext.UDCCdis | T | For accesses using:<br>• MCR or MRC instructions, trapped MCR or MRC CP14 access, using EC value 0x05.<br>• LDC or STC instructions, trapped LDC or STC access to CP14, using EC value 0x06.<br>• MRRC instructions, trapped MCRR or MRRC CP14 access, using EC value 0x0C: |
| CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN} | T | For accesses using:<br>• MRC or MCR instructions, trapped MCR or MRC CP15 access, using EC value 0x03.<br>• MCRR or MRRC instructions, trapped MCRR or MRRC CP15 access, using EC value 0x04. |
| PMUSERENR.{ER, CR, SW, EN} | T | For accesses using:<br>• MRC or MCR instructions, trapped MCR or MRC CP15 access, using EC value 0x03.<br>• MCRR or MRRC instructions, trapped MCRR or MRRC CP15 access, using EC value 0x04. |

a. T indicates a trap control, E indicates an instruction enable, and D indicates an instruction disable. For the definition of these terms, see the list that begins with *Instruction enables and instruction disables* on page G1-3846.

## Exception reporting when HCR_EL2.TGE routes an exception to EL2 using AArch64

This section give more information about the behavior of synchronous exceptions that are routed from Non-secure EL0 using AArch32 to EL2 using AArch64 because the value of HCR_EL2.TGE is 1. That is, it describes the behavior of exceptions that are routed from Non-secure User mode to EL2 using AArch64. See:

- *Synchronous aborts from Non-secure User mode, when the value of HCR_EL2.TGE is 1*.
- *Reporting of exceptions routed to EL2 using AArch64 because the value of HCR_EL2.TGE is 1* on page G1-3794.

### *Synchronous aborts from Non-secure User mode, when the value of HCR_EL2.TGE is 1*

When executing in Non-secure User mode, the following sections describe synchronous exceptions that can occur:

- *Undefined Instruction exception* on page G1-3811.
- *Supervisor Call (SVC) exception* on page G1-3815.
- *Prefetch Abort exception* on page G1-3818.
- *Data Abort exception* on page G1-3821.

When EL2 is using AArch64 and the value of HCR_EL2.TGE is 1, these exceptions are routed to EL2, and reported in the ESR_EL2. Table G1-12 shows how they are reported.

**Table G1-12 Syndrome reporting in ESR_EL2 of HCR_EL2 routing of exceptions**

| AArch32 exception | Syndrome reporting in ESR_EL2 |
|---|---|
| Undefined Instruction | Exception for an unknown reason, using EC value 0x00 |
| Supervisor Call | Exception from SVC executed in AArch32 state, using EC value 0x11 |
| Prefetch Abort | Exception from an Instruction abort at a lower Exception level, using EC value 0x20 |
| Data Abort | Exception from a Data abort at a lower Exception level, using EC value 0x24 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Reporting of exceptions routed to EL2 using AArch64 because the value of HCR_EL2.TGE is 1*

Table G1-13 shows syndrome reporting for exceptions that are routed from Non-secure User mode to EL2 using AArch64 by HCR_EL2.TGE.

For:

- Instruction enable and disable controls provided by PL1, instructions are UNDEFINED when disabled. The Undefined Instruction exceptions that are routed to EL2 using AArch64 are reported with ESR_EL2.EC value `0x00`, indicating an unknown reason.
- Trap controls provided by PL1, trapped instructions generate Trap exceptions. These are reported in the ESR_EL2 with an appropriate EC value.

**Table G1-13 Syndrome reporting in ESR_EL2 of HCR_EL2 routing of traps. enables and disables**

| Control provided by PL1 | Control type[a] | Syndrome reporting in ESR_EL2 |
|---|---|---|
| SCTLR.{nTWE, nTWI} | T | Trapped `WFI` or `WFE` instruction, using EC value `0x01` |
| SCTLR.{SED, ITD} | D | Exception for an unknown reason, using EC value `0x00` |
| SCTLR.CP15BEN | E | Exception for an unknown reason, using EC value `0x00` |
| CPACR.TRCDIS | T | For accesses using:<br>• `MCR` or `MRC` instructions, trapped `MCR` or `MRC` CP14 access, using EC value `0x05`.<br>• `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` CP14 access, using EC value `0x0C`. |
| CPACR.{cp11, cp10} | E | Exception for an unknown reason, using EC value `0x00` |
| FPEXC.EN | E | Exception for an unknown reason, using EC value `0x00` |
| CPACR.ASEDIS | D | Exception for an unknown reason, using EC value `0x00` |
| DBGDSCRext.UDCCdis | T | For accesses using:<br>• `MCR` or `MRC` instructions, trapped `MCR` or `MRC` CP14 access, using EC value `0x05`.<br>• `LDC` or `STC` instructions, trapped `LDC` or `STC` access to CP14, using EC value `0x06`.<br>• `MRRC` instructions, trapped `MCRR` or `MRRC` CP14 access, using EC value `0x0C`: |
| CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN} | T | For accesses using:<br>• `MRC` or `MCR` instructions, trapped `MCR` or `MRC` CP15 access, using EC value `0x03`.<br>• `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` CP15 access, using EC value `0x04`. |
| PMUSERENR.{ER, CR, SW, EN} | T | For accesses using:<br>• `MRC` or `MCR` instructions, trapped `MCR` or `MRC` CP15 access, using EC value `0x03`.<br>• `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` CP15 access, using EC value `0x04`. |

a. T indicates a trap control, E indicates an instruction enable, and D indicates an instruction disable. For the definition of these terms, see the list that begins with *Instruction enables and instruction disables* on page G1-3846.

For all exceptions that Table G1-13 applies to except those generated by CPACR.{cp11, cp10}, ESR_EL2 reporting is the same as it would have been in ESR_EL1 under the following conditions:

- EL1 using AArch64.
- HCR_EL2.TGE set to 0.

For exceptions generated by CPACR.{cp11, cp10}, ESR_EL2 reporting is as follows:

- The ESR_EL2.EC value is reported as `0x00`. This differs from the EC value reported in ESR_EL1 when an exception is generated by CPACR_EL1.FPEN and is taken to EL1 using AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G1.11.7 Routing debug exceptions to EL2

When the value of HDCR.TDE is 1, if the PE is executing in a Non-secure mode other than Hyp mode, any Debug exception is routed to Hyp mode. This means it generates a Hyp Trap exception. This applies to:

- Debug exceptions associated with an instruction fetch, that would otherwise generate a Prefetch Abort exception. These are the Breakpoint, Software Breakpoint Instruction, and Vector Catch exception, see Chapter G2 *AArch32 Self-hosted Debug*.
- Watchpoint exceptions associated with data accesses, that would otherwise generate a Data Abort exception. See *Watchpoint exceptions* on page G2-3923.

When the value of HDCR.TDE is 1, each of the HDCR.{TDRA, TDOSA, TDA} bits is treated as 1 for all purposes other than reading the HDCR register.

**Note**

- A Breakpoint or Watchpoint debug event that generates entry to Debug state cannot be trapped to Hyp mode. See *Breakpoint and Watchpoint debug events* on page H2-4822.
- When HDCR.TDE is set to 1, the Hyp Trap exception is generated instead of the Prefetch Abort exception or Data Abort exception that is otherwise generated by the Debug exception.
- Debug exceptions, other than Software Breakpoint Instruction exceptions, are never generated in Hyp mode.

When a Hyp Trap exception is generated because HDCR.TDE is set to 1, The HSR reports the exception either:

- As a Prefetch Abort exception routed to Hyp mode, using the EC value `0x20`.
- As a Data Abort exception routed to Hyp mode, using the EC value `0x24`.

For more information see *Use of the HSR* on page G4-4105.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.12 Exception return to an Exception level using AArch32

In the ARM architecture, *exception return* to an Exception level that is using AArch32 requires the simultaneous restoration of the PC and PSTATE to values that are consistent with the desired state of execution on returning from the exception. Typically, exception return involves returning to one of:

- The instruction after the instruction boundary at which an asynchronous exception was taken.
- The instruction following an `SVC`, `SMC`, or `HMC` instruction, for an exception generated by one of those instructions.
- The instruction that caused the exception, after the reason for the exception has been removed.
- The subsequent instruction, if the instruction that caused the exception has been emulated in the exception handler.

The ARM architecture defines a *preferred return address* for each exception other than Reset, see *Link values saved on exception entry* on page G1-3783. The values of the SPSR.IT[7:0] bits generated on exception entry are always correct for this preferred return address, but might require adjustment by the exception handler if returning elsewhere.

In some cases, to calculate the appropriate preferred return address for a return to an Exception level that is using AArch32, a subtraction must be performed on the link value saved on taking the exception. The description of each exception includes any value that must be subtracted from the link value, and other information about the required exception return.

On an exception return, the PSTATE takes either:

- The value loaded by the `RFE` instruction.
- If the exception return is not performed by executing an `RFE` instruction, the value of the current SPSR at the time of the exception return.

*Illegal return events from AArch32 state* on page G1-3797 describes the behavior if the restored PE state would not be valid for the Exception level, PE mode, and Security state targeted by the exception return.

### G1.12.1 Exception return instructions

The instructions that an exception handler can use to return from an exception depend on whether the exception was taken to an EL1 mode, or in an EL2 mode, see:

- *Return from an exception taken to a PE mode other than Hyp mode*.
- *Return from an exception taken to Hyp mode* on page G1-3797.

#### Return from an exception taken to a PE mode other than Hyp mode

For an exception taken to a PE mode other than Hyp mode, the ARM AArch32 architecture provides the following *exception return instructions*:

- Data-processing instructions with the S bit set and the PC as a destination, see *MOV, MOVS (register)* on page F6-2813 and *SUB, SUBS (immediate)* on page F6-3088.

  **Note**

  The A32 instruction set includes other instructions that can be used for an exception return, but ARM deprecates any use of those instructions.

  Typically:

  — A return where no subtraction is required uses `SUBS` with an operand of 0, or the equivalent `MOVS` instruction.

  — A return requiring subtraction uses `SUBS` with a nonzero operand.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The RFE instruction, see *RFE, RFEDA, RFEDB, RFEIA, RFEIB* on page F6-2915. If a subtraction is required, typically it is performed before saving the LR value to memory.
- In A32 state, a form of the LDM instruction, see *LDM (exception return)* on page F6-2714. If a subtraction is required, typically it is performed before saving the LR value to memory.

#### Return from an exception taken to Hyp mode

For an exception taken to Hyp mode, the ARM architecture provides the ERET instruction, see *ERET* on page F6-2683. An exception handler executing in Hyp mode must return using the ERET instruction.

Both Hyp mode and the ERET instruction are implemented only as part of EL2.

### G1.12.2 Alignment of exception returns

The T bit of the value transferred to the PSTATE by an exception return controls the target instruction set of that return. The behavior of the hardware for exception returns for different values of the T bit is as follows:

**T == 0** The target instruction set state is A32 state. Bits[1:0] of the address transferred to the PC are ignored by the hardware.

**T == 1** The target instruction set state is T32 state:
- Bit[0] of the address transferred to the PC is ignored by the hardware.
- Bit[1] of the address transferred to the PC is part of the instruction address.

**Note**

In previous versions of the ARM architecture, the PSTATE.{J, T} bits determined the Instruction set state. In ARMv8, PSTATE.J is RES0.

ARM deprecates any dependence on the requirements that the hardware ignores bits of the address. ARM recommends that the address transferred to the PC for an exception return is correctly aligned for the target instruction set.

After an exception entry other than Reset, the LR value has the correct alignment for the instruction set indicated by the SPSR.T bit. This means that if exception return instructions are used with the LR and SPSR values produced by such an exception entry, the only precaution software needs to take to ensure correct alignment is that any subtraction is of a multiple of four if returning to A32 state, or a multiple of two if returning to T32 state.

### G1.12.3 Illegal return events from AArch32 state

Throughout this section:

**Return** In AArch32 state, refers to any of:
- Execution of any exception return instruction.
- Execution of a DRPS instruction in Debug state.
- Exit from Debug state.

If an exception or debug return from an Exception level using AArch32 triggers an illegal exception return, then bit[1] of the PC is either:
- Zero.
- The value of bit[1] of the return address for the exception or debug return.

The choice between these two alternatives is made by the implementation, and might differ from instance to instance of an illegal exception return.

**Note**

This means software must support both alternatives.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Saved process state value**
In AArch32 state, refers to any of:

- The value held in the SPSR for any exception return other than an exception return made by executing an RFE instruction.
- The value read from memory that is to be restored to PSTATE by the execution of an RFE instruction.
- The value held in the SPSR for the execution of a DRPS instruction in Debug state.
- The value held in the DSPSR for a Debug state exit.

**Link address** In AArch32 state, refers to any of:

- The address held in the link register for any exception return other than an exception return made by executing an ERET, LDM, or RFE instruction.
- The address held in ELR_hyp for any exception return made by executing an ERET instruction.
- The address read from memory that is to be restored to the PC by the execution of an LDM or RFE instruction.
- The address held in the DLR for Debug state exit.

**Configured from reset**
Indicates the state determined on powerup or reset by a configuration input signal, or by another IMPLEMENTATION DEFINED mechanism.

The ARMv8 architecture has a generic mechanism for handling exception or debug returns to a mode or state that is illegal. In AArch32 state, this can occur as a result of any of the following situations:

- A return where the Exception level being returned to is higher than the current Exception level.
- A return where the mode being returned to is not implemented. For example:
  - A return to Hyp mode when EL2 is not implemented.
  - A return to Monitor mode, when EL3 is either not implemented or using AArch64 state.
- A return to EL2 when:
  - EL3 is implemented and using AArch64, and the value of the SCR_EL3.NS bit is 0.
  - EL3 is implemented and using AArch32, and the value of the SCR.NS bit is 0.
- A return to Non-secure EL1 when:
  - EL2 is implemented and using AArch64, and the value of the HCR_EL2.TGE bit is 1.
  - EL2 is implemented and using AArch32, and the value of the HCR.TGE bit is 1.
- A return where the value of the saved process state M[4:0] field is not a valid AArch32 PE mode for the implementation. Table G1-2 on page G1-3758 shows the valid M[4:0] values for AArch32 PE modes.

In these cases:

- PSTATE.IL is set to 1, to indicate an illegal return.
- PSTATE.M is unchanged. This means the PE mode does not change.
- The SS bit is handled in the same way as any other exception or debug return, see *Software Step exceptions* on page D2-1686.
- The following PSTATE bits are restored from the saved process state value:
  - The N, Z, C, V Condition flags.
  - The Q Overflow or saturation flag.
  - The GE Greater than or Equal flags.
  - The E Endianness mapping bit.
  - The A, I, F exception mask bits.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The PSTATE.{IT, T} bits are each either:
  - Set to 0.
  - Copied from the saved process state in the SPSR for the PE mode in which the exception is handled.

  The choice between these two options is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.
- The PC is restored from the link address, unless the illegal return is the execution of a DRPS instruction in Debug state.

When the value of the PSTATE.IL bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception*.

All aspects of the illegal return, other than the effects described in this section, are the same as for a legal return.

### G1.12.4 Legal returns that set PSTATE.IL to 1

In this section, return, saved process state value, and link address have the meaning that is defined in *Illegal return events from AArch32 state* on page G1-3797.

If the IL bit in the saved process state value is 1, then it is copied to PSTATE meaning that PSTATE.IL is set to 1. In this case, the PSTATE.{IT, T} bits are each either:

- Set to 0.
- Copied from the SPSR, or loaded from memory if the exception return was performed by executing an RFE instruction.

The choice between these two options is determined by an implementation, and might vary dynamically within the implementation. This means software must regard each value as being an UNKNOWN choice between the two permitted values.

Because the return sets the PSTATE.IL bit to 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception*.

### G1.12.5 The Illegal Execution state exception

When the value of the PSTATE.IL bit is 1, any attempt to execute an instruction generates an Illegal Execution state exception. In AArch32 state, the PSTATE.IL bit can be set to 1 by one of the following:

- An illegal return, as described in *Illegal return events from AArch32 state* on page G1-3797.
- An illegal change to PSTATE.M, as described in *Illegal changes to PSTATE.M* on page G1-3771.
- A legal return that sets PSTATE.IL to 1, as described in *Legal returns that set PSTATE.IL to 1*.

An Illegal Execution state exception is taken in the same way as an Undefined Instruction exception in the current Exception level. If the current Exception level is EL2 using AArch32 state, the HSR provides additional syndrome information for the exception, see *Use of the HSR* on page G4-4105.

An Illegal Execution state exception has priority over any other Undefined Instruction exception that might arise from instruction execution.

**Note**

This section only describes the handling of an Illegal Execution state exception that is taken to an Exception level that is using AArch32 state. *The Illegal Execution state exception* on page D1-1555 describes the cases where an Illegal Execution state exception is taken to an Exception level that is using AArch64 state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On taking any exception to an Exception level that is using AArch32 state:

1. The value of the PSTATE.IL bit is 1 and this is copied to the SPSR.IL bit for the PE mode to which the exception is taken.
2. The PSTATE.IL bit is cleared to 0.

**Note**

This means that it is not possible for software to observe the value of PSTATE.IL.

### Pseudocode description of exception return

The `AArch32.ExceptionReturn()` function transfers the return address to the PC and restores PSTATE to its saved value.

This function uses the function `SetPSTATEFromPSR()`.

The `IllegalExceptionReturn()` function checks for an Illegal Execution state exception.

Chapter J1 *ARMv8 Pseudocode* includes the definitions of these functions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.13 Asynchronous exception behavior for exceptions taken from AArch32 state

In an implementation that does not include EL2 or EL3, the asynchronous exceptions behave as follows when EL1 and EL0 are both using AArch32:

- An asynchronous abort is taken to Abort mode.
- An IRQ exception is taken to IRQ mode.
- An FIQ exception is taken to FIQ mode.

These are the *default PE modes* for taking these exceptions.

However, the PSTATE.{A, I, F} bits *mask* the asynchronous exceptions, meaning that when the value of one of these PSTATE bits is 1, the corresponding exception is not taken.

If a masked asynchronous exceptions remains signaled, then the exception remains pending unless the value of the PSTATE bit is changed to 0.

EL2 and EL3 provide controls that affect:

- The routing of these exceptions, see *Asynchronous exception routing controls* on page G1-3803.
- Masking of these exceptions in Non-secure state, see *Asynchronous exception masking controls* on page G1-3804.

Similar register control bits are provided regardless of whether EL2 and EL3 are using AArch32 or AArch64:

- The EL2 controls are provided by the HCR when EL2 is using AArch32, and by the HCR_EL2 when EL2 is using AArch64.
- The EL3 controls are provided by the SCR when EL3 is using AArch32, and by the SCR_EL3 when EL3 is using AArch64.

Therefore, most references to the HCR or SCR in this section are to entries in Table K13-1 on page K13-5708, that disambiguates between AArch32 registers and AArch64 registers. However, the Execution states used by EL2 and EL3 do affect some aspects of the routing and masking of the asynchronous exceptions, see *Asynchronous exception routing and masking with higher Exception levels using AArch64* on page G1-3806.

### G1.13.1 Virtual exceptions when an implementation includes EL2

When implemented, EL2 provides the following virtual exceptions, that correspond to the physical asynchronous exceptions:

- Virtual Abort, that corresponds to a physical external asynchronous abort.
- Virtual IRQ, that corresponds to a physical IRQ.
- Virtual FIQ, that corresponds to a physical FIQ.

When the value of an HCR.{AMO, IMO, FMO} routing control bit is 1 and the value of HCR.TGE is 0, the corresponding virtual interrupt is enabled and a virtual exception is generated either:

- By setting the corresponding virtual interrupt pending bit, HCR.{VA, VI, VF}, to 1.
- For a Virtual IRQ or Virtual FIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller. See, for example, the *ARM Generic Interrupt Controller Architecture Specification*.

When the value of HCR_EL2.TGE is 1 all virtual interrupts are disabled.

When a virtual interrupt is disabled:

- It cannot be taken.
- It cannot be seen in the ISR.

In AArch32 state, a virtual exception is taken only from a Non-secure EL1 or EL0 mode. In any other mode, if the exception is generated it is not taken.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A virtual exception is taken in Non-secure state to the default mode for the corresponding physical exception. This means:

- A Virtual Abort is taken to Non-secure Abort mode.
- A Virtual IRQ is taken to Non-secure IRQ mode.
- A Virtual FIQ is taken to Non-secure FIQ mode.

Table G1-14 summarizes the HCR bits that route asynchronous exceptions to EL2, and the bits that generate the virtual exceptions.

**Table G1-14 HCR bits controlling asynchronous exceptions**

| Exception | Routing the physical exception to EL2 | Generating the virtual exception |
|---|---|---|
| Asynchronous abort | HCR.AMO | HCR.VA |
| IRQ | HCR.IMO | HCR.VI |
| FIQ | HCR.FMO | HCR.VF |

The HCR.{VA, VI, VF} bits generate a virtual exception only if set to 1 when the value of the corresponding HCR.{AMO, IMO, FMO} is 1.

Similarly, if the implementation also includes EL3, the HCR.{AMO, IMO, FMO} bits route the corresponding physical exception to Hyp mode only if the physical exception is not routed to Monitor mode by the SCR.{EA, IRQ, FIQ} bit. For more information, see *Asynchronous exception routing controls* on page G1-3803.

When the value of an HCR.{AMO, IMO, FMO} control bit is 1, the corresponding mask bit in PSTATE:

- Does not mask the physical exception.
- Masks the virtual exception when the PE is executing in a Non-secure EL1 or EL0 mode.

Taking a Virtual Abort exception clears HCR.VA to zero. Taking a Virtual IRQ exception or a Virtual FIQ exception does not affect the value of HCR.VI or HCR.VF.

**Note**

This means that the exception handler for a Virtual IRQ exception or a Virtual FIQ exception must cause software that is executing at EL2 or EL3 to update the HCR to clear the appropriate virtual exception bit to 0.

See *WFE wake-up events* on page G1-3836 and *Wait For Interrupt* on page G1-3837 for information about how virtual exceptions affect wake up from power-saving states.

**Note**

A hypervisor can use virtual exceptions to signal exceptions to the current Guest OS. The Guest OS takes a virtual exception exactly as it would take the corresponding physical exception, and is unaware of any distinction between virtual exception and the corresponding physical exception.

### Effects of the HCR.{AMO, IMO, FMO} bits

As described in this section, the HCR.{AMO, IMO, FMO} bits are part of the mechanism for enabling the virtual exceptions. In addition, for exceptions generated in Non-secure state:

- As mentioned in this section, affect the routing of the exceptions. See *Asynchronous exception routing controls* on page G1-3803.
- Affect the masking of the exceptions. See *Asynchronous exception masking controls* on page G1-3804.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### G1.13.2 Asynchronous exception routing controls

**Note**

This section describes the behavior when all exception levels are using AArch32. For the differences when this is not the case see *Asynchronous exception routing and masking with higher Exception levels using AArch64* on page G1-3806.

In an implementation that includes EL3 the following bits in the SCR control the routing of asynchronous exceptions, and also the routing of synchronous external aborts:

**SCR.EA** When the value of this bit is 1, any external abort is taken to EL3.

**Note**

- Although this section describes the asynchronous exception routing controls, SCR.EA controls the routing of both synchronous and asynchronous external aborts.
- The other classes of abort cannot be routed to EL3. For more information about the classification of aborts, see *VMSAv8-32 memory aborts* on page G4-4078.

**SCR.FIQ** When the value of this bit is 1, any FIQ exception is taken to EL3.

**SCR.IRQ** When the value of this bit is 1, any IRQ exception is taken to EL3.

When EL3 is using AArch32 and the value of one of the SCR.{EA, FIQ, IRQ} bits is 1, the exception is taken to Monitor mode.

Only Secure software can change the values of these bits.

In an implementation that includes EL2, the following bits in the HCR route asynchronous exceptions to EL2, for exceptions that are both:

- Taken from a Non-secure EL1 or EL0 mode.
- If the implementation also includes EL3, not configured, by the SCR.{EA, FIQ, IRQ} controls, to be taken to EL3.

**HCR.AMO** When the value of this bit is 1, an asynchronous external abort taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure Abort mode. If the implementation also includes EL3, this control applies only if the value of SCR.EA is 0. When the value of SCR.EA is 1, the value of the AMO bit is ignored.

**Note**

Figure G1-8 on page G1-3822 also shows how synchronous external aborts are handled.

**HCR.FMO** When the value of this bit is 1, an FIQ exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure FIQ mode. If the implementation also includes EL3, this control applies only if the value of SCR.FIQ is 0. When the value of SCR.FIQ is 1, the value of the FMO bit is ignored.

**HCR.IMO** When the value of this bit is 1, an IRQ exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure IRQ mode. If the implementation also includes EL3, this control applies only if the value of SCR.IRQ is 0. When the value of SCR.IRQ is 1, the value of the IMO bit is ignored.

When EL2 is using AArch32 and the value of one of the HCR.{AMO, FMO, IMO} bits is 1, the exception is taken to Hyp mode.

Only software executing in Hyp mode, or Secure software executing at EL3 with SCR.NS set to 1, can change the values of these bits. If EL3 is using AArch32, this requires the Secure software to be executing in Monitor mode.

The HCR.{AMO, FMO, IMO} bits also affect the masking of asynchronous exceptions in Non-secure state, as described in *Asynchronous exception masking controls* on page G1-3804.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The SCR.{EA, FIQ, IRQ} and HCR.{AMO, FMO, IMO} bits have no effect on the routing of Virtual Abort, Virtual FIQ, and Virtual IRQ exceptions.

**Note**

When the PE is in Hyp mode:

- Physical asynchronous exceptions that are not routed to Monitor mode are taken to Hyp mode.
- Virtual exceptions are not signaled to the PE.

See also *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-3801.

### G1.13.3 Asynchronous exception masking controls

**Note**

This section describes the behavior when all exception levels are using AArch32. For the differences when this is not the case see *Asynchronous exception routing and masking with higher Exception levels using AArch64* on page G1-3806.

The PSTATE.{A, I, F} bits can mask the taking of the corresponding exceptions from AArch32 state, as follows:

- PSTATE.A can mask asynchronous aborts.
- PSTATE.I can mask IRQ exceptions.
- PSTATE.F can mask FIQ exceptions.

In an implementation that does not include either of EL2 and EL3, setting one of these bits to 1 masks the corresponding exception, meaning the exception cannot be taken.

In an implementation that includes EL2, the HCR.{AMO, IMO, FMO} bits modify the masking of exceptions taken from Non-secure state.

Similarly, in an implementation that includes EL3, the SCR.{AW, FW} bits modify the masking of exceptions taken from Non-secure state by the PSTATE.{A, F} bits.

An implementation that includes only EL1 and EL0 does not provide any masking of the PSTATE.{A, I, F} bits. The following subsections describe the masking of these bits in other implementations:

- *Asynchronous exception masking in an implementation that includes EL2 but not EL3*.
- *Asynchronous exception masking in an implementation that includes EL3 but not EL2*.
- *Asynchronous exception masking in an implementation that includes both EL2 and EL3* on page G1-3805.
- *Summary of the asynchronous exception masking controls* on page G1-3805.

#### Asynchronous exception masking in an implementation that includes EL2 but not EL3

The HCR.{AMO, IMO, FMO} bits modify the effect of the PSTATE.{A, I, F} bits. When the value of an HCR.{AMO, IMO, FMO} mask override bit is 1, the value of the corresponding PSTATE.{A, I, F} bit is ignored when the exception is taken from a Non-secure mode other than Hyp mode.

#### Asynchronous exception masking in an implementation that includes EL3 but not EL2

The SCR.{AW, FW} bits modify the effect of the PSTATE.{A, F} bits. When the value of one of the SCR.{AW, FW} bits is 0, the corresponding PSTATE bit is ignored when both of the follow apply:

- The corresponding exception is taken from Non-secure state.
- The value of the corresponding SCR.{EA, FIQ} bit is 1, routing the exception to EL3. This means the exception is routed to Monitor mode if EL3 is using AArch32.

**Note**

Whenever the value of PSTATE.I is 1, IRQ exceptions are masked and cannot be taken.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Asynchronous exception masking in an implementation that includes both EL2 and EL3

When the value of an HCR.{AMO, IMO, FMO} mask override bit is 1, the value of the corresponding PSTATE.{A, I, F} bit is ignored when both of the following apply:

- The exception is taken from Non-secure state.
- Either:
  - The corresponding SCR.{EA, IRQ, FIQ} bit routes the exception to Monitor mode.
  - The exception is taken from a Non-secure mode other than Hyp mode.

In addition, when the value of an SCR.{AW, FW} bit is 0, the value of the corresponding PSTATE.{A, F} bit is ignored when all of the following apply:

- The exception is taken from Non-secure state.
- The corresponding SCR.{EA, FIQ} bit routes the exception to Monitor mode.
- The corresponding HCR.{AMO, FMO} mask override bit is set to 0.

### Summary of the asynchronous exception masking controls

The tables in this section show the masking controls for each of the PSTATE.{A, I, F} bits. For an implementation that does not include all of the exception levels:

**If the implementation includes only EL1 and EL0**

The PSTATE bits cannot be masked. The behavior is as shown in the *Secure* row of the tables.

**If the implementation includes EL2 but not EL3**

The behavior is as shown in the *Non-secure* table rows when the control bits in the SCR are both 0.

**If the implementation includes EL3 but not EL2**

The behavior is as shown in the table rows where the control bit in the HCR is 0.

Table G1-15 shows the controls of the masking of asynchronous exceptions by PSTATE.A.

**Table G1-15 Control of masking by PSTATE.A**

| Security state | HCR.AMO | SCR.EA | SCR.AW | Mode | PSTATE.A |
|---|---|---|---|---|---|
| Secure | x | x | x | x | Masks asynchronous aborts, when set to 1 |
| Non-secure | 0 | 0 | x | x | Masks asynchronous aborts, when set to 1 |
| | | 1 | 0 | x | Ignored |
| | | | 1 | x | Masks asynchronous aborts, when set to 1 |
| | 1 | x | x | Not Hyp | Ignored |
| | | 0 | x | Hyp | Masks asynchronous aborts, when set to 1 |
| | | 1 | x | x | Ignored |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-16 shows the controls of the masking of FIQ exceptions by PSTATE.F:

**Table G1-16 Control of masking by PSTATE.I**

| Security state | HCR.IMO | SCR.IRQ | Mode | PSTATE.I |
|---|---|---|---|---|
| Secure | x | x | x | Masks IRQs, when set to 1 |
| Non-secure | 0 | x | x | Masks IRQs, when set to 1 |
| | 1 | x | Not Hyp | Ignored |
| | | 0 | Hyp | Masks IRQs, when set to 1 |
| | | 1 | x | Ignored |

Table G1-17 shows the controls of the masking of FIQ exceptions by PSTATE.F:

**Table G1-17 Control of masking by PSTATE.F**

| Security state | HCR.FMO | SCR.FIQ | SCR.FW | Mode | PSTATE.F |
|---|---|---|---|---|---|
| Secure | x | x | x | x | Masks FIQs, when set to 1 |
| Non-secure | 0 | 0 | x | x | Masks FIQs, when set to 1 |
| | | 1 | 0 | x | Ignored |
| | | | 1 | x | Masks FIQs, when set to 1 |
| | 1 | x | x | Not Hyp | Ignored |
| | | 0 | x | Hyp | Masks FIQs, when set to 1 |
| | | 1 | x | x | Ignored |

### G1.13.4 Asynchronous exception routing and masking with higher Exception levels using AArch64

*Asynchronous exception routing controls* on page G1-3803 and *Asynchronous exception masking controls* on page G1-3804 give full descriptions of the routing and masking of the asynchronous exceptions when all Exception levels are using AArch32. However, when EL0 and EL1 are using AArch32:

- As already described, the SCR and HCR controls might be from Exception levels that are using AArch64.
- If EL3 is using AArch64, or EL2 is using AArch64, there are some changes to the asynchronous exception behaviors.

Therefore, the following sections summarize the asynchronous exception behaviors, taking account of the Execution state being used at EL2 and EL3:

- *Summary of physical interrupt routing*.
- *Summary of physical interrupt masking* on page G1-3808.

#### Summary of physical interrupt routing

The following tables show the routing of physical interrupts. Table G1-18 on page G1-3807 shows the routing of physical FIQ interrupts, Table G1-19 on page G1-3807 shows the routing of physical IRQ interrupts, and Table G1-20 on page G1-3808 shows the routing of physical asynchronous aborts.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In these tables, for exceptions that must be taken to an Exception level that is using AArch32, the table shows the target Exception level and PE mode. In these entries, *Mon* indicates Monitor mode, and *Abt* indicates Abort mode.

**Table G1-18 Routing of physical FIQ exceptions**

| EL3 Execution state | Control bits | | | Target when take from: | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | SCR | | HCR | Non-secure | | | Secure | | |
| | FIQ | RW[a] | FMO[b] | EL0 | EL1 | EL2 | EL0 | EL1[c] | EL3 |
| AArch32 | 0 | x | 0 | EL1 FIQ | EL1 FIQ | EL2 Hyp | EL3 FIQ | - | EL3 FIQ |
| | | | 1 | EL2 Hyp | EL2 Hyp | EL2 Hyp | EL3 FIQ | - | EL3 FIQ |
| | 1 | x | x | EL3 Mon | EL3 Mon | EL3 Mon | EL3 Mon | - | EL3 Mon |
| AArch64 | 0 | 0 | 0 | EL1 FIQ | EL1 FIQ | EL2 Hyp | EL1 FIQ | EL1 FIQ | P[d] |
| | | x | 1 | EL2[e] | EL2[e] | EL2[e] | EL1[f] | EL1[f] | P[d] |
| | | 1 | 0 | EL1 | EL1 | P[d] | EL1 | EL1 | P[d] |
| | 1 | x | x | EL3 | EL3 | EL3 | EL3 | EL3 | EL3 |

a. SCR_EL3.RW. When 1, the next lower Exception level is using AArch64. This control is not present when EL3 is using AArch32.
b. When the value of HCR.TGE is 1, the effective value of this bit is 1.
c. When EL3 is using AArch32, the only Secure Exception levels are EL0 and EL3.
d. Interrupt is not taken, but remains pending.
e. If EL2 is using AArch32, taken to Hyp mode.
f. If EL1 is using AArch32.taken to Abort mode.

**Table G1-19 Routing of physical IRQ exceptions**

| EL3 Execution state | Control bits | | | Target when take from: | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | SCR | | HCR | Non-secure | | | Secure | | |
| | IRQ | RW[a] | IMO[b] | EL0 | EL1 | EL2 | EL0 | EL1[c] | EL3 |
| AArch32 | 0 | x | 0 | EL1 IRQ | EL1 IRQ | EL2 Hyp | EL3 IRQ | - | EL3 IRQ |
| | | | 1 | EL2 Hyp | EL2 Hyp | EL2 Hyp | EL3 IRQ | - | EL3 IRQ |
| | 1 | x | x | EL3 Mon | EL3 Mon | EL3 Mon | EL3 Mon | - | EL3 Mon |
| AArch64 | 0 | 0 | 0 | EL1 IRQ | EL1 IRQ | EL2 Hyp | EL1 IRQ | EL1 IRQ | P[d] |
| | | x | 1 | EL2[e] | EL2[e] | EL2[e] | EL1[f] | EL1[f] | P[d] |
| | | 1 | 0 | EL1 | EL1 | P[d] | EL1 | EL1 | P[d] |
| | 1 | x | x | EL3 | EL3 | EL3 | EL3 | EL3 | EL3 |

a. SCR_EL3.RW. When 1, the next lower Exception level is using AArch64. This control is not present when EL3 is using AArch32.
b. When the value of HCR.TGE is 1, the effective value of this bit is 1.
c. When EL3 is using AArch32, the only Secure Exception levels are EL0 and EL3.
d. Interrupt is not taken, but remains pending.
e. If EL2 is using AArch32, taken to Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

f. If EL1 is using AArch32.taken to Abort mode.

**Table G1-20 Routing of physical Asynchronous aborts**

| EL3 Execution state | Control bits | | | Target when take from: | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | SCR | | HCR | Non-secure | | | Secure | | |
| | EA | RW[a] | AMO[b] | EL0 | EL1 | EL2 | EL0 | EL1[c] | EL3 |
| AArch32 | 0 | x | 0 | EL1 Abt | EL1 Abt | EL2 Hyp | EL3 Abt | - | EL3 Abt |
| | | | 1 | EL2 Hyp | EL2 Hyp | EL2 Hyp | EL3 Abt | - | EL3 Abt |
| | 1 | x | x | EL3 Mon | EL3 Mon | EL3 Mon | EL3 Mon | - | EL3 Mon |
| AArch64 | 0 | 0 | 0 | EL1 Abt | EL1 Abt | EL2 Hyp | EL1 Abt | EL1Abt | P[d] |
| | | x | 1 | EL2[e] | EL2[e] | EL2[e] | EL1[f] | EL1[f] | P[d] |
| | | 1 | 0 | EL1 | EL1 | P[d] | EL1 | EL1 | P[d] |
| | 1 | x | x | EL3 | EL3 | EL3 | EL3 | EL3 | EL3 |

a. SCR_EL3.RW. When 1, the next lower Exception level is using AArch64. This control is not present when EL3 is using AArch32.
b. When the value of HCR.TGE is 1, the effective value of this bit is 1.
c. When EL3 is using AArch32, the only Secure Exception levels are EL0 and EL3.
d. Interrupt is not taken, but remains pending.
e. If EL2 is using AArch32, taken to Hyp mode.
f. If EL1 is using AArch32.taken to Abort mode.

### Summary of physical interrupt masking

The following tables show the masking of physical interrupts. Table G1-21 on page G1-3809 shows the masking of physical FIQ interrupts, Table G1-22 on page G1-3809 shows the masking of physical IRQ interrupts, and Table G1-23 on page G1-3810 shows the masking of physical asynchronous aborts. In these tables:

**M** Indicates that the exception is masked when the value of the PSTATE mask bit is 1.

**T** Indicates that the exception is taken, regardless of the value of the PSTATE mask bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**P** Indicates that the exception is not taken but remains pending. The value of the PSTATE mask bit has no effect on this behavior.

**Table G1-21 Masking of physical FIQ exceptions**

| EL3 Execution state | Control bits | | | | Effect of PSTATE.F[a] mask in Exception level | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | SCR | | | HCR | Non-secure | | | Secure | | |
| | FIQ | FW | RW[b] | FMO[c] | EL0 | EL1 | EL2 | EL0 | EL1[d] | EL3 |
| AArch32 | 0 | x | x | 0 | M | M | M | M | - | M |
| | | | | 1 | T | T | M | M | - | M |
| | 1 | 0 | x | 0 | T | T | T | M | - | M |
| | | 1 | x | 0 | M | M | M | M | - | M |
| | | x | x | 1 | T | T | T | M | - | M |
| AArch64 | 0 | x | 0 | 0 | M | M | M | M | M | P |
| | | | | 1 | T | T | M | M | M | P |
| | | | 1 | 0 | M | M | P | M | M | P |
| | | | | 1 | T | T | M | M | M | P |
| | 1 | x | x | x | T | T | T | T | T | M |

a. Or the PSTATE.F mask in an Exception level that is using AArch64.
b. SCR_EL3 only, this control is not present when EL3 is using AArch32. When the value of RW is 1, the next lower Exception level is using AArch64.
c. When the value of HCR.TGE is 1, the effective value of this bit is 1.
d. When EL3 is using AArch32, the only Secure Exception levels are EL0 and EL3.

**Table G1-22 Masking of physical IRQ exceptions**

| EL3 Execution state | Control bits | | | Effect of PSTATE.I[a] mask in Exception level | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | SCR | | HCR | Non-secure | | | Secure | | |
| | IRQ | RW[b] | IMO[c] | EL0 | EL1 | EL2 | EL0 | EL1[d] | EL3 |
| AArch32 | 0 | x | 0 | M | M | M | M | - | M |
| | | | 1 | T | T | M | M | - | M |
| | 1 | x | 0 | M | M | M | M | - | M |
| | | | 1 | T | T | T | M | - | M |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G1-22 Masking of physical IRQ exceptions (continued)**

| EL3 Execution state | Control bits | | | Effect of PSTATE.I[a] mask in Exception level | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | SCR | | HCR | Non-secure | | | Secure | | |
| | IRQ | RW[b] | IMO[c] | EL0 | EL1 | EL2 | EL0 | EL1[d] | EL3 |
| AArch64 | 0 | 0 | 0 | M | M | M | M | M | P |
| | | | 1 | T | T | M | M | M | P |
| | | 1 | 0 | M | M | P | M | M | P |
| | | | 1 | T | T | M | M | M | P |
| | 1 | x | x | T | T | T | T | T | M |

a. Or the PSTATE.I mask in an Exception level that is using AArch64.
b. SCR_EL3 only, this control is not present when EL3 is using AArch32. When the value of RW is 1, the next lower Exception level is using AArch64.
c. When the value of HCR.TGE is 1, the effective value of this bit is 1.
d. When EL3 is using AArch32, the only Secure Exception levels are EL0 and EL3.

**Table G1-23 Masking of physical Asynchronous aborts**

| EL3 Execution state | Control bits | | | | Effect of PSTATE.A[a] mask in Exception level | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | SCR | | | HCR | Non-secure | | | Secure | | |
| | EA | AW | RW[b] | AMO[c] | EL0 | EL1 | EL2 | EL0 | EL1[d] | EL3 |
| AArch32 | 0 | x | x | 0 | M | M | M | M | - | M |
| | | | | 1 | T | T | M | M | - | M |
| | 1 | 0 | x | 0 | T | T | T | M | - | M |
| | | 1 | x | 0 | M | M | M | M | - | M |
| | | x | x | 1 | T | T | T | M | - | M |
| AArch64 | 0 | x | 0 | 0 | M | M | M | M | M | P |
| | | | | 1 | T | T | M | M | M | P |
| | | | 1 | 0 | M | M | P | M | M | P |
| | | | | 1 | T | T | M | M | M | P |
| | 1 | x | x | x | T | T | T | T | T | M |

a. Or the PSTATE.A mask in an Exception level that is using AArch64.
b. SCR_EL3 only, this control is not present when EL3 is using AArch32. When the value of RW is 1, the next lower Exception level is using AArch64.
c. When the value of HCR.TGE is 1, the effective value of this bit is 1.
d. When EL3 is using AArch32, the only Secure Exception levels are EL0 and EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.14 AArch32 state exception descriptions

*Handling exceptions that are taken to an Exception level using AArch32* on page G1-3774 gives general information about exception handling. This section describes each of the exceptions, in the following subsections:

- *Undefined Instruction exception*.
- *Monitor Trap exception* on page G1-3813.
- *Hyp Trap exception* on page G1-3814.
- *Supervisor Call (SVC) exception* on page G1-3815.
- *Secure Monitor Call (SMC) exception* on page G1-3816.
- *Hypervisor Call (HVC) exception* on page G1-3817.
- *Prefetch Abort exception* on page G1-3818.
- *Data Abort exception* on page G1-3821.
- *Virtual Abort exception* on page G1-3824.
- *IRQ exception* on page G1-3825.
- *Virtual IRQ exception* on page G1-3826.
- *FIQ exception* on page G1-3827.
- *Virtual FIQ exception* on page G1-3828.

*Additional pseudocode functions for exception handling* on page G1-3829 gives additional pseudocode that is used in the pseudocode descriptions of a number of the exceptions.

### G1.14.1 Undefined Instruction exception

An Undefined Instruction exception might be caused by:

- A coprocessor instruction that is not accessible because of the settings in one or more of the CPACR, NSACR, HCPTR, and DBGDSCRext.
- A coprocessor instruction that is not implemented.
- A coprocessor instruction that causes an exception during execution. This includes:
  - Trapped floating-point exceptions that are taken to AArch32, if an implementation supports these traps. See *Floating-point exceptions* on page E1-2329.
  - Execution of certain floating-point instructions when one or both of the FPSCR.{Stride, Len} fields in nonzero, in an implementation in which those fields are RW. The description of FPEXC specifies the instructions to which this applies.
- An instruction that is UNDEFINED.

**Note**

The Undefined Instruction exception is taken using offset `0x04` in the Hyp, Secure, or Non-secure vector table. In the Monitor vector table this offset is used for the Monitor Trap exception. See *Monitor Trap exception* on page G1-3813 and *The vector tables and exception offsets* on page G1-3775.

By default, an Undefined Instruction exception is taken to Undefined mode, but an Undefined Instruction exception can be taken to EL2, meaning it is taken to Hyp mode if EL2 is using AArch32, see *The PE mode to which the Undefined Instruction exception is taken* on page G1-3812.

The Undefined Instruction exception can provide:

- Signaling of an illegal instruction execution.
- *Lazy context switching* of coprocessor registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The preferred return address for an Undefined Instruction exception is the address of the instruction that generated the exception. This return is performed as follows:

- If returning from Secure or Non-secure Undefined mode, the exception return uses the SPSR and LR_und values generated by the exception entry, as follows:
  - If SPSR.T is 0, indicating that the exception occurred in A32 state, the return uses an exception return instruction with a subtraction of 4.
  - If SPSR.T is 1, indicating that the exception occurred in T32 state, the return uses an exception return instruction with a subtraction of 2.
- If returning from Hyp mode, the exception return is performed by an ERET instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see *Exception return to an Exception level using AArch32* on page G1-3796.

**Note**

If handling the Undefined Instruction exception requires instruction emulation, followed by return to the next instruction after the instruction that caused the exception, the instruction emulator must use the instruction length to calculate the correct return address, and to calculate the updated values of the IT bits if necessary.

### The PE mode to which the Undefined Instruction exception is taken

Figure G1-4 shows how the implementation, state, and configuration options determine the PE mode to which an Undefined Instruction exception is taken.

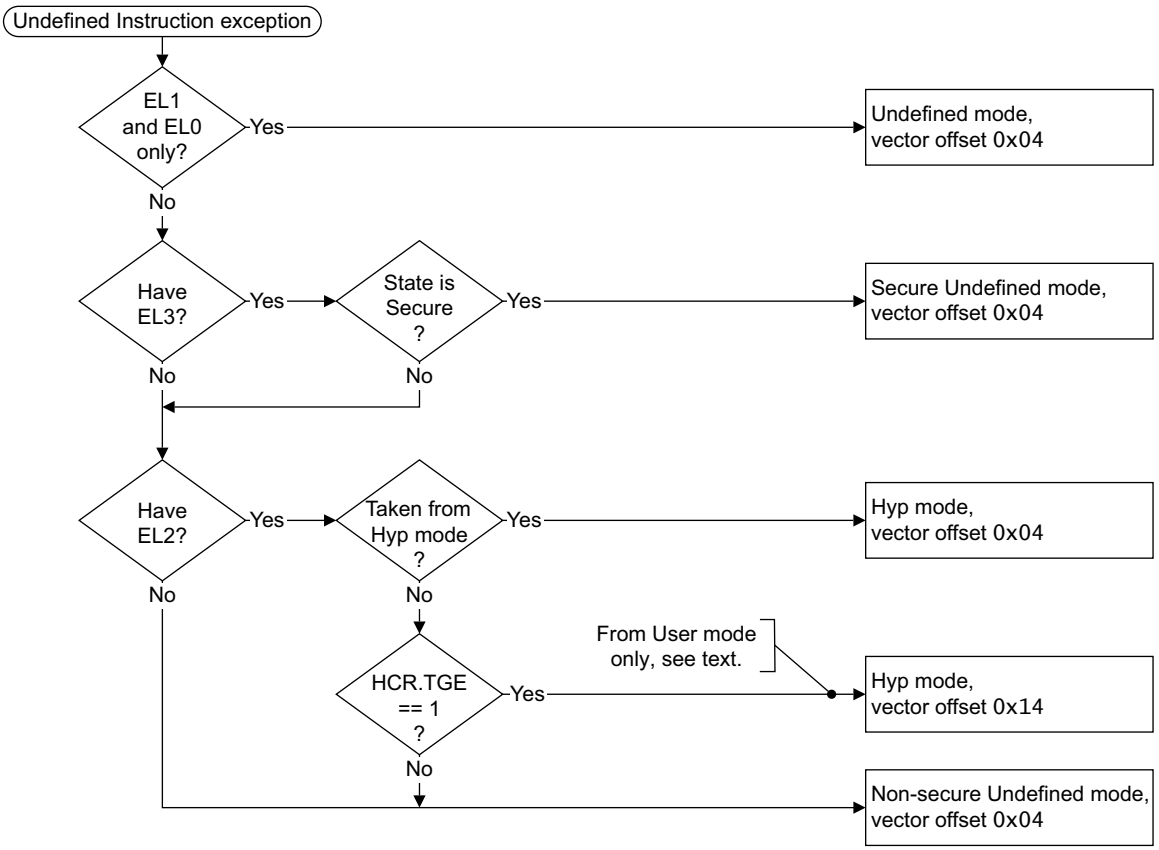



**Figure G1-4 The PE mode the Undefined Instruction exception is taken to**

See also *UNPREDICTABLE cases when the value of HCR.TGE is 1* on page G1-3788.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Pseudocode description of taking the Undefined Instruction exception

The `AArch32.TakeUndefInstrException()` pseudocode procedure describes how the PE takes the exception.

### Conditional execution of undefined instructions

The conditional execution rules described in *Conditional execution* on page F2-2447 apply to all instructions. This includes undefined instructions and other instructions that would cause entry to the Undefined Instruction exception.

If such an instruction fails its condition check, the behavior depends on the potential cause of entry to the Undefined Instruction exception, as follows:

- If the potential cause is the execution of the instruction itself and depends on data values used by the instruction, the instruction executes as a `NOP` and does not cause an Undefined Instruction exception.
- If the potential cause is the execution of an earlier coprocessor instruction, or the execution of the instruction itself without dependence on the data values used by the instruction, it is IMPLEMENTATION DEFINED whether the instruction executes as a `NOP` or causes an Undefined Instruction exception.

  An implementation must handle all such cases in the same way.

**Note**

Before ARMv7, all implementations executed any instruction that failed its condition check as a `NOP`, even if it would otherwise have caused an Undefined Instruction exception. An Undefined Instruction handler written for these implementations might assume without checking that the undefined instruction passed its condition check. Such an Undefined Instruction handler is likely to need rewriting, to check the condition is passed, before it functions correctly on all AArch32 implementations.

### Interaction of UNPREDICTABLE and UNDEFINED instruction behavior

If this manual describes an instruction as both UNPREDICTABLE and UNDEFINED then the instruction is UNPREDICTABLE.

**Note**

An example of this is where both:

- An instruction, or instruction class, is made UNDEFINED by some general principle, or by a configuration field.
- A particular encoding of that instruction or instruction class is specified as UNPREDICTABLE.

## G1.14.2 Monitor Trap exception

The Monitor Trap exception is implemented only as part of EL3, and can be generated only if EL3 is using AArch32.

**Note**

The Monitor Trap exception is taken using offset `0x04` in the Monitor vector table. In the other vector tables, this offset is used for the Undefined Instruction exception. See *Undefined Instruction exception* on page G1-3811 and *The vector tables and exception offsets* on page G1-3775.

A Monitor Trap exception is generated if the PE is running in a mode other than Monitor mode, and commits for execution a `WFI` or `WFE` instruction that would otherwise cause suspension of execution when:

- In the case of the `WFI` instruction, the value of the SCR.TWI bit is 1.
- In the case of the `WFE` instruction, the value of the SCR.TWE bit is 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The preferred return address for a Monitor Trap exception is the address of the instruction that generated the exception. The exception return uses the SPSR and LR_mon values generated by the exception entry, as follows:

- If SPSR.T is 0, indicating that the exception occurred in A32 state, the return uses an exception return instruction with a subtraction of 4.
- If SPSR.T is 1, indicating that the exception occurred in T32 state, the return uses an exception return instruction with a subtraction of 2.

For more information, see *Exception return to an Exception level using AArch32* on page G1-3796.

#### The PE mode to which the Monitor Trap exception is taken

When EL3 is using AArch32, a Monitor Trap exception is taken to Monitor mode, using a vector offset of 0x04 from the Monitor exception base address.

#### Pseudocode description of taking the Monitor Trap exception

The AArch32.TakeMonitorTrapException() pseudocode procedure describes how the PE takes the exception.

### G1.14.3 Hyp Trap exception

The Hyp Trap exception provides the standard mechanism for trapping Guest OS functions to the hypervisor.

The Hyp Trap exception is implemented only as part of EL2 and can be generated only if EL2 is using AArch32.

A Hyp Trap exception is generated if the PE is running in a Non-secure mode other than Hyp mode, and commits for execution an instruction that is trapped to Hyp mode. Instruction traps are enabled by setting bits to 1 in the HCR, HCPTR, HDCR, or HSTR. For more information see *EL2 configurable controls* on page G1-3855.

Traps to Hyp mode never apply in Secure state, regardless of the value of the SCR.NS bit.

The preferred return address for a Hyp Trap exception is the address of the trapped instruction. The exception return is performed by an ERET instruction, using the SPSR and ELR_hyp values generated by the exception entry.

**Note**

The SPSR and ELR_hyp values generated on exception entry can be used, without modification, for an exception return to re-execute the trapped instruction. If the exception handler emulates the trapped instruction, and must return to the following instruction, the emulation of the instruction must include modifying ELR_hyp, and possibly updating SPSR_hyp.

When the PE enters the handler for a Hyp Trap exception, the HSR holds syndrome information for the exception. For more information see *Use of the HSR* on page G4-4105.

#### The PE mode to which the Hyp Trap exception is taken

A Hyp Trap exception is taken to Hyp mode, using a vector offset of 0x14 from the Hyp exception base address.

#### Pseudocode description of taking the Hyp Trap exception

The AArch32.TakeHypTrapException() pseudocode procedure describes how the PE takes the exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G1.14.4 Supervisor Call (SVC) exception

The Supervisor Call instruction, SVC, requests a supervisor function, typically to request an operating system function. When EL1 is using AArch32, executing an SVC instruction causes the PE to enter Supervisor mode. For more information, see *SVC* on page F6-3103.

**Note**

In an implementation that includes EL2:

- When an SVC instruction is executed in Hyp mode, the Supervisor Call exception is taken to Hyp mode. For more information see *SVC* on page F6-3103.
- When the HCR.TGE bit is set to 1, the Supervisor Call exception generated by execution of an SVC instruction in Non-secure User mode is routed to Hyp mode. For more information, see *Supervisor Call exception, when the value of HCR.TGE is 1* on page G1-3791.

By default, a Supervisor Call exception is taken to Supervisor mode, but a Supervisor Call exception can be taken to EL2, meaning it is taken to Hyp mode if EL2 is using AArch32, see *The PE mode to which the Supervisor Call exception is taken*.

The preferred return address for a Supervisor Call exception is the address of the next instruction after the SVC instruction. This return is performed as follows:

- If returning from Secure or Non-secure Supervisor mode, the exception return uses the SPSR and LR_svc values generated by the exception entry, in an exception return instruction without subtraction.
- If returning from Hyp mode, the exception return is performed by an ERET instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see *Exception return to an Exception level using AArch32* on page G1-3796.

#### The PE mode to which the Supervisor Call exception is taken

Figure G1-5 on page G1-3816 shows how the implementation, state, and configuration options determine the PE mode to which a Supervisor Call exception is taken.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

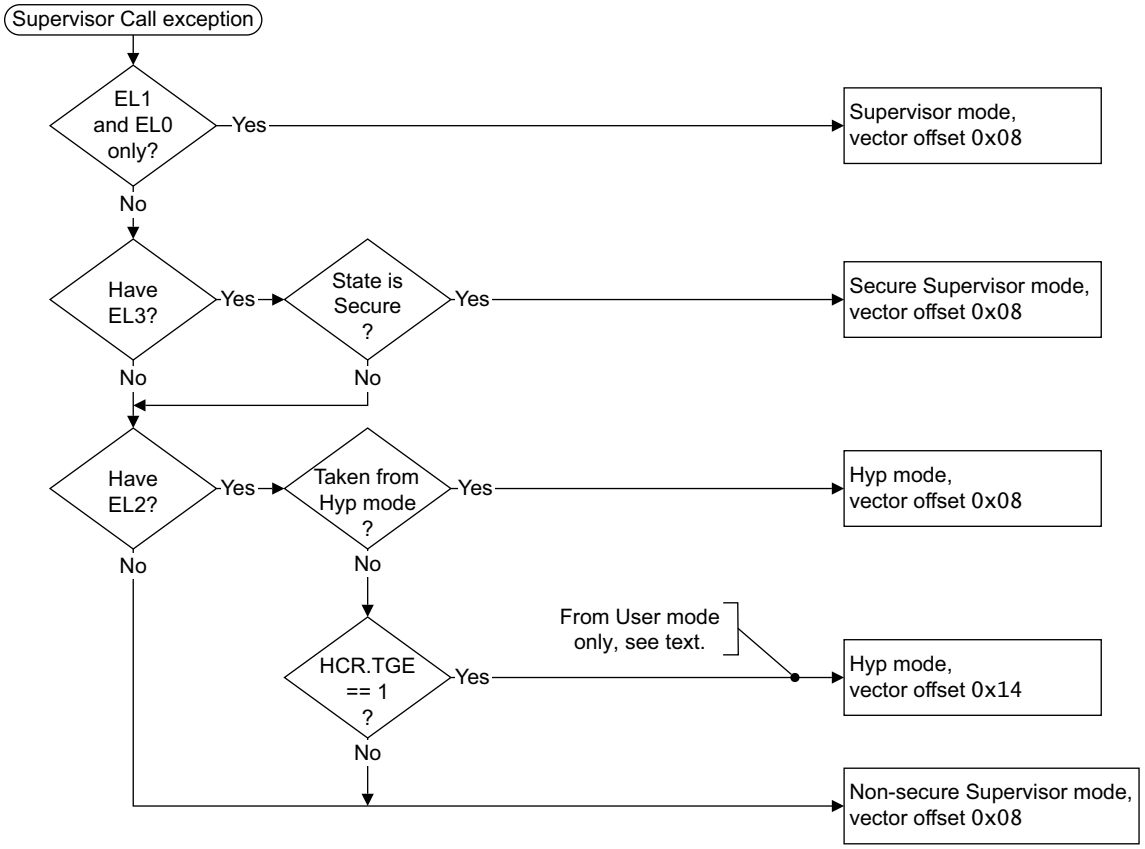



**Figure G1-5 The PE mode the Supervisor Call exception is taken to**

See also *UNPREDICTABLE cases when the value of HCR.TGE is 1* on page G1-3788.

### Pseudocode description of taking the Supervisor Call exception

The `AArch32.TakeSVCException()` pseudocode procedure describes how the PE takes the exception.

## G1.14.5 Secure Monitor Call (SMC) exception

The Secure Monitor Call exception is implemented only as part of EL3. When EL3 is using AArch32, the exception is taken to Monitor mode.

The Secure Monitor Call instruction, `SMC`, requests a Secure Monitor function. When EL3 is using AArch32, executing an `SMC` instruction causes the PE to enter Monitor mode. For more information, see *SMC* on page F6-2979.

**Note**

In an implementation that includes EL2, execution of an `SMC` instruction in a Non-secure EL1 mode can be trapped to EL2. When EL2 is using AArch32, this means that when HCR.TSC 1, execution of an `SMC` instruction in a Non-secure EL1 mode generates a Hyp Trap Exception that is taken to Hyp mode. For more information see *Traps to Hyp mode of Non-secure EL1 execution of SMC instructions* on page G1-3861.

The preferred return address for a Secure Monitor Call exception is the address of the next instruction after the `SMC` instruction. This return is performed using the SPSR and LR_mon values generated by the exception entry, using an exception return instruction without a subtraction.

For more information, see *Exception return to an Exception level using AArch32* on page G1-3796.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The exception handler can return to the SMC instruction itself by returning using a subtraction of 4, without any adjustment to the SPSR.IT[7:0] bits. If it does this, the return occurs, then interrupts or external aborts might occur and be handled, then the SMC instruction is re-executed and another Secure Monitor Call exception occurs.

This relies on:

- The SMC instruction being used correctly, either outside an IT block or as the last instruction in an IT block, so that the SPSR.IT[7:0] bits indicate unconditional execution.
- The Secure Monitor Call handler not changing the result of the original conditional execution test for the SMC instruction.

### The PE mode to which the Secure Monitor Call exception is taken

The Secure Monitor Call exception is supported only as part of EL3. When EL3 is using AArch32, a Secure Monitor Call exception is taken to Monitor mode, using vector offset 0x08 from the Monitor exception base address.

**Note**

- An SMC instruction that is trapped to Hyp mode because HCR.TSC is set to 1 generates a Hyp Trap exception, see *The PE mode to which the Hyp Trap exception is taken* on page G1-3814.
- If EL3 is using AArch64 then *Security behavior in Exception levels using AArch32 when EL3 is using AArch64* on page G1-3786 describes the effect of executing an SMC instruction in a mode that is part of an Exception level that is using EL1.

### Pseudocode description of taking the Secure Monitor Call exception

The AArch32.TakeSMCException() pseudocode procedure describes how the PE takes the exception.

## G1.14.6 Hypervisor Call (HVC) exception

The Hypervisor Call exception is implemented only as part of EL2.

The Hypervisor Call instruction, HVC, requests a hypervisor function. When EL2 is using AArch32, executing an HVC instruction generates a Hypervisor Call exception that is taken to Hyp mode. For more information, see *HVC* on page F6-2686.

The preferred return address for a Hypervisor Call exception is the address of the next instruction after the HVC instruction. The exception return is performed by an ERET instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see *Exception return to an Exception level using AArch32* on page G1-3796.

When EL2 is using AArch32, executing an HVC instruction transfers the immediate argument of the instruction to the HSR. The exception handler retrieves the argument from the HSR, and therefore does not have to access the original HVC instruction. For more information see *Use of the HSR* on page G4-4105.

### The PE mode to which the Hypervisor Call exception is taken

The Hypervisor Call exception is supported only as part of EL2. When EL2 is using AArch32, a Hypervisor Call exception is taken to Hyp mode, using a vector offset that depends on the mode from which the exception is taken, as Figure G1-6 on page G1-3818 shows. This offset is from the Hyp exception base address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

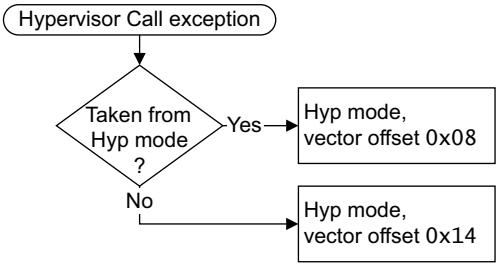



**Figure G1-6 The PE mode the Hypervisor Call exception is taken to**

### Pseudocode description of taking the Hypervisor Call exception

The `AArch32.TakeHVCException()` pseudocode procedure describes how the PE takes the exception.

## G1.14.7 Prefetch Abort exception

A Prefetch Abort exception can be generated by:

- A synchronous memory abort on an instruction fetch.

  **Note**

  Asynchronous aborts on instruction fetches are reported using the Data Abort exception, see *Data Abort exception* on page G1-3821.

  A Prefetch Abort exception entry is synchronous to the instruction whose fetch aborted.

  For more information about memory aborts see *VMSAv8-32 memory aborts* on page G4-4078.

- A Breakpoint, Vector Catch or Software Breakpoint Instruction exception, see Chapter G2 *AArch32 Self-hosted Debug*.

**Note**

If an implementation fetches instructions speculatively, it must handle a synchronous abort on such an instruction fetch by:

- Generating a Prefetch Abort exception only if the instruction would be executed in a simple sequential execution of the program.
- Ignoring the abort if the instruction would not be executed in a simple sequential execution of the program.

By default, when EL1 is using AArch32, a Prefetch Abort exception is taken to Abort mode, but a Prefetch Abort exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information, see *The PE mode to which the Prefetch Abort exception is taken* on page G1-3820.

The preferred return address for a Prefetch Abort exception is the address of the aborted instruction. This return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_abt and LR_abt if returning from Abort mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information, see *Exception return to an Exception level using AArch32* on page G1-3796.

### Prefetch Abort exception reporting a Misaligned PC exception

A Misaligned PC exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception, and:

**If the exception is taken to EL1 using AArch32 or EL3 using AArch32**

- The IFSR indicates the cause of the exception:
  - If the value of TTBCR.EAE is 0, IFSR.FS takes the value `0b00001`.
  - If the value of TTBCR.EAE is 1, IFSR.STATUS takes the value `0b100001`.
- IFAR holds the value of the address that faulted, including the misaligned low order bit or bits.
- R14_abt holds the address that faulted, including the misaligned low order bit or bits, with the standard offset for a Prefetch Abort exception.

**If the exception is taken to EL2 using AArch32**

- HSR.EC takes the value `0b100010`.
- HSR.IL is UNKNOWN.
- HSR.ISS is RES0.
- HIFAR and ELR_hyp each hold the value of the address that faulted, including the misaligned low order bit or bits.

For a Misaligned PC exception taken to an Exception level that is using AArch32:

- If the exception occurred because of the CONSTRAINED UNPREDICTABLE behavior of a branch to an unaligned PC value, as described in *Branching to an unaligned PC* on page K1-5424, then bit[0] of the faulting address is forced to zero, and therefore the misalignment is because the value of bit[1] of this address is 1.
- If the exception occurred on an exit from Debug state, as described in *Exiting Debug state* on page H2-4858, then it is CONSTRAINED UNPREDICTABLE whether bit[0] of the faulting address is forced to zero.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### The PE mode to which the Prefetch Abort exception is taken

Figure G1-7 shows how the implementation, state, and configuration options determine the PE mode to which a Prefetch Abort exception is taken.

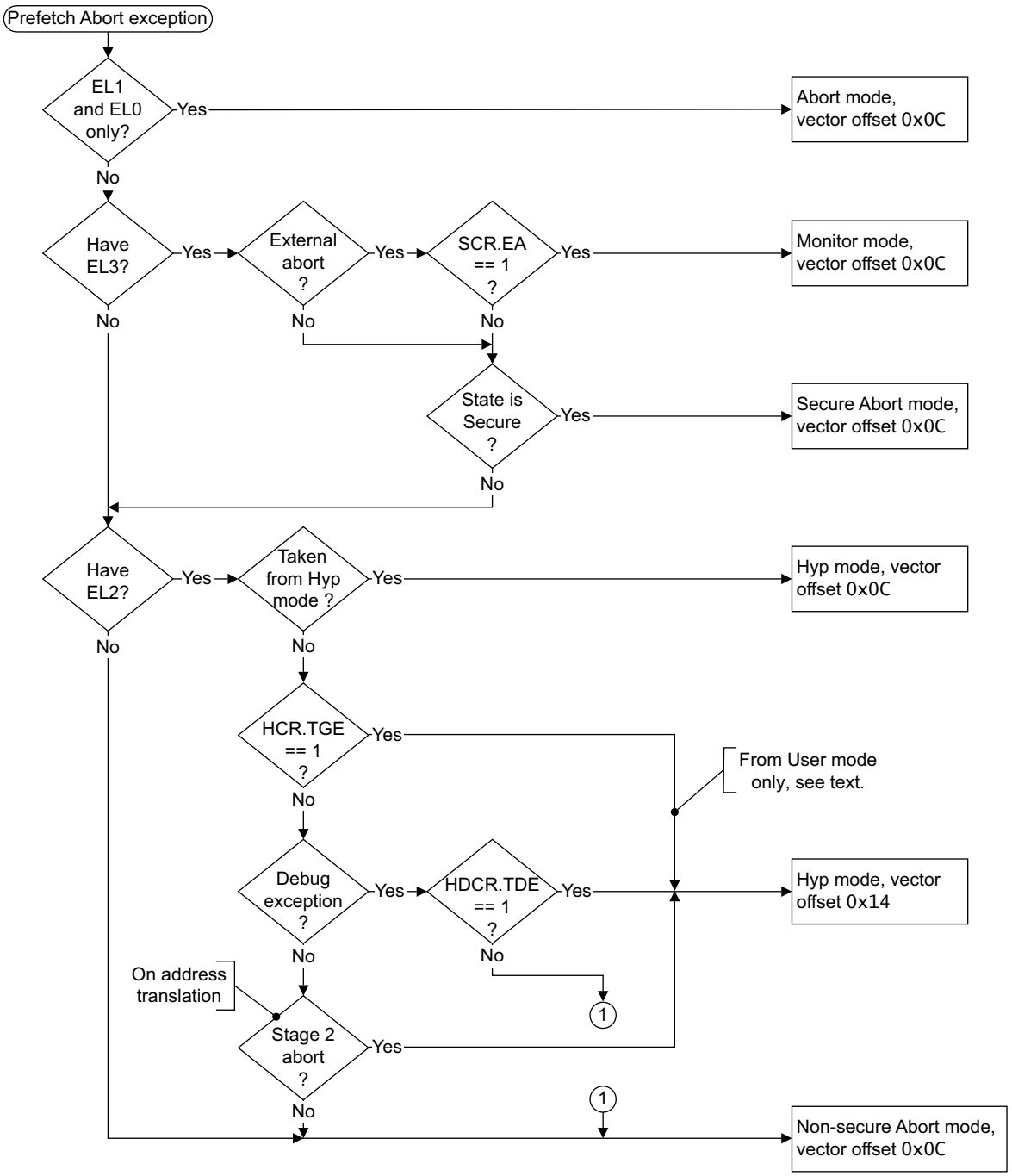



**Figure G1-7 The PE mode the Prefetch Abort exception is taken to**

See also *UNPREDICTABLE cases when the value of HCR.TGE is 1* on page G1-3788.

### Pseudocode description of taking the Prefetch Abort exception

The `AArch32.TakePrefetchAbortException()` pseudocode procedure describes how the PE takes the exception.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### G1.14.8 Data Abort exception

A Data Abort exception can be generated by:

- A synchronous abort on a data read or write memory access. Exception entry is synchronous to the instruction that generated the memory access.
- An asynchronous abort. The memory access that caused the abort can be any of:
  - A data read or write access.
  - An instruction fetch.
  - In a VMSA memory system, a translation table access.

  Exception entry occurs asynchronously, and is similar to an interrupt.

  As described in *Asynchronous exception masking controls* on page G1-3804, asynchronous aborts can be masked. When this happens, a generated asynchronous abort is not taken until it is not masked.

  **Note**

  There are no asynchronous internal aborts in the ARM architecture, so asynchronous aborts are always asynchronous external aborts.

- A watchpoint, see *Watchpoint exceptions* on page G2-3923.

By default, when EL1 is using AArch32 a Data Abort exception is taken to Abort mode, but a Data Abort exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information see *The PE mode to which the Data Abort exception is taken* on page G1-3822.

For more information about memory aborts see *VMSAv8-32 memory aborts* on page G4-4078.

The preferred return address for a Data Abort exception is the address of the instruction that generated the aborting memory access, or the address of the instruction following the instruction boundary at which an asynchronous Data Abort exception was taken. This return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 8. This means using:
  - SPSR_abt and LR_abt if returning from Abort mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.

For more information, see *Exception return to an Exception level using AArch32* on page G1-3796.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### The PE mode to which the Data Abort exception is taken

Figure G1-8 shows the determination of the mode to which a Data Abort exception is taken.

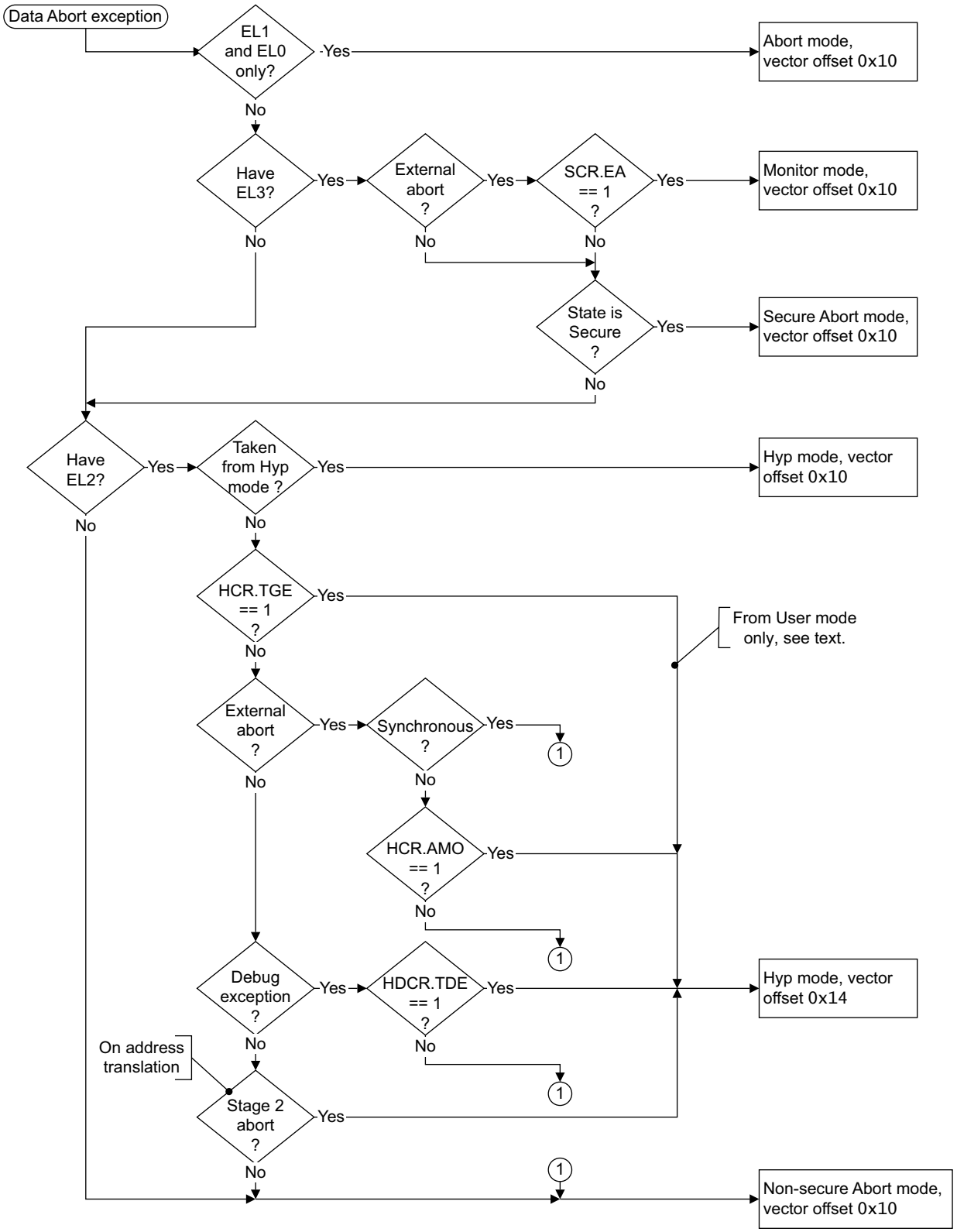



**Figure G1-8 The PE mode the Data Abort exception is taken to**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See also *UNPREDICTABLE cases when the value of HCR.TGE is 1* on page G1-3788.

### Pseudocode description of taking the Data Abort exception

The `AArch32.TakeDataAbortException()` pseudocode procedure describes how the PE takes the exception.

### Effects of data-aborted instructions

An instruction that accesses data memory can modify memory by storing one or more values. If the execution of such an instruction generates a Data Abort exception, or causes Debug state entry because of a watchpoint set on the instruction, the value of each memory location that the instruction stores to is:

- Unchanged for any location for which one of the following applies:
  - An MMU fault is generated.
  - A Watchpoint is generated.
  - An external abort is generated, if that external abort is taken synchronously.
- UNKNOWN for any location for which no exception and no debug event is generated.

If the access to a memory location generates an external abort that is taken asynchronously, it is outside the scope of the architecture to define the effect of the store on that memory location, because this depends on the system-specific nature of the external abort. However, in general, ARM recommends that such locations are unchanged.

For external aborts and Watchpoints, where in principle faulting could be identified at byte or halfword granularity, the size of a location in this definition is the size for which a memory access is single-copy atomic.

In AArch32 state, instructions that access data memory can modify registers in the following ways:

- By loading values into one or more of the general-purpose registers. The registers loaded can include the PC.
- By loading values into one or more of the registers in the Advanced SIMD and floating-point register file.
- By specifying *base register writeback*, in which the base register used in the address calculation has a modified value written to it. All instructions that support base register writeback have UNPREDICTABLE results if base register writeback is specified with the PC as the base register. Only general-purpose registers can be modified reliably in this way.
- By a direct or indirect write to one or more coprocessor registers, for example:
  - An `LDC` instruction is a *direct write* to a coprocessor register with a value read from memory.
  - An `STC` instruction that reads DBGDTRTXint makes an *indirect write* to DBGDSCRint.RXfull.
- By modifying PSTATE.

If the execution of such an instruction generates a synchronous Data Abort exception, the following rules determine the values left in these registers:

- On entry to the Data Abort exception handler:
  - The PC value is the Data Abort vector address, see *Exception vectors and the exception base address* on page G1-3774.
  - The LR_abt value is determined from the address of the aborted instruction.

  Neither value is affected by the results of any load specified by the instruction.
- The base register is restored to its original value if either:
  - The aborted instruction is a load and the list of registers to be loaded includes the base register.
  - The base register is being written back.
- If the instruction only loads one general-purpose register the value in that register is unchanged.
- If the instruction loads more than one general-purpose register, UNKNOWN values are left in destination registers other than the PC and the base register of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If the instruction affects any registers in the Advanced SIMD and floating-point register file, UNKNOWN values are left in the registers that are affected.
- PSTATE bits that are not defined as updated on exception entry retain their current value.
- If the instruction is a STREX, STREXB, STREXH, or STREXD, <Rd> is not updated.

After taking a Data Abort exception, the state of the exclusive monitors is UNKNOWN. Therefore, ARM strongly recommends that the abort handler performs a CLREX instruction, or a dummy STREX instruction, to clear the exclusive monitor state.

#### The ARM abort model

The abort model used by an ARM PE is described as a *Base Restored Abort Model*. This means that if a synchronous Data Abort exception is generated by executing an instruction that specifies base register writeback, the value in the base register is unchanged.

The abort model applies uniformly across all instructions.

### G1.14.9 Virtual Abort exception

The Virtual Abort exception is implemented only as part of EL2.

A Virtual Abort exception is generated if all of the following apply:

- The PE is in a Non-secure mode other than Hyp mode.
- The value of PSTATE.A is 0.
- Either:
  - EL2 is using AArch32 and the values of the HCR.{AMO, VA} bits are {1, 1}.
  - EL2 is using AArch64 and the values of the HCR_EL2.{AMO, VA} bits are {1, 1}.

The conditions for generating a Virtual Abort exception mean the exception is always:

- Taken from a Non-secure EL1 or EL0 mode.
- Taken to Non-secure Abort mode.

For more information see *Virtual exceptions when an implementation includes EL2* on page G1-3801.

**Note**

Because the Virtual Abort exception is always taken to Non-secure Abort mode, on exception entry the preferred return address is always saved to LR_abt.

The preferred return address for a Virtual Abort exception is the address of the instruction immediately after the instruction boundary where the exception was taken. This return is performed using the SPSR and LR_abt values generated by the exception entry, using an exception return instruction without subtraction.

#### The PE mode to which the Virtual Abort exception is taken

The Virtual Abort exception is supported only as part of EL2. A Virtual Abort exception is taken from a Non-secure EL1 or EL0 mode, and is taken to Non-secure Abort mode, using a vector offset of 0x10 from the Non-secure exception base address.

For more information about this exception see *Virtual exceptions when an implementation includes EL2* on page G1-3801.

#### Pseudocode description of taking the Virtual Asynchronous Abort exception

The AArch32.TakeVirtualAsynchAbortException() pseudocode procedure describes how the PE takes the exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G1.14.10 IRQ exception

The IRQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an IRQ interrupt request input to the PE.

When an IRQ exception is taken, exception entry is precise to an instruction boundary.

As described in *Asynchronous exception masking controls* on page G1-3804, IRQ exceptions can be masked. When this happens, a generated IRQ exception is not taken until it is not masked.

By default, when EL1 is using AArch32, an IRQ exception is taken to IRQ mode, but an IRQ exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information, see *The PE mode to which the physical IRQ exception is taken*.

The preferred return address for an IRQ exception is the address of the instruction following the instruction boundary at which the exception was taken. This return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_irq and LR_irq if returning from IRQ mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.

For more information, see *Exception return to an Exception level using AArch32* on page G1-3796.

#### The PE mode to which the physical IRQ exception is taken

Figure G1-9 on page G1-3826 shows how the implementation, state, and configuration options determine the mode to which an IRQ exception is taken.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

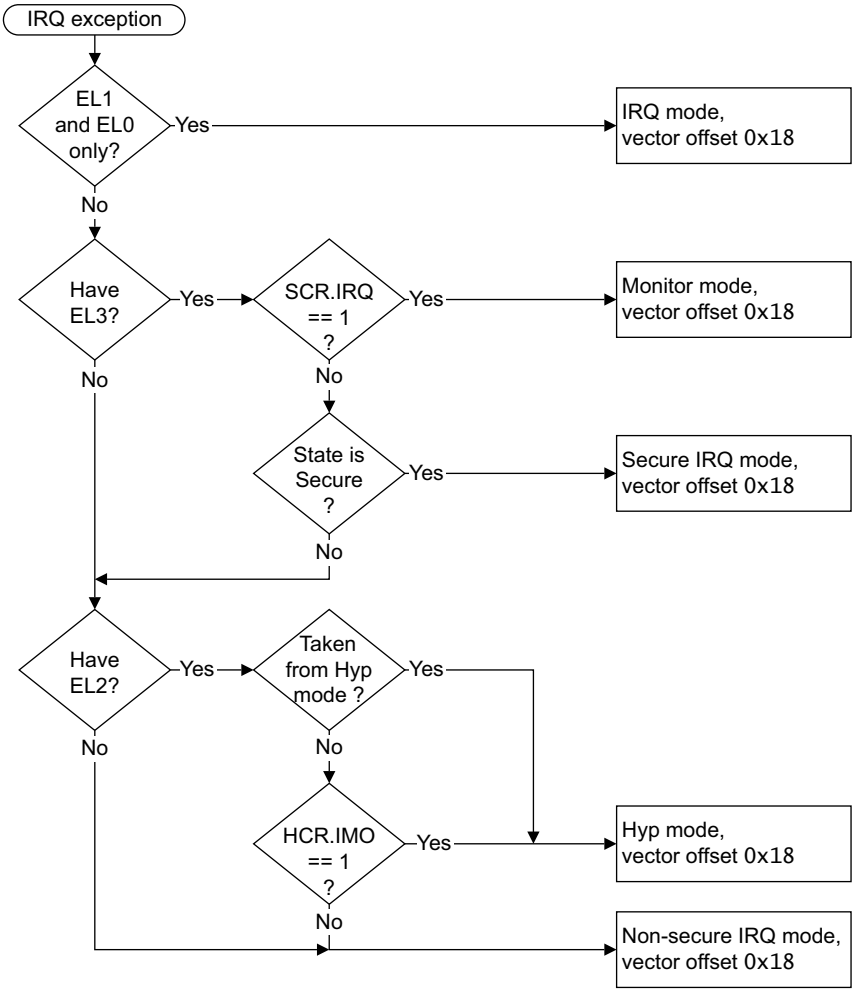



**Figure G1-9 The PE mode the IRQ exception is taken to**

### Pseudocode description of taking the IRQ exception

The `AArch32.TakePhysicalIRQException()` pseudocode procedure describes how the PE takes the exception.

## G1.14.11 Virtual IRQ exception

The Virtual IRQ exception is implemented only as part of EL2.

A Virtual IRQ exception is generated if all of the following apply:

- The PE is in a Non-secure mode other than Hyp mode.
- The value of PSTATE.I is 0.
- Either:
  - EL2 is using AArch32 and the value of HCR.IMO is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.IMO is 1.
- One of the following applies:
  - EL2 is using AArch32 and the value of HCR.VI is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.VI is 1.
  - A Virtual IRQ exception is generated by an IMPLEMENTATION DEFINED mechanism.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The conditions for generating a Virtual IRQ exception mean the exception is always:

- Taken from a Non-secure EL1 or EL0 mode.
- Taken to Non-secure IRQ mode.

For more information see *Virtual exceptions when an implementation includes EL2* on page G1-3801.

The preferred return address for a Virtual IRQ exception is the address of the instruction immediately after the instruction boundary where the exception was taken. This return is performed using the SPSR and LR_irq values generated by the exception entry, using an exception return instruction with a subtraction of 4.

#### The PE mode to which the Virtual IRQ exception is taken

The Virtual IRQ exception is supported only as part of EL2. A Virtual IRQ exception is taken from a Non-secure EL1 or EL0 mode, and is taken to Non-secure IRQ mode, using a vector offset of `0x18` from the Non-secure exception base address.

For more information about this exception see *Virtual exceptions when an implementation includes EL2* on page G1-3801.

#### Pseudocode description of taking the Virtual IRQ exception

The `AArch32.TakeVirtualIRQException()` pseudocode procedure describes how the PE takes the exception.

### G1.14.12 FIQ exception

The FIQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an FIQ interrupt request input to the PE.

When an FIQ exception is taken, exception entry is precise to an instruction boundary.

As described in *Asynchronous exception masking controls* on page G1-3804, FIQ exceptions can be masked. When this happens, a generated FIQ exception is not taken until it is not masked.

By default, an FIQ exception is taken to FIQ mode, but an FIQ exception can be taken:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information, see *The PE mode to which the physical FIQ exception is taken* on page G1-3828.

The preferred return address for an FIQ exception is the address of the instruction following the instruction boundary at which the exception was taken. This return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_fiq and LR_fiq if returning from FIQ mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an `ERET` instruction.

For more information, see *Exception return to an Exception level using AArch32* on page G1-3796.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### The PE mode to which the physical FIQ exception is taken

Figure G1-9 on page G1-3826 shows how the implementation, state, and configuration options determine the PE mode to which an FIQ exception is taken.

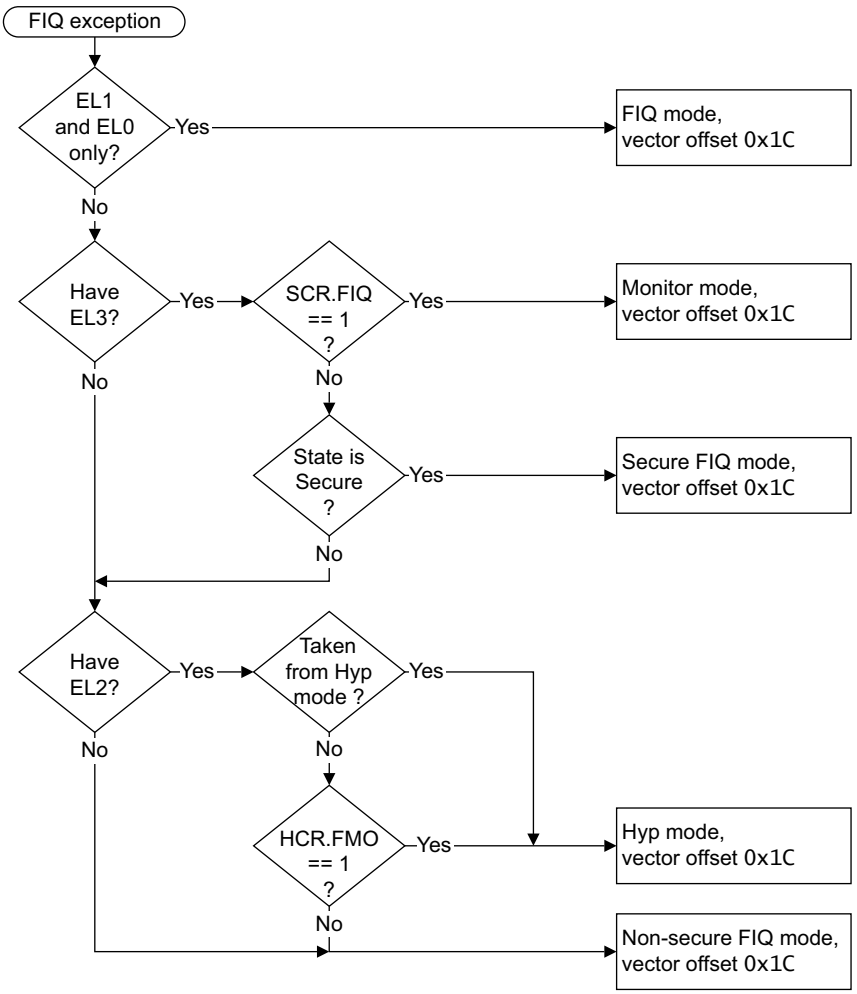



**Figure G1-10 The PE mode the FIQ exception is taken to**

### Pseudocode description of taking the FIQ exception

The `AArch32.TakePhysicalFIQException()` pseudocode procedure describes how the PE takes the exception.

## G1.14.13 Virtual FIQ exception

The Virtual FIQ exception is implemented only as part of EL2.

A Virtual FIQ exception is generated if all of the following apply:

- The PE is in a Non-secure mode other than Hyp mode.
- The value of PSTATE.F is 0.
- Either:
  - EL2 is using AArch32, the value of HCR.FMO is 1, and the value of HCR.VF is 1, or a Virtual FIQ exception is generated by an IMPLEMENTATION DEFINED mechanism.
  - EL2 is using AArch64, the value of HCR_EL2.FMO is 1, and the value of HCR_EL2.VF is 1 or a Virtual FIQ exception is generated by an IMPLEMENTATION DEFINED mechanism.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The conditions for generating a Virtual FIQ exception mean the exception is always:

- Taken from a Non-secure EL1 or EL0 mode.
- Taken to Non-secure FIQ mode.

For more information see *Virtual exceptions when an implementation includes EL2* on page G1-3801.

The preferred return address for a Virtual FIQ exception is the address of the instruction immediately after the instruction boundary where the exception was taken. This return is performed using the SPSR and LR_irq values generated by the exception entry, using an exception return instruction with a subtraction of 4.

### The PE mode to which the Virtual FIQ exception is taken

The Virtual FIQ exception is supported only as part of EL2. A Virtual FIQ exception is taken from a Non-secure EL1 or EL0 mode, and is taken to Non-secure FIQ mode, using a vector offset of `0x1C` from the Non-secure exception base address.

For more information about this exception see *Virtual exceptions when an implementation includes EL2* on page G1-3801.

### Pseudocode description of taking the Virtual FIQ exception

The `AArch32.TakeVirtualFIQException()` pseudocode procedure describes how the PE takes the exception.

## G1.14.14 Additional pseudocode functions for exception handling

The `AArch32.EnterMonitorMode()` pseudocode function changes the PE mode to Monitor mode, with the required state changes.

The `AArch32.EnterHypMode()` pseudocode function changes the PE mode to Hyp mode, with the required state changes.

The `AArch32.EnterMode()` pseudocode function changes the PE mode to a PL1 mode, with the required state changes. It is used for all exceptions that are not routed to Hyp mode or Monitor mode.

The `AArch32.EnterMonitorMode()`, ,`AArch32.EnterHypMode()` and `AArch32.EnterMode()` functions are described in Chapter J1 *ARMv8 Pseudocode*.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G1.15 Reset into AArch32 state

*Reset* on page D1-1533 describes the ARMv8 reset model, including the defined levels of reset. When reset is deasserted, the PE starts executing instructions in the highest implemented Exception level. If that Exception level is using AArch32, then it starts execution:

- In Secure state, if the implementation includes EL3.
- With interrupts disabled:
  - In Hyp mode, if the highest implemented Exception level is EL2.
  - In Supervisor mode, otherwise.

**Note**

- This section describes the architectural requirements for a reset into AArch32 state. It takes no account of whether ARM licenses any particular combination of Exception levels and Execution state. For more information about the licensed combinations see *Support for Exception levels and Execution states* on page D1-1632.
- The Execution state in which the highest implemented Execution level starts executing instructions on coming out of reset might be determined by a configuration input signal.

Reset returns some PE state to architecturally-defined or IMPLEMENTATION DEFINED values, and makes other state UNKNOWN, as described in *PE state on reset into AArch32 state*. For more information about behavior when resetting into an Exception level using AArch32, see:

- *Behavior of caches at reset* on page G3-3955.
- *Enabling stages of address translation* on page G4-4002.
- *TLB behavior at reset* on page G4-4059.
- *Reset and debug* on page H6-4935.

When reset is deasserted, if the PE resets into an Exception level that is using AArch32, it is IMPLEMENTATION DEFINED whether execution starts:

- From an IMPLEMENTATION DEFINED address.
- If reset is into EL3 or EL1, from the low or high reset vector address, as determined by the reset value of the SCTLR.V bit. This reset value can be determined by an IMPLEMENTATION DEFINED configuration input signal.

  **Note**

  This option might be implemented for compatibility with earlier versions of the architecture.

Software might be able to identify the reset address:

- If reset is into EL3, by reading the reset value of MVBAR. That is, after coming out of reset, by reading MVBAR before the boot software has updated it. It is IMPLEMENTATION DEFINED whether this discovery mechanism is supported.
- If reset is into EL2 or EL1, by reading RVBAR. RVBAR can only be implemented at the highest implemented Exception level, and only if that Exception level is not EL3.

If RVBAR is not implemented, and at all Exception levels other than the highest implemented Exception level, the encoding for RVBAR is UNDEFINED.

The ARM architecture does not define any way of returning to a previous Execution state from a reset.

### G1.15.1 PE state on reset into AArch32 state

Immediately after a reset, much of the PE state is UNKNOWN. However, some of the PE state is defined. If the PE resets to AArch32 state using either a Cold or a Warm reset, the PE state that is defined is as follows:

- If reset is into EL3 using AArch32 then all fields of the SCR reset to zero.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

This means SCR.NS correctly indicates that the PE is in Secure state.

- If reset is into EL2 using AArch32 then reset is into Hyp mode and CPSR.M resets to `0b1010`, otherwise reset is into Supervisor mode and CPSR.M resets to `0b0011`,
- CPSR.IL resets to 0.
- The CPACR.{cp11, cp10} fields reset to zero, and if CPACR.ASEDIS is implemented as an RW field it resets to zero.

**Note**

When CPACR.TRCDIS is a RW field, its reset value is architecturally UNKNOWN.

- PSTATE is reset to the values defined by the `TakeReset()` pseudocode function, see *Pseudocode descriptions of reset* on page G1-3832.
- The FPEXC.EN field resets to 0.
- In the SCTLR:
  - The {AFE, TRE, UWXN, WXN, I, SED, ITD, C, A, M} fields reset to 0.
  - The {nTWE, nTWI, CP15BEN} fields reset to 1.
  - The {TE, EE, V} fields reset to IMPLEMENTATION DEFINED values, see the register description for more information.

  When the reset is to EL3 using AArch32 then these reset values apply only to the Secure instance of the SCTLR, and the reset value of the Non-secure SCTLR is architecturally UNKNOWN.
- All field of the TTBCR reset to 0.

  When the reset is to EL3 using AArch32 then:
  - All fields of the Secure TTBCR reset to 0.
  - In the Non-secure TTBCR, the EAE field resets to 0, and the reset values of all other fields are architecturally UNKNOWN.
- The VBAR resets to an IMPLEMENTATION DEFINED value.

  When the reset is to EL3 using AArch32 then this reset value applies only to the Secure instance of the register, and the reset value of the Non-secure VBAR is architecturally UNKNOWN.
- All fields of the DBGDCCINT reset to 0.
- The DBGDSCRext.{ MDBGen, UDCCdis} fields reset to 0.
- The DBGOSDLR.DLK field resets to 0.

In addition:

**If the reset is into EL1 using AArch32**

- In the RMR (at EL1) register, the RR field resets to 0 on any warm or cold reset, and the AA64 field resets to 0 on a cold reset.

**If the reset is into EL2 using AArch32**

- In the HRMR, the RR field resets to 0 on any warm or cold reset, and the AA64 field resets to 0 on a cold reset.
- The HSCTLR.{I, C, M} fields all reset to 0, and the HSCTLR.EE field resets to an IMPLEMENTATION DEFINED value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**If the reset is into EL2 using AArch32 or into EL3 using AArch32**

For a reset into EL3 using AArch32 these reset values apply only if the implementation includes EL2, see the register descriptions for more information.

- All fields of the HCPTR reset to zero.
- All fields of the HCR reset to zero.
- The HCR2.{ID, CD} fields reset to zero.
- All fields of the HSTR reset to zero.
- The VMPIDR resets to the value of the MPIDR, see the register description for more information.
- The VPIDR resets to the value of the MIDR, see the register description for more information.
- The VTTBR.VMID field resets to zero.
- In the HDCR:
  - The HPMN field resets to the IMPLEMENTATION DEFINED value of PMCR.N.
  - The reset value of the HPME field is architecturally UNKNOWN.
  - All other fields reset to 0.

**If the reset is into EL3 using AArch32**

- The MVBAR resets to an IMPLEMENTATION DEFINED value, see the register description for more information.
- If the NSACR.{NSTRCDIS, NSASEDIS} fields are RW fields then they reset to 0.
- In the RMR (at EL3) register, the RR field resets to 0 on any warm or cold reset, and the AA64 field resets to 0 on a cold reset.
- All fields of the SCR reset to zero.
- All fields of the SDER reset to 0.
- All fields of the SDCR reset to zero.

**For either a warm or a cold reset**

- The EDPRSR.SR field resets to 1.
- The EDESR.{SS, RC, OSUC} fields reset to 0.

**For a cold reset only**

- The EDSCR.{RXO, TXU, INTdis, TDA, MA, HDE, ERR, RXfull, TXfull} fields reset to 0.
- The EDECCR.{NSE, SE} fields reset to 0.
- The EDPRSR.{SPMAD, SDAD} fields reset to 0, and the EDPRSR.SPD field resets to 1.
- The DBGOSLSR.OSLK field resets to 1.
- The DBGPRCR.CORENPDRQ field resets to the value of EDPRCR.COREPURQ.
  An External Debug reset sets EDPRCR.COREPURQ to 0, see *External debug register resets on page H8-4963*. If an External Debug reset and a cold reset coincide, both DBGPRCR.CORENPRDRQ and EDPRCR.COREPURQ are reset to 0.
- The debug CLAIM bits are reset to 0.

  **Note**

  These are the bits that are set to 1 by writing to DBGCLAIMSET.CLAIM, and reset to 0 by writing to DBGCLAIMCLR.CLAIM.

## G1.15.2 Pseudocode descriptions of reset

The `AArch32.TakeReset()` pseudocode procedure describes how the PE behaves when reset is deasserted.

The `AArch32.ResetGeneralRegisters()` pseudocode function resets the general-purpose registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The `AArch32.ResetSIMDFPRegisters()` pseudocode function resets the SIMD and floating-point registers.

The `AArch32.ResetSpecialRegisters()` pseudocode function resets the Special-purpose registers, and the debug System registers DLR and DSPSR, that are used for handling Debug exceptions.

The `AArch32.ResetSystemRegisters()` pseudocode function resets all System registers in the CP14 and CP15 encoding space to their reset state as defined in the register descriptions in Chapter G6 *AArch32 System Register Descriptions*.

**Note**

The `ResetSystemRegisters()` function only resets the System registers. It has no effect on memory-mapped registers.

The `ResetExternalDebugRegisters()` pseudocode function resets all external debug registers to their reset state as defined in the register descriptions in Chapter H9 *External Debug Register Descriptions*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.16 Mechanisms for entering a low-power state

The following sections describe the architectural mechanisms that a PE can use to request entry to a low-power state:

- *Wait For Event and Send Event*.
- *Wait For Interrupt* on page G1-3837.

### G1.16.1 Wait For Event and Send Event

The Wait For Event (WFE) mechanism permits a PE to request entry to a low-power state, and, if the request succeeds, to remain in that state until an event is generated by a Send Event operation, or another WFE wake-up event occurs. Example G1-2 describes how a spinlock implementation might use this mechanism to save energy.

**Example G1-2 Spinlock as an example of using Wait For Event and Send Event**

---

A multiprocessor operating system requires locking mechanisms to protect data structures from being accessed simultaneously by multiple PEs. These mechanisms prevent the data structures becoming inconsistent or corrupted if different PEs try to make conflicting changes. If a lock is busy, because a data structure is being used by one PE, it might not be practical for another PE to do anything except wait for the lock to be released. For example, if a PE is handling an interrupt from a device it might need to add data received from the device to a queue. If another PE is removing data from the queue, it will have locked the memory area that holds the queue. The first PE cannot add the new data until the queue is in a consistent state and the lock has been released. It cannot return from the interrupt handler until the data has been added to the queue, so it must wait.

Typically, a spin-lock mechanism is used in these circumstances:

- A PE requiring access to the protected data attempts to obtain the lock using single-copy atomic synchronization primitives such as the Load-Exclusive and Store-Exclusive operations described in *Synchronization and semaphores* on page E2-2395.
- If the PE obtains the lock it performs its memory operation and releases the lock.
- If the PE cannot obtain the lock, it reads the lock value repeatedly in a tight loop until the lock becomes available. At this point it again attempts to obtain the lock.

A spin-lock mechanism is not ideal for all situations:

- In a low-power system the tight read loop is undesirable because it uses energy to no effect.
- In a multi-threaded implementation the execution of spin-locks by waiting threads can significantly degrade overall performance.

Using the Wait For Event and Send Event mechanism can improve the energy efficiency of a spinlock. In this situation, a PE that fails to obtain a lock can execute a Wait For Event instruction, `WFE`, to request entry to a low-power state. When a PE releases a lock, it must execute a Send Event instruction, `SEV`, causing any waiting PEs to wake up. Then, these PEs can attempt to gain the lock again.

---

The execution of a `WFE` instruction can cause suspension of execution only if all of the following are true:

- The instruction does not cause any other exception.
- When the instruction is executed:
  - The Event Register is not set.
  - There is not a pending WFE wakeup event.

For more information about the trap to EL2 see *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-3865.

The architecture does not define the exact nature of the low power state entered as a result of executing a `WFE` instruction, but the execution of a `WFE` instruction must not cause a loss of memory coherency.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Although a complex operating system can contain thousands of distinct locks, the event sent by this mechanism does not indicate which lock has been released. If the event relates to a different lock, or if another PE acquires the lock more quickly, the PE fails to acquire the lock and can reenter the low-power state waiting for the next event.

The Wait For Event system relies on hardware and software working together to achieve energy saving:

- The hardware provides the mechanism to enter the Wait For Event low-power state.
- The operating system software is responsible for issuing:
  - A Wait For Event instruction, to request entry to the low-power state, used in the example when waiting for a spin-lock.
  - A Send Event instruction, required in the example when releasing a spin-lock.

The mechanism depends on the interaction of:

- WFE wake-up events, see *WFE wake-up events* on page G1-3836.
- The Event Register, see *The Event Register*.
- The Send Event instructions, see *The Send Event instructions* on page G1-3836.
- The Wait For Event instruction, see *The Wait For Event instruction*.

## The Event Register

The Event Register is a single bit register for each PE. When set, an event register indicates that an event has occurred, since the register was last cleared, that might require some action by the PE. Therefore, the PE must not suspend operation on issuing a `WFE` instruction.

The reset value of the Event Register is UNKNOWN.

The Event Register for a PE is set by:

- The execution of an `SEV` instruction on any PE in the multiprocessor system.
- The execution of an `SEVL` instruction by the PE.
- An exception return.
- An event from a Generic Timer event stream, see *Event streams* on page G5-4205.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

As shown in this list, the Event Register might be set by IMPLEMENTATION DEFINED mechanisms.

The Event Register is cleared only by a Wait For Event instruction.

Software cannot read or write the value of the Event Register directly.

## The Wait For Event instruction

The action of the Wait For Event instruction depends on the state of the Event Register:

- If the Event Register is set, the instruction clears the register and completes immediately. Normally, if this happens the software makes another attempt to claim the lock.
- If the Event Register is clear the PE can suspend execution, and hardware might enter a low-power state. The PE can remain suspended until a WFE wake-up event or a reset occurs. When a WFE wake-up event occurs, or earlier if the implementation chooses, the `WFE` instruction completes.

The execution in AArch32 state of a `WFE` instruction that would otherwise cause suspension of execution might be trapped, see:

- *Traps to Undefined mode of EL0 execution of WFE and WFI instructions* on page G1-3849.
- *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-3865.
- *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode* on page G1-3875.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Wait For Event instruction, WFE, is available at all privilege levels, see *WFE* on page F6-3196.

Software using the Wait For Event mechanism must tolerate spurious wake-up events, including multiple wake ups.

### WFE wake-up events

The following events are *WFE wake-up events*:

- The execution of an SEV instruction on any PE in the system.
- The execution of an SEVL instruction on the PE.
- A physical IRQ interrupt, unless masked by the PSTATE.I bit.
- A physical FIQ interrupt, unless masked by the PSTATE.F bit.
- A physical asynchronous abort, unless masked by the PSTATE.A bit.
- In Non-secure state in any mode other than Hyp mode:
  - When HCR.IMO is set to 1, a virtual IRQ interrupt, unless masked by the PSTATE.I bit.
  - When HCR.FMO is set to 1, a virtual FIQ interrupt, unless masked by the PSTATE.F bit.
  - When HCR.AMO is set to 1, a virtual asynchronous abort, unless masked by the PSTATE.A bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed* see *Halting allowed and halting prohibited* on page H2-4821.

  See also *External Debug Request debug event* on page H3-4878.
- An event sent by the timer event stream, see *Event streams* on page D6-1901.
- An event sent by some IMPLEMENTATION DEFINED mechanism.
- An event caused by the clearing of the global monitor associated with the PE.

In addition to the possible masking of WFE wake-up events shown in this list, when invasive debug is enabled and EDSCR.HDE is set to 1, EDSCR.INTdis can mask interrupts, including masking them acting as WFE wake-up events. For more information, see *EDSCR, External Debug Status and Control Register* on page H9-5043.

As shown in the list of wake-up events, an implementation can include IMPLEMENTATION DEFINED hardware mechanisms to generate wake-up events.

**Note**

For more information about PSTATE masking see *Asynchronous exception masking controls* on page G1-3804. If the configuration of the masking controls provided by EL2 and EL3 mean that a PSTATE mask bit cannot mask the corresponding exception, then the physical exception is a WFE wake-up event, regardless of the value of the PSTATE mask bit.

### The Send Event instructions

The Send Event instructions are:

**SEV, Send Event** This causes an event to be signaled to all PEs in the multiprocessor system.

**SEVL, Send Event Local**
This must set the local Event Register. It might signal an event to other PEs, but is not required to do so.

The mechanism that signals an event to other PEs is IMPLEMENTATION DEFINED. The PE is not required to guarantee the ordering of this event with respect to the completion of memory accesses by instructions before the SEV instruction. Therefore, ARM recommends that software includes a DSB instruction before any SEV instruction.

**Note**

A DSB instruction ensures that no instruction, including any SEV instruction, that appears in program order after the DSB instruction, can execute until the DSB instruction has completed. For more information, see *Data Synchronization Barrier (DSB)* on page E2-2377.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The SEVL instruction appears to execute in program order relative to any subsequent WFE instruction executed on the same PE, without the need for any explicit insertion of barrier instructions.

Execution of the Send Event instruction sets the Event Register.

The Send Event instructions are available at all privilege levels.

### Pseudocode description of the Wait For Event mechanism

This section defines pseudocode functions that describe the operation of the Wait For Event mechanism.

The ClearEventRegister() pseudocode procedure clears the Event Register of the current PE.

The EventRegistered() pseudocode function returns TRUE if the Event Register of the current PE is set and FALSE if it is clear.

The WaitForEvent() pseudocode procedure optionally suspends execution until a WFE wake-up event or reset occurs, or until some earlier time if the implementation chooses. It is IMPLEMENTATION DEFINED whether restarting execution after the period of suspension causes a ClearEventRegister() to occur.

The SendEvent() pseudocode procedure sets the Event Register of every PE in the system.

## G1.16.2 Wait For Interrupt

AArch32 state supports Wait For Interrupt through an instruction, WFI, that is provided in the A32 and T32 instruction sets. For more information, see *WFI* on page F6-3198.

When a PE issues a WFI instruction, its execution can be suspended, and a low-power state can be entered.

The execution in AArch32 state of a WFI instruction that would otherwise cause suspension of execution might be trapped, see:

- *Traps to Undefined mode of EL0 execution of WFE and WFI instructions* on page G1-3849.
- *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-3865.
- *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode* on page G1-3875.

The execution of a WFI instruction can cause suspension of execution only if both:

- The instruction does not cause any other exception.
- When the instruction is executed, there is not a pending WFI wakeup event.

### WFI wake-up events

The PE can remain suspended in its WFI state until it is reset, or one of the following *WFI wake-up events* occurs:

- A physical IRQ interrupt, regardless of the value of the PSTATE.I bit.
- A physical FIQ interrupt, regardless of the value of the PSTATE.F bit.
- A physical asynchronous abort, regardless of the value of the PSTATE.A bit.
- In Non-secure state in any mode other than Hyp mode:
  - When HCR.IMO is set to 1, a virtual IRQ interrupt, regardless of the value of the PSTATE.I bit.
  - When HCR.FMO is set to 1, a virtual FIQ interrupt, regardless of the value of the PSTATE.F bit.
  - When HCR.AMO is set to 1, a virtual asynchronous abort, regardless of the value of the PSTATE.A bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed* see *Halting allowed and halting prohibited* on page H2-4821.
  See also *External Debug Request debug event* on page H3-4878.

An implementation can include other IMPLEMENTATION DEFINED hardware mechanisms to generate WFI wake-up events.

When a WFI wake-up event is detected, or earlier if the implementation chooses, the WFI instruction completes.

WFI wake-up events cannot be masked by the mask bits in the PSTATE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The architecture does not define the exact nature of the low power state, but the execution of a WFI instruction must not cause a loss of memory coherency.

**Note**

- Because debug events are WFI wake-up events, ARM strongly recommends that is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures the intervention of debug while waiting does not significantly change the function of the program being debugged.
- In some previous implementations of Wait For Interrupt, the idle loop is followed by exit functions that must be executed before taking the interrupt. The operation of Wait For Interrupt remains consistent with this model, and therefore differs from the operation of Wait For Event.
- Some implementations of Wait For Interrupt drain down any pending memory activity before suspending execution. The ARM architecture does not require this operation, and software must not rely on Wait For Interrupt operating in this way.

### Using WFI to indicate an idle state on bus interfaces

A common implementation practice is to complete any entry into powerdown routines with a WFI instruction. Typically, the WFI instruction:

1. Forces the completion of execution of any instructions that are in progress, and of all associated bus activity.
2. Suspends the execution of instructions by the PE.

The control logic required to do this tracks the activity of the bus interfaces used by the PE. This means it can signal to an external power controller when there is no ongoing bus activity.

However, memory-mapped and external debug interface accesses to debug registers must continue to be processed while the PE is in the WFI state. The indication of idle state to the system normally only applies to the non-debug functional interfaces used by the PE, not the debug interfaces.

When the value of DBGOSDLR.DLK, the OS Double Lock status bit, is set to 1, this idle state must not be signaled to the PE unless the system can guarantee, also, that the debug interface is idle.

**Note**

When separate core and debug power domains are implemented, the debug interface referred to in this section is the interface between the core and debug power domains, since the signal to the power controller indicates that the core power domain is idle. For more information about the power domains see *Power domains and debug* on page H6-4925.

The exact nature of this interface is IMPLEMENTATION DEFINED, but the use of Wait For Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it very suitable as the preferred powerdown entry mechanism.

### Pseudocode description of Wait For Interrupt

The WaitForInterrupt() pseudocode function optionally suspends execution until a WFI wake-up event or reset occurs, or until some earlier time if the implementation chooses.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.17 The conceptual coprocessor interface and system control

AArch32 state includes a coprocessor interface that can access sixteen coprocessors, CP0 to CP15. *Conceptual coprocessor support* on page E1-2353 introduces this interface. Part of this interface is reserved for conceptual coprocessors, as follows:

- CP10 and CP11 provide Advanced SIMD and floating-point functionality.
- CP14 and CP15 provide configuration and control related to the architecture:

In ARMv8, AArch32 has no support for coprocessors other than CP10, CP11, CP14, and CP15.

This section gives:

- An introduction to the CP14 and CP15 registers, see *System registers in the CP14 and CP15 encoding space*.
- Information about access controls for the other coprocessors, see *Access controls on CP10 and CP11*.

### G1.17.1 System registers in the CP14 and CP15 encoding space

In AArch32 state:

- The CP14 encoding space is reserved for the configuration and control of:
  - Debug features, see *Debug registers* on page G6-4648.
  - Trace features, see the *Embedded Trace Macrocell Architecture Specification*.
  - Identification registers for the Trivial Jazelle implementation, see *Trivial implementation of the Jazelle extension* on page G1-3772.
- The CP15 encoding space is reserved for the control and configuration of the PE, including architecture and feature identification. This means the CP15 encoding space provides access to the System registers that control and return status information for PE operation.

  **Note**

  This encoding space was previously called the System Control coprocessor.

See Chapter G6 *AArch32 System Register Descriptions*.

#### Access to CP14 and CP15 registers

Most CP14 and CP15 registers are accessible only from EL1 or higher. For possible accesses from EL0:

- The register descriptions in Chapter G6 *AArch32 System Register Descriptions* indicate whether a register is accessible from EL0.
- The descriptions of the CP14 interface in *Debug registers* on page G6-4648 include the permitted accesses to the debug registers from EL0.
- *PL0 views of the CP15 registers* on page G4-4158 summarizes the permitted accesses to CP15 registers from EL0.

### G1.17.2 Access controls on CP10 and CP11

The CP10 and CP11 part of the coprocessor interface supports the Advanced SIMD and floating-point instructions, providing access to System registers relating to the use of these instructions, and some instruction encodings. See also *Advanced SIMD and floating-point support* on page G1-3841.

In ARMv8, the CPACR controls access to CP10 and CP11 from software executing at EL1 or EL0 in AArch32 state.

Initially on powerup or reset into AArch32 state, access to coprocessors CP10 and CP11 is disabled.

**Note**

The CPACR has no effect on accesses from Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If an implementation includes EL3, the NSACR determines whether CP10 and CP11 can be accessed from the Non-secure state.

If an implementation includes EL2, the HCPTR provides additional controls on Non-secure accesses to CP10 and CP11. For accesses that are otherwise permitted by the CPACR and NSACR settings, setting HCPTR bits to 1:

- Traps otherwise-permitted accesses from EL1 or EL0 to EL2. When EL2 is using AArch32, these accesses are trapped to Hyp mode.
- Makes accesses from EL2 mode UNDEFINED. When EL2 is using AArch32, this makes accesses from Hyp mode UNDEFINED.

For more information, see *General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers* on page G1-3865.

**Note**

The access settings for CP10 and CP11 must be identical. In ARMv8, if these settings are not identical the behavior of the Advanced SIMD and floating-point functionality is CONSTRAINED UNPREDICTABLE, so that, for all purposes other than reading the value of the CP11 control field, behavior is as if the access control field for CP11 has the same value as the access control field for CP10. For more information see *Handling of conceptual coprocessors CP10 and CP11* on page K1-5428.

### G1.17.3 Pseudocode description of checking accesses to the conceptual coprocessors CP14 and CP15

The `Coproc_CheckInstr()` function determines whether a coprocessor instruction that targets the conceptual CP14 or CP15 coprocessor is accepted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.18 Advanced SIMD and floating-point support

*Advanced SIMD and floating-point instructions* on page E1-2325 introduces:

- The scalar floating-point instructions in the A32 and T32 instruction sets.
- The Advanced SIMD integer and floating-point vector instructions in the A32 and T32 instruction sets.
- The SIMD and floating-point register file, that can be viewed as:
  - Singleword registers S0 - S31.
  - Doubleword registers D0 - D31.
  - Quadword registers Q0 - Q15.
- The *Floating-Point Status and Control Register* (FPSCR).

For more information about the System registers for the Advanced SIMD and floating-point operation see *Advanced SIMD and floating-point System registers* on page G1-3843. Software can interrogate these registers to discover the implemented Advanced SIMD and floating-point support.

*AArch32 implications of not including support for Advanced SIMD and floating-point* summarizes the effects of not supporting these instructions, and the following subsections give more information about the Advanced SIMD and Floating-point support:

- *Enabling Advanced SIMD and floating-point support*.
- *Advanced SIMD and floating-point System registers* on page G1-3843.
- *Context switching when using Advanced SIMD and floating-point functionality* on page G1-3844.
- *Floating-point exception traps* on page G1-3844.

### G1.18.1 AArch32 implications of not including support for Advanced SIMD and floating-point

As stated in *Implementations not including Advanced SIMD and floating-point instructions* on page D1-1633, although ARMv8-A generally requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets, for implementations targeting specialized markets, ARM might produce or license ARMv8-A implementations that do not provide any support for Advanced SIMD and floating-point instructions. In such an implementation, in AArch32 state:

- The CPACR.{ASEDIS, cp11, cp10} fields are RES0.
- The NSACR.{NSASEDIS, cp11, cp10} fields are RES0.
- The HCPTR.{TASE, TCP11, TCP10} fields are RES1.
- The FPEXC, FPSCR, and FPSID registers are not implemented.
- Attempted accesses to floating-point and Advanced SIMD functionality are UNDEFINED. This means:
  - All floating-point and Advanced SIMD instructions are UNDEFINED.
  - Attempts to access the floating-point and Advanced SIMD System registers are UNDEFINED.

### G1.18.2 Enabling Advanced SIMD and floating-point support

Software must ensure that the required access to the Advanced SIMD and floating-point features is enabled. Most of those controls are described in *Configurable instruction enables and disables, and trap controls* on page G1-3846, and this section:

- Summarizes those controls.
- Provides additional information in the following subsections:
  - *FPEXC control of access to Advanced SIMD and floating-point functionality* on page G1-3843.
  - *EL0 access to Advanced SIMD and floating-point functionality* on page G1-3843.

**Note**

This section shows the controls when the controlling Exception levels are using AArch32. Similar controls are provided when the Exception levels are using AArch64, and then apply to lower Exception levels that are using AArch32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The controls of access to floating-point and Advanced SIMD functionality are:

**General {CP10, CP11} controls**

**Note**

The naming of these controls reflects the conceptual coprocessors CP10 and CP11, see *The conceptual coprocessor interface and system control* on page G1-3839.

The {CP10, CP11} controls provide general control of the use of floating-point and Advanced SIMD functionality, as follows:

- CPACR.{cp10, cp11} control access from PE modes other than Hyp mode.

  These fields have no effect on accesses to CP10 and CP11 from Hyp mode.
- In an implementation that includes EL3, NSACR.{cp10, cp11} control access from Non-secure state.
- In an implementation that includes EL2, if NSACR.{cp10, cp11} permit Non-secure accesses, or if EL3 is not implemented, HCPTR.{TCP10, TCP11} provide an additional control on those accesses.

In each case, the {CP10, CP11} controls must be programmed to the same value, otherwise operation is CONSTRAINED UNPREDICTABLE.

The ARMv8 CONSTRAINED UNPREDICTABLE behavior is that, for all purposes other than reading the value of the register field, behavior is as if the access control field for CP11 has the same value as the access control field for CP10. For more information see *Handling of conceptual coprocessors CP10 and CP11* on page K1-5428.

For more information about these controls, see:

- *Enabling PL0 and PL1 accesses to the SIMD and floating-point registers* on page G1-3851.
- *General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers* on page G1-3865.
- *Enabling Non-secure access to SIMD and floating-point functionality* on page G1-3877.

**Control of accesses to the CPACR from Non-secure PL1 modes**

As stated in *General {CP10, CP11} controls*, the CPACR controls access to floating-point and Advanced SIMD functionality from PE modes other than Hyp mode. Accesses to the CPACR from Non-secure PL1 modes can be trapped to EL2, see *Traps to Hyp mode of Non-secure EL1 accesses to the CPACR* on page G1-3867.

**Additional controls of Advanced SIMD functionality**

- If implemented as an RW field, CPACR.ASEDIS can make all Advanced SIMD instructions UNDEFINED in all modes other than Hyp mode.
- In an implementation that includes EL3, when CPACR.ASEDIS permits use of the Advanced SIMD instructions or if the CPACR.ASEDIS control is not implemented, NSACR.NSASEDIS can make all Advanced SIMD instructions UNDEFINED in Non-secure state.
- In an implementation that includes EL2, when the CPACR and NSACR settings permit Non-secure use of the Advanced SIMD instructions, if HCPTR.TASE is implemented as an RW field it can make these instructions UNDEFINED in Hyp mode, and trap to Hyp mode any use of these instructions in a Non-secure PL0 or PL1 mode.

For more information about these controls, see:

- *Disabling PL0 and PL1 execution of Advanced SIMD instructions* on page G1-3852.
- *Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality* on page G1-3866.
- *Disabling Non-secure access to Advanced SIMD functionality* on page G1-3878.

*Pseudocode description of enabling SIMD and floating-point functionality* on page G1-3879 all of these controls.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### FPEXC control of access to Advanced SIMD and floating-point functionality

In addition, FPEXC.EN is an enable bit for most Advanced SIMD and floating-point operations. When FPEXC.EN is 0, all Advanced SIMD and floating-point instructions are treated as UNDEFINED except for:

- A VMSR to the FPEXC or FPSID register.
- A VMRS from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2 register.

These instructions can be executed only at EL1 or higher.

**Note**

- When the FPSID is accessible, any write access to the FPSID is ignored.
- When FPEXC.EN is 0, these operations are treated as UNDEFINED:
  - A VMSR to the FPSCR.
  - A VMRS from the FPSCR.

### EL0 access to Advanced SIMD and floating-point functionality

When the access controls summarized in this section permit EL0 access to the Advanced SIMD and floating-point functionality, this applies only to the subset of functionality that is available at EL0. In particular:

- Only Advanced SIMD and Floating-point System register that is accessible is the FPSCR.
- The Advanced SIMD and floating-point instructions are available.

Execution at EL0 corresponds to the application level view of the Advanced SIMD and floating-point functionality, as described in *Advanced SIMD and floating-point System registers* on page E1-2328.

## G1.18.3 Advanced SIMD and floating-point System registers

AArch32 state provides a common set of System registers for the Advanced SIMD and floating-point functionality. This section gives general information about this set of registers, and indicates where each register is described in detail. It contains the following subsections:

- *Register map of the Advanced SIMD and floating-point System registers*.
- *Accessing the Advanced SIMD and floating-point System registers*.

### Register map of the Advanced SIMD and floating-point System registers

Table G4-70 on page G4-4180 shows the register map of the Advanced SIMD and Floating-point registers. Each register is 32 bits wide. In an implementation that includes EL3, the Advanced SIMD and Floating-point registers are common registers, see *Common System registers* on page G4-4128.

### Accessing the Advanced SIMD and floating-point System registers

Software accesses the Advanced SIMD and floating-point System registers using the VMRS and VMSR instructions, see:

- *VMRS* on page F7-3495.
- *VMSR* on page F7-3497.

For example:

```
VMRS <Rt>, FPSID    ; Read  Floating-Point System ID Register
VMRS <Rt>, MVFR1    ; Read  Media and VFP Feature Register 1
VMSR FPSCR, <Rt>    ; Write Floating-Point System Control Register
```

Software can access the Advanced SIMD and floating-point System registers only if the access controls permit the access, see *Enabling Advanced SIMD and floating-point support* on page G1-3841.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

All hardware ID information can be accessed only from EL1 or higher. This means:

**The FPSID is accessible only from EL1 or higher.**

This is a change introduced from VFPv3. Previously, the FPSID register can be accessed in all modes.

**The MVFR registers are accessible only from EL1 or higher.**

Unprivileged software must issue a system call to determine what features are supported.

### G1.18.4 Context switching when using Advanced SIMD and floating-point functionality

When the Advanced SIMD and floating-point functionality is used by only a subset of processes, the operating system might implement lazy context switching of the Advanced SIMD and floating-point register file and System registers.

In the simplest lazy context switch implementation, the primary context switch software disables the Advanced SIMD and floating-point functionality, by disabling access to coprocessors CP10 and CP11 in the CPACR, see *Enabling Advanced SIMD and floating-point support* on page G1-3841. Subsequently, when a process or thread attempts to use an Advanced SIMD or Floating-point instruction, it triggers an Undefined Instruction exception. The operating system responds by saving and restoring the Advanced SIMD and floating-point register file and System registers. Typically, it then re-executes the Advanced SIMD or floating-point instruction that generated the Undefined Instruction exception.

### G1.18.5 Floating-point exception traps

Execution of a floating-point instruction can generate an exceptional condition, called a *floating-point exception*.

The ARMv8-A architecture supports synchronous exception generation in the event of any or all of the following floating-point floating-point exceptions:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

**Note**

Do not confuse floating-point exceptions with the AArch32 architectural exceptions summarized in *AArch32 state exception descriptions* on page G1-3811.

Whether an implementation includes synchronous exception generation for these floating-point exceptions is IMPLEMENTATION DEFINED:

- For an implementation that does provide this capability, FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} are the control bits that enable synchronous exception generation for each of the floating-point exceptions.
- For an implementation that does provide this capability, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} are RAZ/WI.

**Note**

The ARMv8-A architecture does not support asynchronous reporting of floating-point exceptions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When generating synchronous exceptions for one or more floating-point exceptions is enabled, the synchronous exceptions generated by the floating-point exception traps are taken to the lowest Exception level that can handle such an exception, while adhering to the rule that an exception can never be taken to a lower Exception level. This means that trapped floating-point exceptions taken:

- From EL0 are taken to EL1, except for the following cases when they are taken from Non-secure EL0 to EL2:
  - EL2 is using AArch32 and the value of HCR.TGE is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.TGE is 1
- From EL1 are taken to EL1.
- From EL2 are taken to EL2.
- From EL3 are taken to EL3.

If the exception is taken to an Exception level that is using AArch64 then it is reported in the ELR_ELx for the Exception level to which it is taken, as described in *Exception entry* on page D1-1537.

If the exception is taken to an Exception level that is using AArch32 then it is taken as an Undefined Instruction exception, see *Undefined Instruction exception* on page G1-3811. The FPEXC identifies the floating-point exceptions that occurred since the corresponding status bits in that register were last set to 0.

See also *Floating-point exceptions* on page E1-2329.

In an implementation that provides synchronous exception generation for floating-point exceptions:

- Synchronous exception generation applies to floating-point exceptions generated by scalar SIMD and floating-point instructions executed in AArch32 state.
- The registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception. An implementation is permitted not to restore the cumulative exception flags in the event of such an exception.

ARMv8 does not support the trapping of floating-point exceptions from Advanced SIMD instructions executed in AArch32 state,

The `AArch32.FPTrappedException()` and `FPProcessException()` pseudocode functions describe the handling of trapped floating-point exceptions generated in AArch32 state.

The `AArch32.FPTrappedException()` and `FPProcessException()` functions are described in Chapter J1 *ARMv8 Pseudocode*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G1.19 Configurable instruction enables and disables, and trap controls

This section describes the controls provided by AArch32 state for enabling, disabling, and trapping particular instructions. Each control is categorized as an *instruction enable*, an *instruction disable*, or a *trap control*:

**Instruction enables and instruction disables**

Enable or disable the use of one or more particular instructions at a particular Privilege level and Security state.

When an instruction is disabled as a result of an instruction enable or disable, it is UNDEFINED.

The exception generated by attempting to executed an UNDEFINED instruction is:

- Taken to EL1 if the UNDEFINED instruction was executed at EL0, unless the instruction was executed at Non-secure EL0 and is routed to EL2 by the control described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.

  When the exception is taken to EL1 it is taken to Undefined mode.
- Otherwise, taken to the Exception level at which the UNDEFINED instruction was executed:
  - If the instruction was executed in Hyp mode the exception is taken to Hyp mode.
  - Otherwise, the exception is taken to Undefined mode.

**Trap controls**

Control whether one or more instructions, when executed at a particular Privilege level, are *trapped*.

**Note**

AArch32 trap controls are described in terms of Privilege levels, rather than Exception levels, because the PL1 traps apply at and are controlled from:

**EL1** In Non-secure state, and in Secure state when EL3 is using AArch64.

**EL3** In Secure state when EL3 is using AArch32.

For more information see *Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990.

Trap controls are grouped as:

**PL1, excluding Monitor mode**

Trapped instructions generate Undefined Instruction exceptions that are taken to Undefined mode, unless the instruction was executed at Non-secure EL0 and is routed to EL2 by the control described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.

For more information about these traps see *PL1 configurable controls* on page G1-3847.

**Hyp mode (PL2)**

These traps apply only to execution in Non-secure state. This section only describes the traps that apply when EL2 is using AArch32.

Trapped instructions generate:

- Hyp Trap exceptions, taken to Hyp mode, if trapped from a mode other than Hyp mode.
- Undefined Instruction exceptions taken to Hyp mode, if trapped from Hyp mode.

For more information about these traps see *EL2 configurable controls* on page G1-3855.

See also *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.

**Monitor mode (Secure PL1)**

This section only describes the traps that apply when EL3 is using AArch32.

Trapped instructions generate Monitor Trap exceptions, that are taken to Monitor mode.

For more information about these traps see *EL3 configurable controls* on page G1-3874.

An exception generated as a result of an instruction enable or disable, or a trap control, is only taken if the instruction does not also generate a higher priority exception. *Exception prioritization for exceptions taken to AArch32 state* on page G1-3778 defines the prioritization of different exceptions on the same instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Exceptions generated as a result of these controls are synchronous exceptions.

For exceptions taken to an Exception level that is using AArch32, only exceptions that are taken to Hyp mode are reported in a syndrome register, the HSR.

**Note**

- A particular control might have a mnemonic that suggests it is different type of control to the control type it is categorized as. For example, CPACR.TRCDIS is a trap control even though TRCDIS is a mnemonic for Trace Disable.
- An implementation might provide additional controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.
- *Configurable instruction enables and disables, and trap controls* on page D1-1575 describes controls provided by AArch64 state for enabling, disabling, and trapping instructions. Generally, where an AArch64 control applies to execution at lower Exception levels, it traps the equivalent functionality when that lower Exception level is using AArch32. See the AArch64 trap controls for more information.

This section is organized as follows:

- *Register access instructions*.
- *PL1 configurable controls*.
- *EL2 configurable controls* on page G1-3855.
- *EL3 configurable controls* on page G1-3874.
- *Pseudocode description of configurable instruction enables, disables, and traps* on page G1-3879.

### G1.19.1 Register access instructions

When an instruction is disabled or trapped, the exception is taken before execution of the instruction. This means that if the instruction is a register access instruction:

- No access is made before the exception is taken.
- Side-effects that are normally associated with the access do not occur before the exception is taken.

### G1.19.2 PL1 configurable controls

In AArch32 state, each control is associated with a particular System register field that is accessible:

- When EL3 is using AArch64, or when an implementation does not include EL3, from EL1.
- When EL3 is using AArch32:
  - In Non-secure state, from EL1.
  - In Secure state, from EL3.

This means that the controls are described as PL1 controls, because PL1 is defined as being the Privilege level of software that is executing:

- At EL3, if the PE is executing in EL3 and EL3 is using AArch32.
- At EL1 under all other conditions.

Where there is an AArch64 control that is equivalent to an AArch32 PL1 control, the AArch64 control is an EL1 control.

Any exception that is generated because of an AArch32 PL1 control is taken to a PL1 mode.

**Note**

Any exception generated because of an AArch32 PL1 control is taken to AArch32 state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-24 shows the AArch32 System registers that contain these controls.

**Table G1-24 System registers that contain instruction enables and disables, and trap controls**

| Register name | Register description |
|---|---|
| SCTLR | System Control Register |
| FPEXC | Floating-point Exception Control Register |
| CPACR | Architectural Feature Access Control Register |
| DBGDSCRext | Monitor System Debug Control Register |
| PMUSERENR | Performance Monitors User Enable Register |

Table G1-25 summarizes these controls.

**Table G1-25 Instruction enables and disables, and trap controls, for exceptions taken to Undefined mode**

| Control | Control type[a] | Description |
|---|---|---|
| SCTLR.{nTWE, nTWI} | T | *Traps to Undefined mode of EL0 execution of WFE and WFI instructions* on page G1-3849 |
| SCTLR.{SED, ITD}<br>SCTLR.CP15BEN | D<br>E | *Disabling or enabling PL0 and PL1 use of AArch32 deprecated functionality* on page G1-3849 |
| CPACR.TRCDIS | T | *Traps to Undefined mode of PL0 and PL1 System register accesses to trace registers* on page G1-3850 |
| CPACR.{cp11, cp10}<br>FPEXC.EN<br>CPACR.ASEDIS | E<br>E<br>D | *Enabling use of Advanced SIMD and floating-point functionality* on page G1-3851 |
| DBGDSCRext.UDCCdis | T | *Traps to Undefined mode of EL0 accesses to the Debug Communications Channel (DCC) registers* on page G1-3852 |
| CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN} | T | *Traps to Undefined mode of EL0 accesses to the Generic Timer registers* on page G1-3853 |
| PMUSERENR.{ER, CR, SW, EN} | T | *Traps to Undefined mode of EL0 accesses to Performance Monitors registers* on page G1-3853 |

a. See Table G1-26.

**Table G1-26 Control types, for exceptions taken to Undefined mode**

| Abbreviation | Type | See |
|---|---|---|
| D | Disable | *Instruction enables and instruction disables* on page G1-3846 |
| E | Enable | *Instruction enables and instruction disables* on page G1-3846 |
| T | Trap | *Trap controls* on page G1-3846 |

When generated in Non-secure User mode, exceptions generated by these controls can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Instructions that fail their condition code check

See *Conditional execution of undefined instructions* on page G1-3813.

### Instructions that are UNPREDICTABLE

For an instruction that is UNPREDICTABLE, the behavior of the instruction when the instruction is disabled or trapped is UNPREDICTABLE. The architecture permits such an instruction to generate an Undefined Instruction exception, but does not require it to do so.

### Traps to Undefined mode of EL0 execution of `WFE` and `WFI` instructions

SCTLR.{nTWE, nTWI} trap EL0 execution of `WFE` and `WFI` instructions to Undefined mode:

**SCTLR.nTWE**

| | |
|---|---|
| **1** | This control has no effect on the EL0 execution of `WFE` instructions. |
| **0** | Any attempt to execute a `WFE` instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state. |

**SCTLR.nTWI**

| | |
|---|---|
| **1** | This control has no effect on the EL0 execution of `WFI` instructions. |
| **0** | Any attempt to execute a `WFI` instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state. |

The attempted execution of a conditional `WFE` or `WFI` instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a `WFE` or `WFI` can complete at any time, even without a Wakeup event, the traps on `WFE` of `WFI` are not guaranteed to be taken, even if the `WFE` or `WFI` is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When generated in Non-secure User mode, exceptions generated by these controls can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait For Event and Send Event* on page G1-3834.
- *Wait For Interrupt* on page G1-3837.

### Disabling or enabling PL0 and PL1 use of AArch32 deprecated functionality

Table G1-27 on page G1-3850 shows the deprecated AArch32 functionality that might have disable controls in the SCTLR:

- The SED control is always implemented.
- Whether each of the ITD or CP15BEN controls is implemented is IMPLEMENTATION DEFINED. If a control is not implemented then the associated functionality cannot be disabled.

When an instruction is disabled by one of these controls, it is UNDEFINED at PL0 and PL1. This means an attempt to execute the instruction at PL0 or PL1 generates an Undefined Instruction exception that is taken to Undefined mode, unless both of the following apply, in which case the attempted execution generates an exception that is taken to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790:

- The instruction is executed at Non-secure EL0 using AArch32.
- Either:
  - EL2 is using AArch32 and the value of HCR.TGE is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G1-27 PL1 controls for disabling and enabling PL0 and PL1 use of AArch32 deprecated functionality**

| Deprecated AArch32 functionality | Instruction enable or disable in the SCTLR | Disabled instructions |
|---|---|---|
| SETEND instructions | SED[a] | SETEND instructions |
| Some uses of IT instructions | ITD[b] | See the SCTLR.ITD description |
| Accesses to the CP15DMB, CP15DSB, and CP15ISB barrier instructions | CP15BEN[c] | MCR accesses to the CP15DMB, CP15DSB, and CP15ISB instructions |

a. SETEND instruction disable. SETEND instructions are disabled when the value of this field is 1.
b. IT instruction disable. If this control is implemented, some uses of IT instructions are disabled when the value of this field is 1.
c. CP15 memory barrier enable. If this control is implemented, the specified register accesses are disabled when the value of CP15BEN is 0.

**Note**

The uses of the IT instruction, and use of the CP15DMB, CP15DSB, and CP15ISB barrier instructions, are deprecated for performance reasons.

### Traps to Undefined mode of PL0 and PL1 System register accesses to trace registers

If implemented, the CPACR.TRCDIS control traps PL0 and PL1 System register accesses to the trace registers to Undefined mode, as follows:

**1** PL0 and PL1 CP14 accesses to the trace registers are trapped to Undefined mode

**0** This control has no effect on PL0 and PL1 CP14 accesses to the trace registers.

If the CPACR.TRCDIS control is not implemented, then the CPACR.TRCDIS field is RAZ/WI. This means the CPACR does not provide a trap to Undefined mode of PL1 and PL0 System register accesses to trace registers. See the register description for more information.

**Note**

- System register accesses to the trace registers use the CP14 encoding space.
- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED.
- The ARMv8-A architecture does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Register access instructions* on page G1-3847.

If EL3 is implemented and is using AArch32, and NSACR.NSTRCDIS is 1, CPACR.TRCDIS behaves as RAO/WI in Non-secure state. This behavior also applies if the CPACR.TRCDIS control is not implemented.

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Enabling use of Advanced SIMD and floating-point functionality

Table G1-28 summarizes the controls of Advanced SIMD and floating-point functionality.

**Table G1-28 Controls of use of Advanced SIMD and floating-point functionality**

| Control | Type | Description, see |
|---|---|---|
| CPACR.{cp11, cp10} | E | *Enabling PL0 and PL1 accesses to the SIMD and floating-point registers* |
| FPEXC.EN | E | *Enabling access to the SIMD and floating-point registers* |
| CPACR.ASEDIS | D | *Disabling PL0 and PL1 execution of Advanced SIMD instructions* on page G1-3852 |

If any of CPACR.{cp11, cp10}, FPEXC.EN, or for Advanced SIMD instructions, CPACR.ASEDIS, disable a floating-point or an Advanced SIMD instruction, the instruction is UNDEFINED. Support for the CPACR.ASEDIS control is optional, and if the control is not implemented behavior is as if the control permits the execution of Advanced SIMD instructions at PL1 and PL0.

When generated in Non-secure User mode, exceptions generated by these controls can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.

#### *Enabling PL0 and PL1 accesses to the SIMD and floating-point registers*

CPACR.{cp11, cp10} enable PL0 and PL1 accesses to the SIMD and floating-point registers.

**Note**

- CPACR.{cp11, cp10} are both two-bit fields. For the definition of when PL0 and PL1 accesses are enabled, see the CPACR register description. When PL0 and PL1 accesses are disabled, they are UNDEFINED.
- Software must set CPACR.cp11 and CPACR.cp10 to the same value.

Table G1-29 shows the registers for which accesses are enabled.

**Table G1-29 Register accesses enabled at PL0 and PL1 by CPACR.{cp11, cp10}**

| Enabled at | Registers |
|---|---|
| PL0 and PL1, or PL0 only[a] | FPSCR, FPEXC, FPSID, MVFR0, MVFR1, MVFR2, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers[b] |

a. Depending on the value of CPACR.{cp11, cp10}. See the register description for details.

b. Permitted `VMSR` accesses to the FPSID are ignored, but for the purposes of the {cp10, cp11} controls the architecture defines a `VMSR` accesses to the FPSID from EL1 or higher is an access to a SIMD and floating-point register.

If EL3 is implemented and is using AArch32, and NSACR.{cp11, cp10} are both set to 0, the functionality described in this section is disabled in Non-secure state, and CPACR.{cp11, cp10} are RAZ/WI in Non-secure state. See *Enabling Non-secure access to SIMD and floating-point functionality* on page G1-3877.

For more information about SIMD and floating-point support, see *Advanced SIMD and floating-point support* on page G1-3841.

#### *Enabling access to the SIMD and floating-point registers*

FPEXC.EN enables accesses to the SIMD and floating-point registers at all Exception levels, but does not control the following:

- `VMSR` accesses to the FPEXC or FPSID.
- `VMRS` accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When FPEXC.EN is:

**1** Accesses to the registers shown in Table G1-30 are enabled at all Exception levels.

**0** All accesses to the registers shown in Table G1-30 are UNDEFINED.

Table G1-30 shows the registers for which accesses are enabled, and for an exception taken to Hyp mode, how the exception is reported in HSR.

**Table G1-30 Register accesses enabled when FPEXC.EN is 1**

| Enabled at | Registers | Syndrome reporting in HSR[a] |
|---|---|---|
| All Exception levels | FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers. | Exception for an unknown reason, using EC value `0x00` |

a. Only for exceptions that are taken to Hyp mode.

For more information, see *Advanced SIMD and floating-point support* on page G1-3841.

#### *Disabling PL0 and PL1 execution of Advanced SIMD instructions*

If implemented as an RW field, CPACR.ASEDIS can disable PL0 and PL1 execution of Advanced SIMD instructions, as follows:

**1** Advanced SIMD instructions are UNDEFINED at PL0 and PL1.

**0** Advanced SIMD instruction execution is enabled at PL0 and PL1.

The instructions that CPACR.ASEDIS disables are those described in Chapter F5 *T32 and A32 Instruction Sets Advanced SIMD and floating-point Encodings*.

When the control is not implemented, meaning the CPACR.ASEDIS field is RAZ/WI, behavior is as if the control permits execution of Advanced SIMD instructions at PL0 and PL1.

If EL3 is implemented and is using AArch32, and NSACR.NSASEDIS is 1, CPACR.ASEDIS is RAO/WI in Non-secure state. This also applies when the CPACR.ASEDIS control is not implemented.

### Traps to Undefined mode of EL0 accesses to the Debug Communications Channel (DCC) registers

DBGDSCRext.UDCCdis traps EL0 accesses to the DCC registers to Undefined mode:

**1** EL0 accesses to the DCC registers are trapped to Undefined mode

**0** This control has no effect on EL0 accesses to the DCC registers.

Traps of EL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

Table G1-31 shows the registers for which accesses are trapped.

**Table G1-31 Register accesses trapped to Undefined mode when DBGDSCRext.UDCCdis is 1**

| Traps from | Registers |
|---|---|
| EL0 | DBGDSCRint, DBGDTRRXint, DBGDTRTXint, DBGDIDR, DBGDSAR, DBGDRAR |

**Note**

All accesses to these registers are trapped, including `LDC` and `STC` accesses to DBGDTRTXint and DBGDTRRXint, and `MRRC` accesses to DBGDSAR and DBGDRAR.

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Traps to Undefined mode of EL0 accesses to the Generic Timer registers

CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN} trap EL0 accesses to the Generic Timer registers to Undefined mode, as follows:

- CNTKCTL.PL0PTEN traps EL0 accesses to the physical timer registers.
- CNTKCTL.PL0VTEN traps EL0 accesses to the virtual timer registers.
- CNTKCTL.PL0PCTEN traps EL0 accesses to the frequency register and physical counter register.
- CNTKCTL.PL0VCTEN traps EL0 accesses to the frequency register and virtual counter register.

For all of these controls:

**1** This control has no effect on EL0 accesses to the corresponding registers.

**0** EL0 accesses to the corresponding registers are trapped to Undefined mode.

Accesses to the frequency register, CNTFRQ, are only trapped if CNTKCTL.PL0PCTEN and CNTKCTL.PL0VCTEN are both 0.

Table G1-32 shows the registers for which accesses are trapped.

**Table G1-32 Register accesses trapped to Undefined mode by CNTKCTL trap controls**

| Traps from | Trap control | Registers |
|---|---|---|
| EL0 | PL0PTEN | CNTP_CTL, CNTP_CVAL, CNTP_TVAL |
| | PL0VTEN | CNTV_CTL, CNTV_CVAL, CNTV_TVAL |
| | PL0PCTEN | CNTFRQ, CNTPCT |
| | PL0VCTEN | CNTFRQ, CNTVCT |

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.

### Traps to Undefined mode of EL0 accesses to Performance Monitors registers

PMUSERENR.{ER, CR, SW, EN} trap EL0 accesses to the Performance Monitors registers to Undefined mode. For each of these controls:

**1** This control has no effect on EL0 accesses to the corresponding registers.

**0** EL0 accesses to the corresponding registers are trapped to Undefined mode.

For those Performance Monitors registers that more than one PMUSERENR.{ER, CR, SW, EN} control applies to, accesses are only trapped if all controls that apply are set to 0.

The accesses that these trap controls trap might be reads, writes, or both.

**Note**

- The architecture does not provide traps on Performance Monitors register accesses through the memory-mapped external debug interface.
- If the Performance Monitors Extension is not implemented, the Performance Monitors registers, including PMUSERENR, are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-33 shows the registers for which EL0 accesses are trapped. For each register, the table shows the type of access trapped.

**Table G1-33 Register accesses trapped to Undefined mode when disabled from EL0**

| Traps from | Trap control | Registers | Access type |
|---|---|---|---|
| EL0 | ER | PMXEVCNTR, PMEVCNTR<n> | R |
| | | PMSELR | RW |
| | CR | PMCCNTR, accessed using an `MRC` | R |
| | CR | PMCCNTR, accessed using an `MRRC` | R |
| | SW | PMSWINC | W |
| | EN | PMCNTENSET, PMCNTENCLR, PMCR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR | RW |

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### G1.19.3 EL2 configurable controls

These controls are ignored in Secure state.

Table G1-34 shows the System registers that contain these controls.

**Table G1-34 System registers that contain instruction enables and disables, and trap controls**

| Register name | Register description |
|---|---|
| FPEXC | Floating-point Exception Control Register |
| HCR | Hypervisor Configuration Register |
| HSTR | Hypervisor System Trap Register |
| HCPTR | Hyp Architectural Feature Trap Register |
| HDCR | Hyp Debug Control Register |

**Note**

- FPEXC.EN is a control that is in a System register provided by PL1. However, it results in exceptions taken to Hyp mode.
- For completeness, Table G1-35 includes the HCR.TGE routing control, that is described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.

Table G1-35 summarizes the controls.

**Table G1-35 Instruction enables and disables, and trap controls, for exceptions taken to Hyp mode**

| Control | Control type[a] | Description |
|---|---|---|
| HSCTLR.{SED, ITD}<br>HSCTLR.CP15BEN | D<br>E | *Disabling or enabling EL2 use of AArch32 deprecated functionality* on page G1-3857 |
| HCR.{TRVM, TVM} | T | *Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers* on page G1-3858 |
| HCR.HCD | D | *Disabling Non-secure state execution of HVC instructions* on page G1-3859 |
| HCR.TGE | R | *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790 |
| HCR.TTLB | T | *Traps to Hyp mode of Non-secure EL1 execution of TLB maintenance instructions* on page G1-3859 |
| HCR.{TSW, TPC, TPU} | T | *Traps to Hyp mode of Non-secure EL1 execution of cache maintenance instructions* on page G1-3860 |
| HCR.TAC | T | *Traps to Hyp mode of Non-secure EL1 accesses to the Auxiliary Control Register* on page G1-3860 |
| HCR.TIDCP | T | *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page G1-3860 |
| HCR.TSC | T | *Traps to Hyp mode of Non-secure EL1 execution of SMC instructions* on page G1-3861 |
| HCR.{TID0, TID1, TID2, TID3} | T | *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the ID registers* on page G1-3862 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G1-35 Instruction enables and disables, and trap controls, for exceptions taken to Hyp mode (continued)**

| Control | Control type[a] | Description |
|---|---|---|
| HCR.{TWI, TWE} | T | *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-3865 |
| HCPTR.{TCP11, TCP10} | T | *General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers* on page G1-3865 |
| FPEXC.EN | T | *Enabling access to the SIMD and floating-point registers* on page G1-3866 |
| HCPTR.TASE | T | *Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality* on page G1-3866 |
| HCPTR.TCPAC | T | *Traps to Hyp mode of Non-secure EL1 accesses to the CPACR* on page G1-3867 |
| HCPTR.TTA | T | *Traps to Hyp mode of Non-secure System register accesses to trace registers* on page G1-3867 |
| HSTR.{T0-T3, T5-T13, T15} | T | *General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to CP15 System registers* on page G1-3868 |
| HDCR.{TDRA, TDOSA, TDA} | T | *Traps to Hyp mode of Non-secure System register accesses to debug registers* on page G1-3869 |
| CNTHCTL.{PL1PCEN, PL1PCTEN} | T | *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the Generic Timer registers* on page G1-3871 |
| HDCR.{TPM, TPMCR} | T | *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page G1-3872 |

a. See Table G1-36.

**Table G1-36 Control types, for exceptions taken to Hyp mode**

| Abbreviation | Type | See |
|---|---|---|
| D | Disable | *Instruction enables and instruction disables* on page G1-3846 |
| E | Enable | *Instruction enables and instruction disables* on page G1-3846 |
| R | Routing control | *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790 |
| T | Trap | *Trap controls* on page G1-3846 |

Also see the following:

- *Register access instructions* on page G1-3847.
- *Instructions that fail their condition code check*.
- *Instructions that are UNPREDICTABLE* on page G1-3857.

## Instructions that fail their condition code check

For UNDEFINED instructions that fail their condition code check, see *Conditional execution of undefined instructions* on page G1-3813.

For an instruction that has a Hyp trap set, that fails its condition code check:

- Unless the trap description states otherwise, it is IMPLEMENTATION DEFINED whether the instruction:
  - Generates a Hyp Trap exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Executes as a NOP.

Any implementation must be consistent in its handling of instructions that fail their condition code check. This means that:

- Whenever a Hyp trap is set on an instruction it must either:
  — Always generate a Hyp Trap exception.
  — Always treat the instruction as a NOP.
- The IMPLEMENTATION DEFINED part of the requirements of *Conditional execution of undefined instructions* on page G1-3813 must be consistent with the handling of Hyp traps on instructions that fail their condition code check. Table G1-37 shows this:

**Table G1-37 Consistent handling of instructions that fail their condition code check**

| Behavior of conditional UNDEFINED instruction[a] | Hyp trap on instruction that fails its condition code check[b] |
|---|---|
| Executes as a NOP | Executes as a NOP |
| Generates an Undefined Instruction exception | Generates a Hyp Trap exception |

a. As defined in *Conditional execution of undefined instructions* on page G1-3813. In Non-secure EL0 and EL1 modes, this applies only if no Hyp trap is set for the instruction, otherwise see the behavior in the other column of the table.
b. For a trapped instruction executed in a Non-secure EL1 or EL0 mode.

**Note**

Hyp traps on WFE and WFI instructions generate Hyp Trap exceptions only if the instruction passes its condition code check. See *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-3865.

### Instructions that are UNPREDICTABLE

For an instruction that is UNPREDICTABLE, the behavior of the instruction when it is disabled or trapped is UNPREDICTABLE. The architecture permits a trapped instruction to generate a Hyp Trap exception, but does not require it to do so.

**Note**

UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or lower Exception level using instructions that are not UNPREDICTABLE. This means that disabling or trapping an instruction changes the set of instructions that might be executed in Non-secure state at EL1 or EL0. This indirectly affects the permitted behavior of UNPREDICTABLE instructions.

If no instructions are trapped, the attempted execution of an UNPREDICTABLE instruction in a Non-secure EL1 or EL0 mode must not generate a Hyp Trap exception.

### Disabling or enabling EL2 use of AArch32 deprecated functionality

Table G1-38 on page G1-3858 shows the deprecated AArch32 functionality that might have disable controls in the HSCTLR:

- The SED control is always implemented.
- Whether each of the ITD, CP15BEN controls is implemented is IMPLEMENTATION DEFINED. If a control is not implemented then the associated functionality cannot be disabled.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These HSCTLR controls apply only to execution at EL2 using AArch32. When an instruction is disabled by one of these controls, it is UNDEFINED at EL2, meaning it is undefined in Hyp mode.

**Table G1-38 EL2 controls for disabling and enabling EL2 use of AArch32 deprecated functionality**

| Deprecated AArch32 functionality | Instruction enable or disable in the HSCTLR | Disabled instructions |
|---|---|---|
| SETEND instructions | SED[a] | SETEND instructions |
| Some uses of IT instructions | ITD[b] | See the HSCTLR.IT description |
| Accesses to the CP15 DMB, DSB, and ISB barrier operations | CP15BEN[c] | MCR accesses to the CP15DMB, CP15DSB, and CP15ISB |

a. SETEND instruction disable. SETEND instructions are disabled when the value of this field is 1.
b. IT instruction disable. If this control is implemented, some uses of IT instructions are disabled when the value of this field is 1.
c. CP15 memory barrier enable. If this control is implemented, the specified register accesses are disabled when the value of CP15BEN is 0.

**Note**

- These controls have no effect on instructions executed in any mode other than Hyp mode. The SCTLR provides similar controls that apply to execution in other modes.
- The uses of the IT instruction, and use of the CP15DMB, CP15DSB, and CP15ISB barrier instructions, are deprecated for performance reasons.

### Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers

HCR.{TRVM, TVM} trap Non-secure EL1 accesses to the virtual memory control registers to Hyp mode:

**HCR.TRVM, for read accesses:**

**1** Non-secure EL1 reads of the virtual memory control registers are trapped to Hyp mode.

**0** This control has no effect on Non-secure EL1 reads of the virtual memory control registers.

**HCR.TVM, for write access:**

**1** Non-secure EL1 writes to the virtual memory control registers are trapped to Hyp mode.

**0** This control has no effect on Non-secure EL1 writes to the virtual memory control registers.

Table G1-39 shows the registers for which:

- Reads are trapped to Hyp mode when HCR.TRVM is 1.
- Writes are trapped to Hyp mode when HCR.TVM is 1.

The table also shows how the exceptions are reported in HSR.

**Table G1-39 Register read and write accesses trapped when HCR.{TRVM, TVM} are 1**

| Traps from | Registers | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL1 | SCTLR, TTBR0, TTBR1, TTBCR, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR | Trapped MCR or MRC access to CP15, using EC value 0x03<br>Trapped MCRR or MRRC access to CP15, using EC value 0x04 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

These registers are not accessible at EL0.

### Disabling Non-secure state execution of HVC instructions

HCR.HCD disables Non-secure state execution of HVC instructions:

**1** HVC instructions are UNDEFINED at EL2 and Non-secure EL1. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

**0** HVC instruction execution is enabled at EL2 and Non-secure EL1.

**Note**

HVC instructions are always UNDEFINED at EL0.

HCR.HCD is only implemented if EL3 is not implemented. Otherwise, it is RES0. See the HCR register description.

Table G1-40 shows how the exceptions are reported in HSR.

**Table G1-40 Instruction that causes exceptions when HCR.HCD is 1**

| Attempted execution in | Disabled instruction | Syndrome reporting in HSR |
|---|---|---|
| Hyp mode | HVC | Exception for an unknown reason, using EC value 0x00 |
| Mode other than Hyp mode | HVC | Not applicable |

### Traps to Hyp mode of Non-secure EL1 execution of TLB maintenance instructions

In the ARMv8-A architecture, the system instruction encoding space includes TLB maintenance instructions.

HCR.TTLB traps Non-secure EL1 execution of TLB maintenance instructions to Hyp mode:

**1** Any attempt to execute a TLBI instruction at Non-secure EL1 is trapped to Hyp mode.

**0** This control has no effect on the Non-secure EL1 execution of TLBI instructions.

Table G1-41 shows the instructions that are trapped, and how the exceptions are reported in HSR.

**Table G1-41 Instructions trapped to Hyp mode when HCR.TTLB is 1**

| Traps from | Trapped instructions | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL1 | TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS, ITLBIALL, ITLBIMVA, ITLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL. | Trapped MCR or MRC access to CP15, using EC value 0x03 |

**Note**

These instructions are always UNDEFINED at EL0.

For more information about these instructions, see *The scope of TLB maintenance instructions* on page G4-4069.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Traps to Hyp mode of Non-secure EL1 execution of cache maintenance instructions

HCR.{TSW, TPC, TPU} trap cache maintenance instructions to Hyp mode. When one of these controls is 1, any attempt to execute a corresponding CP15 c7 instructions at Non-secure EL1 is trapped to Hyp mode.

**Table G1-42 Controls for trapping cache maintenance instructions to Hyp mode**

| Trap control | Trapped instructions |
|---|---|
| HCR.TSW | Data or unified cache maintenance by set/way |
| HCR.TPC | Data or unified cache maintenance to point of coherency |
| HCR.TPU | Cache maintenance to point of unification |

Table G1-43 shows the instructions that are trapped to Hyp mode, and how the exceptions are reported in HSR.

**Table G1-43 Instructions trapped to Hyp mode when HCR.{TSW, TPC, TPU} are 1**

| Traps from | Trap control | Trapped instructions | Syndrome reporting in HSR |
|---|---|---|---|
| Non-secure EL1 | TSW | DCISW, DCCSW, DCCISW | Trapped MCR or MRC access to CP15, using EC value 0x3 |
| | TPC | DCIMVAC, DCCIMVAC, DCCMVAC | |
| | TPU | ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAU | |

**Note**

These instructions are always UNDEFINED at EL0.

For more information about these instructions, see *Cache maintenance instructions, functional group* on page G4-4169.

### Traps to Hyp mode of Non-secure EL1 accesses to the Auxiliary Control Register

HCR.TAC traps Non-secure EL1 accesses to the Auxiliary Control Registers to Hyp mode:

**1** Non-secure EL1 accesses to the Auxiliary Control Registers are trapped to Hyp mode.

**0** This control has no effect on Non-secure EL1 accesses to the Auxiliary Control Registers.

Table G1-44 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR:

**Table G1-44 Register accesses trapped to Hyp mode when HCR.TAC is 1**

| Traps from | Registers | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL1 | ACTLR and, if implemented, ACTLR2. | Trapped MCR or MRC access to CP15 access, using EC value 0x03 |

**Note**

The ACTLR and ACTLR2 are not accessible at EL0.

### Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations

The lockdown, DMA, and TCM features of the ARMv8-A architecture are IMPLEMENTATION DEFINED. The architecture reserves the encodings of a number of System registers for control of these features.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

HCR.TIDCP traps the execution of System register access instructions that access these registers, as follows:

**1** At Non-secure EL1, any attempt to execute an MCR or MRC instruction with a reserved register encoding shown in Table G1-45 is trapped to Hyp mode.

At Non-secure EL0, it is IMPLEMENTATION DEFINED whether attempts to execute MCR or MRC instructions with reserved register encodings are:

- Trapped to Hyp mode.
- UNDEFINED, and the PE takes the Undefined Instruction exception to Non-secure Undefined mode.

Any lockdown fault in the memory system caused by the use of these operations in Non-secure state generates a Data Abort exception that is taken to Hyp mode.

**0** This control has no effect on Non-secure EL0 and EL1 System register access instructions with reserved register encodings shown in Table G1-45.

**Note**

This means that a Hyp Trap exception taken from Non-secure EL1 to Hyp mode, generated because of a configuration setting in HCR.TIDCP is a higher priority exception than an Undefined Instruction exception generated because either the System register encoding is unallocated or because the register is never accessible at EL1. As *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 shows, this is an exception to the general exception prioritization rules, that prioritize most Undefined Instruction exceptions taken to Undefined mode above traps to EL2.

Table G1-45 shows the register encodings for which accesses are trapped to Hyp mode, and how the exceptions are reported in HSR.

**Table G1-45 Encodings trapped to Hyp mode when HCR.TIDCP is 1**

| Traps from | Register encodings | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL0 and EL1 | An access to any of the following encodings:<br>• CRn==c9, opc1=={0-7}, CRm=={c0-c2, c5-c8}, opc2=={0-7}. See *VMSAv8-32 CP15 c9 register summary* on page G4-4144.<br>• CRn==c10, opc1=={0-7}, CRm=={c0, c1, c4, c8}, opc2=={0-7}. See *VMSAv8-32 CP15 c10 register summary* on page G4-4145.<br>• CRn==c11, opc1=={0-7}, CRm=={c0-c8, c15}, opc2=={0-7}. See *VMSAv8-32 CP15 c11 register summary* on page G4-4145. | Trapped MCR or MRC access to CP15, using EC value 0x03 |

An implementation can also include IMPLEMENTATION DEFINED registers that provide additional controls, to give finer-grained control of the trapping of IMPLEMENTATION DEFINED features.

**Note**

ARM expects the trapping of Non-secure User mode accesses to these functions to Hyp mode to be unusual, and used only when the hypervisor is virtualizing User mode operation. ARM strongly recommends that unless the hypervisor must virtualize User mode operation, a Non-secure User mode access to any of these functions generates an Undefined Instruction exception, as it would if the implementation did not include EL2. The PE then takes this exception to Non-secure Undefined mode.

### Traps to Hyp mode of Non-secure EL1 execution of SMC instructions

HCR.TSC traps Non-secure EL1 execution of SMC instructions to Hyp mode:

**1** Any attempt to execute an SMC instruction at Non-secure EL1 is trapped to Hyp mode, regardless of the value of SCR.SCD.

**0** This control has no effect on Non-secure EL1 execution of SMC instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-46 shows how the exceptions are reported in HSR:

**Table G1-46 SMC Instruction trapped to Hyp mode when HCR.TSC is 1**

| Traps from | Trapped instruction | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL1 | *SMC* on page F6-2979 | Trapped SMC instruction execution in AArch32 state, using EC value 0x13 |

The ARMv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

**Note**

- This trap is implemented only if the implementation includes EL3.
- SMC instructions are always UNDEFINED at EL0.
- HCR.TSC traps execution of the SMC instruction. It is not a routing control for the SMC exception. Hyp Trap and SMC exceptions have different preferred return addresses.

For more information about SMC instructions, see *SMC* on page F6-2979.

### Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the ID registers

Other than the MIDR, MPIDR, and PMCR.N, the ID registers are divided into groups, with a trap control in the HCR for each group.

**Table G1-47 ID register groups**

| Trap control | Register group |
|---|---|
| HCR.TID0 | *ID group 0, Primary device identification registers* on page G1-3863 |
| HCR.TID1 | *ID group 1, Implementation identification registers* on page G1-3863 |
| HCR.TID2 | *ID group 2, Cache identification registers* on page G1-3864 |
| HCR.TID3 | *ID group 3, Detailed feature identification registers* on page G1-3864 |

These controls trap register accesses from Non-secure EL0 or EL1 to Hyp mode, as follows:

**HCR.TID0** When 1, any attempt at Non-secure EL0 or EL1 to read any register in ID group 0 is trapped to Hyp mode.

**HCR.TID1** When 1, any attempt at Non-secure EL1 to read any register in ID group 1 is trapped to Hyp mode.

**HCR.TID2** When 1, any attempt at Non-secure EL0 or EL1 to read any register in ID group 2, and any attempt at Non-secure EL0 or EL1 to write to the CSSELR, is trapped to Hyp mode.

**HCR.TID3** When 1, any attempt at Non-secure EL1 to read any register in ID group 3 is trapped to Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the MIDR and MPIDR, and for PMCR.N, the architecture provides read/write aliases. The original register becomes accessible only from Hyp mode and Secure state, and a Non-secure EL0 or EL1 read of the original register returns the value of the read/write alias. This substitution is invisible to the EL0 or EL1 software reading the register.

**Table G1-48 ID register substitution**

| Register | Original | Alias, EL2 using AArch32 |
|---|---|---|
| Main ID | MIDR | VPIDR |
| Multiprocessor Affinity | MPIDR | VMPIDR |
| Performance Monitors Control Register | PMCR.N | HDCR.HPMN |

Reads of the MIDR, MPIDR, or PMCR.N from Hyp mode or Secure state are unchanged by the implementation of EL2, and access the physical registers.

**Note**

- If the optional Performance Monitors Extension is not implemented, HDCR.HPMN is RES0 and PMCR is reserved.
- HDCR.HPMN also affects whether a Performance Monitors counter can be accessed from Non-secure EL1 or EL0. See the register description of HDCR for more information.
- PMCR contains other fields that identify the implementation. For more information about trapping accesses to the PMCR, see *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page G1-3872.

A reset into AArch32 state sets VPIDR to the MIDR value, VMPIDR to the MPIDR value, and HDCR.HPMN to the PMCR.N value.

### *ID group 0, Primary device identification registers*

These registers identify some top-level implementation choices.

Table G1-49 shows the registers that are in ID group 0 for traps to Hyp mode, and how the exceptions are reported in HSR.

**Table G1-49 ID group 0 registers**

| Traps from | Group 0 registers | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL1 | FPSID | Trapped `MRC` or `VMRS` access to CP10, for ID group traps, using EC value `0x08` |
| Non-secure EL0 and EL1 | JIDR | Trapped `MCR` or `MRC` access to CP14, using EC value `0x05` |

**Note**

The FPSID is not accessible at EL0.

If HCPTR.{TCP11, TCP10} traps accesses to SIMD and floating-point functionality, then for a read of FPSID, that trap has priority over this trap.

When the FPSID is accessible, a `VMSR FPSID, <Rt>` instruction is permitted but is ignored. The execution of this `VMSR` instruction is not trapped by the ID group 0 trap.

### *ID group 1, Implementation identification registers*

These registers often provide coarse-grained identification mechanisms for implementation-specific features.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-50 shows the registers that are in ID group 1 for traps to Hyp mode, and how the exceptions are reported in HSR.

**Table G1-50 ID group 1 registers**

| Traps from | Group 1 registers | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL1 | TCMTR, TLBTR, REVIDR, AIDR | Trapped MCR or MRC access to CP15, using EC value 0x03 |

#### *ID group 2, Cache identification registers*

These registers describe and control the cache implementation.

Table G1-51 shows the registers that are in ID group 2 for traps to Hyp mode, and how the exceptions are reported in HSR.

**Table G1-51 ID group 2 registers**

| Traps from | Group 2 registers | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL0 and EL1 | CTR, CCSIDR, CLIDR, CSSELR | Trapped MCR or MRC access to CP15, using EC value 0x03 |

#### *ID group 3, Detailed feature identification registers*

These registers provide detailed information about the features of the implementation.

**Note**

. These registers are called the CPUID registers. There is no requirement for this trap to apply to those registers that the CPUID Identification Scheme defines as reserved. See *The CPUID identification scheme* on page G4-4163.

Table G1-52 shows the registers that are in ID group 3 for traps to Hyp mode, and how the exceptions are reported in HSR.

**Table G1-52 ID group 3 registers**

| Traps from | Group 3 registers | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL1 | MVFR0, MVFR1, MVFR2. | Trapped MRC or VMRS access to CP10, using EC value 0x08 |
| | ID_PFR0, ID_PFR1, ID_DFR0, ID_AFR0.<br>ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, and ID_MMFR4, except that if ID_MMFR4 is implemented as RAZ/WI then it is IMPLEMENTATION DEFINED whether reads of the register are trapped.<br>ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5.<br>Any MRC access to any of the following CP15 encodings:<br>• opc1 == 0, CRn == c0, CRm == {c3-c7}, opc2 == {0, 1}.<br>• opc1 == 0, CRn == c0, CRm == c3, opc2 == 2.<br>• opc1 == 0, CRn == c0, CRm == c5, opc2 == {4, 5}.<br>It is IMPLEMENTATION DEFINED whether HCR.TID3 traps MRC accesses to CP15 encodings in the following range that are not already mentioned in this table:<br>• CRn == c0, opc1 == 0, CRm == {c2-c7}, opc2 == {0-7}. | Trapped MCR or MRC access to CP15, using EC value 0x03 |

If HCPTR traps accesses to SIMD and floating-point functionality, then for reads of MVFR0, MVFR1, and MVFR2, that trap has priority over this trap.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions

HCR.{TWE, TWI} trap Non-secure EL0 and EL1 execution of WFE and WFI instructions to Hyp mode:

**HCR.TWE:**

**1** Any attempt to execute a WFE instruction at Non-secure EL0 or EL1 is trapped to Hyp mode, if the instruction would otherwise have caused the PE to enter a low-power state.

**0** This control has no effect on Non-secure EL0 or EL1 execution of WFE instructions.

**HCR.TWI:**

**1** Any attempt to execute a WFI instruction at Non-secure EL0 or EL1 is trapped to Hyp mode, if the instruction would otherwise have caused the PE to enter a low-power state.

**0** This control has no effect on Non-secure EL0 or EL1 execution of WFI instructions.

Table G1-53 shows how the exceptions are reported in HSR.

**Table G1-53 Instructions trapped to Hyp mode when HCR.{TWE, TWI} are 1**

| Traps from | Trapped instructions | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL0 and EL1 | WFE | Trapped WFI or WFE instruction, using EC value 0x01 |
| | WFI | |

The attempted execution of a conditional WFE or WFI instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait For Event and Send Event* on page G1-3834.
- *Wait For Interrupt* on page G1-3837.

### General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers

HCPTR.{TCP11, TCP10} trap Non-secure accesses to the SIMD and floating-point registers to Hyp mode:

**0b11** All Non-secure accesses to the SIMD and floating-point registers are trapped to Hyp mode. Trapped instructions generate:

- Hyp Trap exceptions, if the exception is taken from Non-secure EL0 or EL1.
- Undefined Instruction exceptions taken to Hyp mode, if the exception is taken from EL2.

**0b00** This control has no effect on Non-secure accesses to the SIMD and floating-point registers.

**Note**

Software must set HCPTR.TCP11 and HCPTR.TCP10 to the same value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-54 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

**Table G1-54 Register accesses trapped to Hyp mode when HCPTR.{TCP11, TCP10} are both 0b11**

| Traps from | Registers | Syndrome reporting in HSR |
|---|---|---|
| Non-secure state | FPSID, MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers. See *Advanced SIMD and floating-point System registers* on page G1-3843. | Trapped access to SIMD and floating-point register, resulting from HCPTR, using EC value 0x07[a] |

a. VMSR accesses to the FPSID are ignored, but for the purposes of this trap the architecture defines a VMSR access to the FPSID from EL1 or higher as an access to a SIMD and floating-point register.

If EL3 is implemented and is using AArch32, and NSACR.{cp11, cp10} are both set to 0, then HCPTR.{TCP11, TCP10} behave as RAO/WI, regardless of their actual value.

For more information about SIMD and floating-point support, see *Advanced SIMD and floating-point support* on page G1-3841.

### Enabling access to the SIMD and floating-point registers

FPEXC.EN is an instruction enable that enables access to the SIMD and floating-point registers from all Exception levels, but does not control the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

FPEXC.EN is a PL1 control that also applies at EL2. See *Enabling access to the SIMD and floating-point registers* on page G1-3851.

### Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality

If implemented as an RW field, HCPTR.TASE can trap Non-secure execution of Advanced SIMD instructions to Hyp mode, as follows. This trap applies only when HCPTR.{TCP11, TCP10} are both 0:

**1** Any attempt to execute an Advanced SIMD instruction in Non-secure state is trapped to Hyp mode. Trapped instructions generate:
- Hyp Trap exceptions, if the exception is taken from Non-secure EL0 or EL1.
- Undefined Instruction exceptions taken to Hyp mode, if the exception is taken from EL2.

**0** This control has no effect on Non-secure execution of Advanced SIMD instructions.

When the control is not implemented, meaning the HCPTR.TASE field is RAZ/WI, the HCPTR does not provide a trap to Hyp mode of the Non-secure execution of Advanced SIMD instructions, other than the HCPTR.{TCP11, TCP10} trap that applies to Non-secure execution of both floating-point and Advanced SIMD instructions.

Table G1-29 on page G1-3851 shows the instructions that are trapped, and how the exceptions are reported in HSR.

**Table G1-55 Instructions trapped to Hyp mode when HCPTR.TASE is set to 1**

| Traps from | Instructions | Syndrome reporting in HSR |
|---|---|---|
| Non-secure state | All Advanced SIMD instructions that are not also floating-point instructions. For more information see *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-2332. | Trapped access to SIMD and floating-point register, resulting from HCPTR, using EC value 0x07 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If EL3 is implemented and is using AArch32, and NSACR.NSASEDIS is 1, then HCPTR.TASE behaves as RAO/WI, regardless of its actual value. This behavior also applies when the HCPTR.TASE control is not implemented.

### Traps to Hyp mode of Non-secure EL1 accesses to the CPACR

HCPTR.TCPAC traps Non-secure EL1 accesses to the CPACR to Hyp mode:

**1** Non-secure EL1 accesses to the CPACR are trapped to Hyp mode.

**0** This control has no effect on Non-secure EL1 accesses to the CPACR.

Table G1-56 shows how the exceptions are reported in HSR:

**Table G1-56 Register accesses trapped to Hyp mode when HCPTR.TCPAC is 1**

| Traps from | Register | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL1 | CPACR | Trapped `MCR` or `MRC` access to CP15, using EC value `0x03` |

**Note**

- The CPACR is not accessible at EL0.
- In ARMv7 and earlier versions of the ARM architecture, one function of the CPACR is as an ID register that identifies what coprocessor functionality is implemented. Legacy software might use this identification mechanism. A hypervisor can use this trap to emulate this mechanism.

### Traps to Hyp mode of Non-secure System register accesses to trace registers

If implemented, the HCPTR.TTA control traps System register accesses to the trace registers from Non-secure state to Hyp mode, as follows:

**1** Non-secure System register accesses to the trace registers are trapped to Hyp mode. Trapped instructions generate:
- Hyp Trap exceptions, if the exception is taken from Non-secure EL0 or EL1.
- Undefined Instruction exceptions taken to Hyp mode, if the exception is taken from EL2.

**0** This control has no effect on Non-secure System register accesses to the trace registers.

If the HCPTR.TTA control is not implemented, then HCPTR.TTA is RAO/WI. See the register description for more information.

**Note**

- System register accesses to the trace registers use the CP14 encoding space.
- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED. A resulting Undefined Instruction exception is higher priority than an HCPTR.TTA Hyp Trap exception.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see *Register access instructions* on page G1-3847.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

Table G1-57 shows the registers for which accesses are trapped to Hyp mode when HCPTR.TTA is 1, and how the exceptions are reported in HSR.

**Table G1-57 Register accesses trapped to Hyp mode when HCPTR.TTA is 1**

| Traps from | Registers | Syndrome reporting in HSR |
|---|---|---|
| Non-secure state | System register accesses to all implemented trace registers | For accesses using:<br>• MCR or MRC instructions, trapped MCR or MRC access to CP14, using EC value 0x05.<br>• MCRR or MRRC instructions, trapped MCRR or MRRC access to CP14, using EC value 0x0C. |

If EL3 is implemented and is using AArch32, and NSACR.NSTRCDIS is 1, then HCPTR.TTA behaves as RAO/WI, regardless of its actual value. This behavior applies, also, when the HCPTR.TTA control is not implemented.

### General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to CP15 System registers

HSTR.{T0-T3, T5-T13, T15} trap Non-secure EL0 and EL1 accesses, using MCR, MRC, MCRR, or MRRC instructions, to the CP15 System registers, by the accessed primary CP15 register number, {c0-c3, c5-c13, c15}.

When an HSTR.Tn trap control is:

**1** Non-secure EL1 accesses to the corresponding CP15 System registers are trapped to Hyp mode.

EL0 accesses to the corresponding CP15 System registers trapped to Hyp mode if they would not be UNDEFINED if the bit was zero.

**0** This control has no effect on Non-secure EL0 or EL1 accesses to the corresponding CP15 System registers.

**Note**

- This means that a Hyp Trap exception taken from EL1 to EL2, generated because of a configuration setting in HSTR.Tn, is a higher priority exception than an Undefined Instruction exception generated because either the CP15 register encoding is unallocated or because a register is never accessible at Non-secure EL1. As *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 shows, this is an exception to the general exception prioritization rules, that prioritize most Undefined Instruction exceptions taken to Undefined mode above traps to EL2. This prioritization includes any access from Non-secure EL1 to a register that is only accessible in Secure state. So, for example, an access to the SCR from Non-secure EL1:
  - When the value of HSTR.T1 is 0, generates an Undefined Instruction exception.
  - When the value of HSTR.T1 is 1, generates a Hyp Trap exception.
- The primary CP15 register is identified by:
  - The CRn argument to MCR and MRC instructions.
  - The CRm argument to MCRR and MRRC instructions.

Table G1-58 shows the accesses that are trapped, and how the exceptions are reported in HSR.

**Table G1-58 Accesses trapped to Hyp mode when an HSTR.Tn trap is enabled**

| Traps from | Trap control | Trapped accesses | Syndrome reporting in HSR |
|---|---|---|---|
| Non-secure EL0 and EL1[a] | Tn | MCR and MRC instructions, with CRn set to n | Trapped MCR or MRC access to CP15, using EC value 0x03 |
| | | MCRR and MRRC instructions, with CRm set to n | Trapped MCRR or MRRC access to CP15, using EC value 0x04 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

a. As described in this section, traps from EL1 apply whenever the value of HSTR.Tn is 1. Traps from EL0 apply only if the value of HSTR.Tn is 1 and the access would not be UNDEFINED if the value of HSTR.Tn was 0.

For example, when HSTR.T7 is 1, considering only accesses from Non-secure EL1:

- Any 32-bit CP15 access from a Non-secure PL1 mode, using an `MRC` or `MCR` instruction with CRn set to c7, is trapped to Hyp mode.
- Any 64-bit CP15 access from a Non-secure PL1 mode, using an `MRRC` or `MCRR` instructions with CRm set to c7, is trapped to Hyp mode.

**Note**

- Bits[4,14] of the HSTR are reserved, RES0. Although the Generic Timer control registers are implemented in CP15 c14, EL2 does not provide a trap on accesses to the Generic Timer CP15 registers.
- An implementation might provide additional controls, in IMPLEMENTATION DEFINED registers, to provide finer-grained control of control of trapping of IMPLEMENTATION DEFINED features.

#### *CP15 register with IMPLEMENTATION DEFINED access permission from EL0*

For a trapped primary CP15 register, if it is IMPLEMENTATION DEFINED whether, when the corresponding trap control is 0, an access from Non-secure User mode is UNDEFINED, then when the corresponding trap control is 1, it is IMPLEMENTATION DEFINED whether an access from Non-secure User mode generates:

- A Hyp Trap exception.
- An Undefined Instruction exception taken to Non-secure Undefined mode.

**Note**

ARM expects that trapping of Non-secure User mode accesses to CP15 to Hyp mode will be unusual, and used only when the hypervisor must virtualize User mode operation. ARM recommends that, whenever possible, Non-secure User mode accesses to CP15 behave as they would if the processor did not implement EL2, generating an Undefined Instruction exception taken to Non-secure Undefined mode if the architecture does not support the User mode access.

### Traps to Hyp mode of Non-secure System register accesses to debug registers

HDCR.{TDRA, TDOSA, TDA} trap Non-secure System register accesses to debug registers to Hyp mode, as follows:

- HDCR.(TDRA, TDA} trap Non-secure EL0 and EL1 accesses.
- HDCR.TDOSA traps Non-secure EL1 accesses.

**Note**

EL2 does not provide traps of debug register accesses through the optional memory-mapped external debug interface.

System register accesses to the debug registers can have side-effects. When a System register access is trapped to Hyp mode, no side-effects occur before the exception is taken to Hyp mode. See *Register access instructions* on page G1-3847.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-59 shows the subsections that list the accesses trapped. The subsections describe how the traps are reported in HSR.

**Table G1-59 Traps of Non-secure EL0 and EL1 accesses to debug registers**

| Trap control | Subsection |
|---|---|
| HDCR.TDRA | *Trapping Non-secure System register accesses to Debug ROM registers* |
| HDCR.TDOSA | *Trapping Non-secure System register accesses to powerdown debug registers* |
| HDCR.TDA | *Trapping general Non-secure System register accesses to debug registers* on page G1-3871 |

**Note**

System register accesses to debug registers use the CP14 encoding space.

### *Trapping Non-secure System register accesses to Debug ROM registers*

HDCR.TDRA traps Non-secure EL0 and EL1 System register accesses to the Debug ROM registers to Hyp mode:

**1** Non-secure EL0 or EL1 System register accesses to the Debug ROM registers are trapped to Hyp mode.

**0** This control has no effect on Non-secure EL0 and EL1 System register accesses to the Debug ROM registers.

Table G1-60 shows the register accesses that are trapped, and how the exceptions are reported in HSR:

**Table G1-60 Register accesses trapped to Hyp mode when HDCR.TDRA is 1**

| Traps from | Registers | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL0 and EL1 | DBGDRAR, DBGDSAR | For accesses using:<br>• `MCR` or `MRC` instructions, trapped `MCR` or `MRC` access to CP14, using EC value `0x05`.<br>• `MRRC` instructions, trapped `MRRC` access to CP14, using EC value `0x0C`. |

If HDCR.TDE or HCR.TGE is 1, behavior is as if HDCR.TDRA is 1 other than for the purpose of a direct read.

### *Trapping Non-secure System register accesses to powerdown debug registers*

HDCR.TDOSA traps Non-secure EL1 System register accesses to the powerdown debug registers to Hyp mode:

**1** Non-secure EL1 System register accesses to the powerdown debug registers are trapped to Hyp mode.

**0** This control has no effect on Non-secure EL1 System register accesses to the powerdown debug registers.

Table G1-61 shows the register accesses that are trapped, and how the exceptions are reported in HSR.

**Table G1-61 Register accesses trapped to Hyp mode when HDCR.TDOSA is 1**

| Traps from | Registers | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL1 | DBGOSLSR, DBGOSLAR, DBGOSDLR, DBGPRCR<br>Any IMPLEMENTATION DEFINED integration registers.<br>Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by HDCR.TDOSA. | Trapped `MCR` or `MRC` access to CP14, using EC value `0x05` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

These registers are not accessible at EL0.

If HDCR.TDE or HCR.TGE is 1, behavior is as if HDCR.TDOSA is 1 other than for the purpose of a direct read.

#### *Trapping general Non-secure System register accesses to debug registers*

HDCR.TDA traps Non-secure EL0 and EL1 System register accesses to the debug registers that are not mentioned in either of the following:

- *Traps to Hyp mode of Non-secure System register accesses to debug registers* on page G1-3869.
- *Trapping Non-secure System register accesses to powerdown debug registers* on page G1-3870.

This means that HDCR.TDA traps to Hyp mode Non-secure EL0 and EL1 System register accesses to all debug registers except the following:

- Non-secure System register accesses to DBGDRAR or DBGDSAR. The HDCR.TDRA trap traps these accesses.
- Non-secure System register access to DBGOSLSR, DBGOSLAR, DBGOSDLR, or DBGPRCR. The HDCR.TDOSA trap traps these accesses.

HDCR.TDA does not trap accesses to DBGDTRTXint or DBGDTRRXint when the PE is in Debug state.

When HDCR.TDA is:

**1** Non-secure EL0 or EL1 CP14 accesses to any of the registers shown in Table G1-62 are trapped to Hyp mode.

**0** This control has no effect on Non-secure EL0 or EL1 CP14 accesses to the registers shown in Table G1-62.

Table G1-62 shows how the exceptions are reported in HSR.

**Table G1-62 Accesses trapped to Hyp mode when HDCR.TDA is 1**

| Traps from | Trapped accesses | Syndrome reporting in HSR |
|---|---|---|
| Non-secure EL0 and EL1 | Accesses to the DBGDIDR, DBGDSCRint, DBGDCCINT, DBGDTRRXint, DBGDTRTXint, DBGWFAR, DBGVCR, DBGDSCRext, DBGDTRTXext, DBGDTRRXext, DBGBVR<n>, DBGBCR<n>, DBGBXVR<n>, DBGWCR<n>, DBGWVR<n>, DBGCLAIMSET, DBGCLAIMCLR, DBGAUTHSTATUS, DBGDEVID, DBGDEVID1, DBGDEVID2, and DBGOSECCR | For accesses using `MCR` or `MRC` instructions, trapped `MCR` or `MRC` access to CP14, using EC value `0x05` |
| | `STC` accesses to DBGDTRRXint.<br>`LDC` accesses to DBGDTRTXint. | Trapped `LDC` or `STC` access to CP14, using EC value `0x06` |

If HDCR.TDE or HCR.TGE is 1, behavior is as if HDCR.TDA is 1 other than for the purpose of a direct read.

### Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the Generic Timer registers

CNTHCTL.{PL1PCEN, PL1PCTEN} trap Non-secure EL0 and EL1 accesses to the Generic Timer registers to Hyp mode, as follows:

- CNTHCTL.PL1PCEN traps Non-secure EL0 and EL1 accesses to the physical timer registers.
- CNTHCTL.PL1PCTEN traps Non-secure EL0 and EL1 accesses to the physical counter register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For each of these controls:

**1** This control has no effect on Non-secure EL0 and EL1 accesses to the registers shown in Table G1-63.

**0** Non-secure EL0 and EL1 accesses are trapped to Hyp mode.

Table G1-63 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

**Table G1-63 Register accesses trapped to Hyp mode by CNTHCTL trap controls**

| Traps from | Trap control | Registers | Syndrome reporting in HSR |
|---|---|---|---|
| Non-secure EL0 and EL1 | PL1PCEN | CNTP_CTL, CNTP_CVAL, CNTP_TVAL | For accesses using:<br>• `MCR` or `MRC` instructions, trapped `MCR` or `MRC` CP15 access, using EC value `0x03`<br>• `MCRR` or `MRRC` instructions, trapped `MCRR` or `MRRC` CP15 access, using EC value `0x04` |
| | PL1PCTEN | CNTPCT | Trapped `MCRR` or `MRRC` CP15 access, using EC value `0x04` |

### Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers

If the Performance Monitors Extension is implemented, HDCR.{TPM, TPMCR} trap Non-secure EL0 and EL1 accesses to the Performance Monitors registers to Hyp mode:

**HDCR.TPM:**

**1** Non-secure EL0 and EL1 accesses to all Performance Monitors registers are trapped to Hyp mode.

**0** This control has no effect on Non-secure EL0 and EL1 accesses to the Performance Monitors registers.

**HDCR.TPMCR:**

**1** Non-secure EL0 and EL1 accesses to the Performance Monitors Control Register are trapped to Hyp mode.

**Note**

The conditions for this trap are identical to those for the trap controlled by HDCR.TPM

**0** This control has no effect on Non-secure EL0 and EL1 accesses to the Performance Monitors Control Registers.

**Note**

- EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.
- If the Performance Monitors Extension is not implemented, HDCR.{TPM, TPMCR} are RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G1-64 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

**Table G1-64 Register accesses trapped to Hyp mode when HDCR.{TPM, TPMCR} are 1**

| Traps from | Trap control | Registers | Syndrome reporting in HSR |
|---|---|---|---|
| Non-secure EL0 and EL1 | TPM | PMCR, PMCNTENSET, PMCNTENCLR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMUSERENR, PMINTENSET, PMINTENCLR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR | For accesses using:<br>• MCR or MRC instructions, trapped MCR or MRC CP15 access, using EC value 0x03.<br>• MCRR or MRRC instructions, trapped MCRR or MRRC CP15 access, using EC value 0x04. |
| | TPMCR | PMCR | Trapped MCR or MRC access to CP15, using EC value 0x03 |

**Note**

HDCR.HPMN affects whether a counter can be accessed from Non-secure EL1 or EL0. See the register description of HDCR for more information.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G1.19.4 EL3 configurable controls

Table G1-65 shows the System registers that contain these controls.

**Table G1-65 System registers that contain instruction enables and disables, and trap controls**

| Register name | Register description |
|---|---|
| SCR | Secure Configuration Register |
| NSACR | Non-secure Access Control Register |

Table G1-66 summarizes the controls.

**Table G1-66 EL3 Instruction enables and disables, and trap controls**

| Control | Type of control[a] | Trap |
|---|---|---|
| SCR.{TWE, TWI} | T | *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode* on page G1-3875 |
| SCR.HCE | E | *Enabling EL2 and Non-secure EL1 execution of HVC instructions* on page G1-3876 |
| SCR.SCD | D | *Disabling SMC instructions* on page G1-3876 |
| NSACR.NSTRCDIS | D | *Disabling Non-secure System register access to the trace registers* on page G1-3877 |
| NSACR.{cp11, cp10} | E | *Enabling Non-secure access to SIMD and floating-point functionality* on page G1-3877 |
| NSACR.NSASEDIS | D | *Disabling Non-secure access to Advanced SIMD functionality* on page G1-3878 |

a. See Table G1-67.

**Table G1-67 Control types, for AArch32 EL3 controls**

| Abbreviation | Type | See |
|---|---|---|
| D | Disable | *Instruction enables and instruction disables* on page G1-3846 |
| E | Enable | *Instruction enables and instruction disables* on page G1-3846 |
| T | Trap | *Trap controls* on page G1-3846 |

Also see the following:

- *Register access instructions* on page G1-3847.
- *Instructions that fail their condition code check*.
- *Instructions that are UNPREDICTABLE* on page G1-3875.

#### Instructions that fail their condition code check

For UNDEFINED instructions that fail their condition code check, see *Conditional execution of undefined instructions* on page G1-3813.

For an instruction that has a Monitor trap set, that fails its condition code check:

- Unless the trap description states otherwise, it is IMPLEMENTATION DEFINED whether the instruction:
  - Generates a Monitor Trap exception.
  - Executes as a `NOP`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Any implementation must be consistent in its handling of instructions that fail their condition code check. This means that:

- Whenever a Monitor trap is set on such an instruction it must either:
  - Always generate a Monitor trap exception.
  - Always treat the instruction as a NOP.
- The IMPLEMENTATION DEFINED part of the requirements of *Conditional execution of undefined instructions* on page G1-3813 must be consistent with the handling of Monitor traps on instructions that fail their condition code check. Table G1-68 shows this:

**Table G1-68 Consistent handling of instructions that fail their condition code check**

| Behavior of conditional UNDEFINED instruction[a] | Monitor trap on instruction that fails its condition code check[b] |
|---|---|
| Executes as a NOP | Executes as a NOP |
| Generates an Undefined Instruction exception | Generates a Monitor trap exception |

a. As defined in *Conditional execution of undefined instructions* on page G1-3813. In Non-secure EL0 and EL1 modes, this applies only if no Monitor trap is set for the instruction, otherwise see the behavior in the other column of the table.

b. For a trapped instruction executed in a Non-secure EL1 or EL0 mode.

**Note**

When SCR{TWE, TWI} is set so that conditional WFE and WFI instructions are trapped to Monitor mode, the attempted execution of a conditional WFE or WFI instruction is only trapped if the instruction passes its condition code check. See *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode*.

### Instructions that are UNPREDICTABLE

For an instruction that is UNPREDICTABLE, the behavior of the instruction when the instruction is trapped or disabled is UNPREDICTABLE. The architecture permits a trapped instruction to generate a Monitor Trap exception, but does not require it to do so.

**Note**

UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or lower Exception level using instructions that are not UNPREDICTABLE. This means that disabling or trapping an instruction changes the set of instructions that might be executed in modes other than Monitor mode. This affects, indirectly, the permitted behavior of UNPREDICTABLE instructions.

If no instructions are trapped, the attempted execution of an UNPREDICTABLE instruction in a mode other than Monitor mode must not generate a Monitor Trap exception.

### Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode

SCR{TWE, TWI} trap WFE and WFI instructions to Monitor mode:

**SCR.TWE**
- **1** Any attempt to execute a WFE instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state.
- **0** This control has no effect on the execution of WFE instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SCR.TWI** **1** Any attempt to execute a WFI instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state.

**0** This control has no effect on the execution of WFI instructions.

For PL0 and PL1, these traps apply to WFE and WFI instruction execution in both Security states.

The attempted execution of a conditional WFE or WFI instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- *Wait For Event and Send Event* on page G1-3834.
- *Wait For Interrupt* on page G1-3837.

### Enabling EL2 and Non-secure EL1 execution of HVC instructions

SCR.HCE enables EL2 and Non-secure EL1 execution of HVC instructions:

**1** HVC instruction execution is enabled at EL2 and Non-secure EL1.

**0** HVC instructions are:

- UNDEFINED at Non-secure EL1. The Undefined Instruction exception is taken to Undefined mode.
- UNPREDICTABLE at EL2. Behavior is one of the following:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.

**Note**

- If EL2 is not implemented, SCR.HCE is RES0 and HVC is UNDEFINED.
- HVC instructions are always UNDEFINED at EL0 and in Secure state.

### Disabling SMC instructions

SCR.SCD disables SMC instructions:

**1** **In Non-secure state**

SMC instructions are UNDEFINED. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

**In Secure state**

Behavior is one of the following:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.

**0** SMC instructions are enabled.

**Note**

- SMC instructions are always UNDEFINED at EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When the value of HCR.TSC is 1, any attempted execution of an SMC instruction at Non-secure EL1 is trapped to EL2, regardless of the value of SCR.SCD, see *Traps to Hyp mode of Non-secure EL1 execution of SMC instructions* on page G1-3861. As *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 shows, this is an exception to the general exception prioritization rules, that prioritize most Undefined Instruction exceptions taken to Undefined mode above traps to a higher Exception level.

### Disabling Non-secure System register access to the trace registers

NSACR.NSTRCDIS disables Non-secure System register accesses to the trace registers, from all Privilege levels:

**1** Non-secure state accesses are disabled. Secure state accesses are enabled. If the PE is in Non-secure state:

- CPACR.TRCDIS behaves as RAO/WI, regardless of its actual value. See *Traps to Undefined mode of PL0 and PL1 System register accesses to trace registers* on page G1-3850.
  This behavior applies even if the CPACR.TRCDIS control is not implemented. See the referenced section for more information.
- HCPTR.TTA behaves as RAO/WI, regardless of its actual value. See *Traps to Hyp mode of Non-secure System register accesses to trace registers* on page G1-3867.

**0** There is no effect on accesses to CPACR.TRCDIS and HCPTR.TTA.

**Note**

- System register accesses to the trace registers use the CP14 encoding space.
- NSACR.NSTRCDIS might be implemented as RAZ/WI. See the NSACR register description for more information.
- The ETMv4 architecture does not permit EL0 to access the trace registers. If the ARMv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED.
- EL3 does not provide Non-secure access controls on trace register accesses through the optional memory-mapped external debug interface.

### Enabling Non-secure access to SIMD and floating-point functionality

NSACR.{cp11, cp10} enable Non-secure access to the SIMD and floating-point registers, from all Privilege levels:

**0b11** All accesses, from both Security states, are enabled.

**0b00** Non-secure state accesses are disabled. Secure state accesses are enabled. If the PE is in Non-secure state:

- CPACR.{cp11, cp10} behave as RAZ/WI. See *Enabling PL0 and PL1 accesses to the SIMD and floating-point registers* on page G1-3851.
- HCPTR.{TCP11, TCP10} behave as RAO/WI. See *General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers* on page G1-3865.

**Note**

Software must set NSACR.cp11 and NSACR.cp10 to the same value.

For more information about SIMD and floating-point support, see *Advanced SIMD and floating-point support* on page G1-3841.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Disabling Non-secure access to Advanced SIMD functionality

NSACR.NSASEDIS disables Non-secure accesses to the Advanced SIMD functionality, from all Privilege levels:

**1** Non-secure state accesses are disabled. Secure accesses are enabled. If the PE is in Non-secure state:

- CPACR.ASEDIS behaves as RAO/WI. See *Disabling PL0 and PL1 execution of Advanced SIMD instructions* on page G1-3852.
- HCPTR.TASE behaves as RAO/WI. See *Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality* on page G1-3866.

These behaviors apply even if one or both of the CPACR.ASEDIS and HCPTR.TASE controls is not implemented. See the referenced sections for more information.

**0** There is no effect on CPACR.ASEDIS and HCPTR.TASE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G1.19.5 Pseudocode description of configurable instruction enables, disables, and traps

The pseudocode function `AArch32.CheckITEnabled()` checks whether the T32 IT instruction is enabled.

The pseudocode function `AArch32.CheckSETENDEnabled()` checks whether the `SETEND` instruction is disabled.

The pseudocode function for `AArch32.CheckForSMCTrap()` checks for traps on an `SMC` instruction.

The `AArch32.CheckForWFxTrap()` pseudocode function checks for traps on `WFE` and `WFI` instructions:

#### Pseudocode description of enabling SIMD and floating-point functionality

The `AArch32.CheckAdvSIMDOrFPEnabled()` and `AArch32.CheckFPAdvSIMDTrap()` pseudocode functions take appropriate action if an SIMD or floating-point instruction is used when the SIMD and floating-point functionality is not enabled or is trapped.

The `CheckAdvSIMDOrVFPEnabled()`, `CheckAdvSIMDEnabled()`, and `CheckVFPEnabled()` wrapper functions support the `AArch32.CheckAdvSIMDOrFPEnabled()` and `AArch32.CheckFPAdvSIMDTrap()` functions.

The `AArch32.CheckAdvSIMDOrFPEnabled()`, `AArch32.CheckFPAdvSIMDTrap()`, `CheckAdvSIMDOrVFPEnabled()`, `CheckAdvSIMDEnabled()`, and `CheckVFPEnabled()` functions are described in Chapter J1 *ARMv8 Pseudocode*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter G2
# AArch32 Self-hosted Debug

When the PE is using self-hosted debug, it generates *debug exceptions*. This chapter describes the AArch32 self-hosted debug exception model. It is organized as follows:

**Introductory information**

- *About debug exceptions* on page G2-3883.
- *The debug exception enable controls* on page G2-3886.

**The debug Exception model**

- *Routing debug exceptions* on page G2-3887.
- *Enabling debug exceptions from the current Privilege level and Security state* on page G2-3889.
- *The effect of powerdown on debug exceptions* on page G2-3891.
- *Summary of permitted routing and enabling of debug exceptions* on page G2-3892.
- *Pseudocode description of debug exceptions* on page G2-3894.

**The debug exceptions**

- *Software Breakpoint Instruction exceptions* on page G2-3895.
- *Breakpoint exceptions* on page G2-3898.
- *Watchpoint exceptions* on page G2-3923.
- *Vector Catch exceptions* on page G2-3937.

**Synchronization requirements**

The behavior of self-hosted debug after changes to System registers, or after changes to the authentication interface, but before a *Context Synchronization Operation* (CSO) guarantees the effects of the changes:

- *Synchronization and debug exceptions* on page G2-3945.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Note**

**Definition of a debugger**

Within this chapter, *debugger* means that part of an operating system, or higher level of system software, that handles debug exceptions and programs the debug System registers. An operating system with rich application environments might provide debug services that support a debugger user interface executing at EL0. From the architectural perspective the debug services are the debugger.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G2.1 About debug exceptions

Debug exceptions occur during normal program flow, if a debugger has programmed the PE to generate them. For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger might enable one or more debug exceptions. The debug exceptions that can be generated in an AArch32 stage 1 translation regime are:

- *Software Breakpoint Instruction exceptions*.
- *Breakpoint exceptions*, generated by hardware breakpoints.
- *Watchpoint exceptions* on page G2-3884, generated by hardware watchpoints.
- *Vector Catch exceptions* on page G2-3884.

**Note**

In addition, *Software Step exceptions* can be generated in stage 1 of an AArch32 translation regime. However, these are always taken to AArch64 state. *Software Step exceptions* on page D2-1638 describes this.

The PE can only generate a particular debug exception when both:

1. Debug exceptions are enabled from the current Exception level and Security state.

   See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-3889. Software Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.

2. A debugger has enabled that particular debug exception.

   All of the debug exceptions except for Software Breakpoint Instruction exceptions have an enable control contained in the DBGDSCRext. See *The debug exception enable controls* on page G2-3886.

**Note**

If *halting is allowed* and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause entry to Debug state instead of causing debug exceptions. In Debug state, the PE is halted.

For the definition of halting is allowed, see *Halting allowed and halting prohibited* on page H2-4821.

The following list summarizes each of the debug exceptions:

**Software Breakpoint Instruction exceptions**

*Breakpoint instructions* generate these. Breakpoint instructions are instructions that software developers can use to cause exceptions at particular points in the program flow.

The breakpoint instruction in the A32 and T32 instruction sets is `BKPT #<immediate>`. Whenever one of these is committed for execution, the PE takes a Software Breakpoint Instruction exception.

**PE behavior**

Software Breakpoint Instruction exceptions cannot be masked. The PE takes Software Breakpoint Instruction exceptions regardless of both of the following:

- The current Privilege level and AArch32 mode.
- The current Security state.

For more information, see *Software Breakpoint Instruction exceptions* on page G2-3895.

**Breakpoint exceptions**

The ARMv8-A architecture provides 2-16 hardware breakpoints. These can be programmed to generate Breakpoint exceptions based on particular instruction addresses, or based on particular PE contexts, or both.

For example, a software developer might program a hardware breakpoint to generate a Breakpoint exception whenever the instruction with address `0x1000` is committed for execution.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The ARMv8-A architecture supports the following types of hardware breakpoint for use in stage 1 of an AArch32 translation regime:

- Address:
  - Address Match.
  - Address Mismatch.

  Comparisons are made with the virtual address of each instruction in the program flow.
- Context:
  - Context ID Match. Matches with the Context ID value held in the CONTEXTIDR.
  - VMID Match. Matches with the VMID value held in the VTTBR.
  - Context ID and VMID Match. Matches with both the Context ID and the VMID value.

An Address breakpoint can link to a Context breakpoint, so that the Address breakpoint only generates a Breakpoint exception if the PE is in a particular context when the address match or mismatch occurs.

A breakpoint generates a Breakpoint exception whenever an instruction that causes a match is committed for execution.

**PE behavior**

If halting is allowed and EDSCR.HDE is 1, hardware breakpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 *Debug State*.

Otherwise:

- If debug exceptions are enabled, hardware breakpoints cause Breakpoint exceptions.
- If debug exceptions are disabled, hardware breakpoints are ignored.

For more information, see *Breakpoint exceptions* on page G2-3898.

### Watchpoint exceptions

The ARMv8-A architecture provides 2-16 hardware watchpoints. These can be programmed to generate Watchpoint exceptions based on accesses to particular data addresses, or based on accesses to any address in a data address range.

For example, a software developer might program a hardware watchpoint to generate a Watchpoint exception on an access to any address in the data address range `0x1000 - 0x101F`.

A hardware watchpoint can link to a hardware breakpoint, if the hardware breakpoint is a *Linked Context* type. In this case, the watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs.

The smallest data address size that a watchpoint can be programmed to match on is a byte. A single watchpoint can be programmed to match on one or more bytes.

A watchpoint generates a Watchpoint exception whenever an instruction that initiates an access that causes a match is committed for execution.

**PE behavior**

If halting is allowed and EDSCR.HDE is 1, hardware watchpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 *Debug State*.

Otherwise:

- If debug exceptions are enabled, hardware watchpoints cause Watchpoint exceptions.
- If debug exceptions are disabled, hardware watchpoints are ignored.

For more information, see *Watchpoint exceptions* on page G2-3923.

### Vector Catch exceptions

These are used to trap exceptions. The ARMv8-A architecture provides two forms of vector catch, *address-matching* and *exception-trapping*. Only one form can be implemented.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Whichever form is implemented, a debugger must enable Vector Catch exceptions for one or more exception vectors by programming the DBGVCR. Generation of Vector Catch exceptions is then as follows:

- For the address-matching form, a Vector Catch exception is generated whenever the virtual address of an instruction matches a vector that Vector Catch exceptions are enabled for.
- For the Exception-trapping form, a Vector Catch exception is generated as part of exception entry for exception types that correspond to vectors that Vector Catch exceptions are enabled for.

**PE behavior**

If debug exceptions are:

- Enabled, Vector Catch exceptions can be generated.
- Disabled, vector catch is ignored.

For more information, see *Vector Catch exceptions* on page G2-3937.

Table G2-1 summarizes PE behavior and shows the location of the pseudocode for each of the debug exceptions.

**Table G2-1 PE behavior and pseudocode for each of the debug exceptions**

| Debug exception | PE behavior if debug exceptions are: | | Pseudocode |
|---|---|---|---|
| | Enabled | Disabled | |
| Software Breakpoint Instruction exceptions | Takes the exception | Takes the exception | page G2-3897 |
| Breakpoint exceptions | Takes the exception[a] | Ignored | page G2-3918 |
| Watchpoint exceptions | Takes the exception[a] | Ignored | page G2-3934 |
| Vector Catch exceptions | Takes the exception | Ignored | page G2-3943 |

a. If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause the PE to enter Debug state instead of causing debug exceptions. See Chapter H2 *Debug State*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G2.2 The debug exception enable controls

The enable controls for each debug exception are as follows:

**Software Breakpoint Instruction exceptions**

None. Software Breakpoint Instruction exceptions are always enabled.

**Breakpoint exceptions**

DBGDSCRext.MDBGen, plus an enable control for each breakpoint, DBGBCR<n>.E.

**Watchpoint exceptions**

DBGDSCRext.MDBGen, plus an enable control for each watchpoint, DBGWCR<n>.E.

**Vector Catch exceptions**

DBGDSCRext.MDBGen.

In addition, for all debug exceptions other than Software Breakpoint Instruction exceptions, software must configure the controls that enable debug exceptions from the current Exception level and Security state. See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-3889.

The PE cannot take a debug exception if debug exceptions are disabled from either the current Exception level or the current Security state.

Software Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G2.3 Routing debug exceptions

Debug exceptions are usually routed to Abort mode. However, if EL2 is implemented, the following applies:

- Software Breakpoint Instruction exceptions taken from Hyp mode are routed to Hyp mode.

  All other debug exceptions are disabled from Hyp mode.

- The routing of debug exceptions taken from Non-secure PL1 and PL0 depends on HDCR.TDE:

  **1** Debug exceptions taken from Non-secure PL1 and PL0 are routed to Hyp mode.

  **0** Debug exceptions taken from Non-secure PL1 and PL0 are routed to Abort mode.

**Note**

If HCR.TGE is 1, HDCR.TDE is treated as being 1 except for the purpose of a direct read of HDCR.

Table G2-2 shows this.

**Table G2-2 The effect of the TGE and TDE control bits on debug exception routing**

| HCR.TGE | HDCR.TDE | Debug exceptions taken from Non-secure PL1 and PL0 are taken to: |
|---|---|---|
| 0 | 0 | Non-secure Abort mode |
| 0 | 1 | Hyp mode |
| 1 | X | Hyp mode |

**Note**

If EL2 is not implemented, the PE behaves as if both HCR.TGE and HDCR.TDE are 0.

The following tables show the routing of debug exceptions:

**Table G2-3 Routing when both EL3 and EL2 are implemented**

| HDCR.TDE[a] | Target AArch32 mode when executing in: | | | |
|---|---|---|---|---|
| | Non-secure: | | | Secure state |
| | PL0 | PL1 | PL2 | |
| 0 | Non-secure Abort mode | Non-secure Abort mode | Hyp mode[b] | Secure Abort mode |
| 1 | Hyp mode | Hyp mode | Hyp mode[b] | Secure Abort mode |

a. If HCR.TGE is 1, this bit is treated as being 1 other than for a direct read of HDCR.

b. Only applies to Software Breakpoint Instruction exceptions. All other debug exceptions are disabled.

**Table G2-4 Routing when EL3 is implemented and EL2 is not implemented**

| Target AArch32 mode when executing in: | |
|---|---|
| Non-secure state | Secure state |
| Non-secure Abort mode | Secure Abort mode |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G2-5 Routing when EL3 is not implemented and EL2 is implemented**

| HDCR.TDE[a] | Target AArch32 mode when executing in Non-secure: | | |
|---|---|---|---|
| | PL0 | PL1 | PL2 |
| 0 | Non-secure Abort mode | Non-secure Abort mode | Hyp mode[b] |
| 1 | Hyp mode | Hyp mode | Hyp mode[b] |

a. If HCR.TGE is 1, this bit is treated as being 1 other than for a direct read of HDCR.

b. Only applies to Software Breakpoint Instruction exceptions. All other debug exceptions are disabled.

### G2.3.1 Pseudocode description of routing debug exceptions

`DebugTarget()` returns the current debug target Exception level.

```
// DebugTarget()
// =============

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

`DebugTargetFrom()` returns the debug target Exception level for the specified Security state.

```
// DebugTargetFrom()
// =================
// Returns the debug exception target Exception level

bits(2) DebugTargetFrom(boolean secure)

    if HaveEL(EL2) && !secure then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G2.4 Enabling debug exceptions from the current Privilege level and Security state

A debug exception can only be taken if all of the following are true:

- The OS lock is unlocked.
- EDPRSR.DLK is 0.
- The debug exception is enabled from the current Privilege level.
- The debug exception is enabled from the current Security state.

Table G2-6 shows when debug exceptions are enabled from the current Privilege level.

**Table G2-6 Whether debug exceptions are enabled from the current Privilege level**

| Current Privilege level | Software Breakpoint Instruction exceptions | All other debug exceptions |
|---|---|---|
| PL2 | Enabled | Disabled |
| PL1 | Enabled | Enabled |
| PL0 | | |

Table G2-7 shows when debug exceptions are enabled from the current Security state.

**Table G2-7 Whether debug exceptions are enabled from the current Security state**

| Current Security state | Software Breakpoint Instruction exceptions | All other debug exceptions |
|---|---|---|
| Non-secure | Enabled | Enabled from PL1 and PL0 only. |
| Secure | Enabled | Depends on SDCR.SPD and SDER.SUIDEN.<br>See *Disabling debug exceptions from Secure state*. |

### G2.4.1 Disabling debug exceptions from Secure state

If EL3 is implemented, software executing at EL3 can enable or disable all debug exceptions taken from Secure PL1 other than Software Breakpoint Instruction exceptions, by using one of:

- The *Secure Privileged Debug* field, SDCR.SPD, if EL3 is using AArch32.
- The *AArch32 Secure Privileged Debug* field, MDCR_EL3.SPD32, if EL3 is using AArch64.

If debug exceptions are disabled from Secure PL1, software executing at Secure PL1 can set the *Secure User Invasive Debug Enable* bit, SDER.SUIDEN, to 1 to enable all debug exceptions taken from Secure PL0 other than Software Breakpoint Instruction exceptions.

**Note**

Software Breakpoint Instruction exceptions are always enabled.

The ARMv8-A architecture does not support disabling debug in Non-secure state.

**Note**

If the boot software that is executed when reset is deasserted programs SUIDEN and SPD so that all debug exceptions are disabled from Secure state, software operating at EL3 never has to switch any of the debug registers between the Security states.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G2.4.2 Pseudocode description of enabling debug exceptions

`AArch32.GenerateDebugExceptions()` determines whether debug exceptions are enabled from the current Exception level and Security state.

```
// AArch32.GenerateDebugExceptions()
// =================================

boolean AArch32.GenerateDebugExceptions()
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());
```

`AArch32.GenerateDebugExceptionsFrom()` determines whether debug exceptions are enabled from the specified Exception level and Security state.

```
// AArch32.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)

    if from == EL0 && !ELStateUsingAArch32(EL1, secure) then
        mask = bit UNKNOWN;                          // PSTATE.D mask, unused for EL0 case
        return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    if HaveEL(EL3) && secure then
        spd = (if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32);
        if spd<1> == '1' then
            enabled = spd<0> == '1';
        else
            // SPD == 0b01 is reserved, but behaves the same as 0b00.
            enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from != EL2;

    return enabled;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G2.5 The effect of powerdown on debug exceptions

*Debug OS Save and Restore sequences* on page H6-4930 describes the *powerdown save routine* and the *restore routine*.

When executing either routine, software must use the OS Lock to disable generation of all of the following:

- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.

This is because the generation of these exceptions depends on the state of the debug registers, and the state of the debug registers might be lost over these routines.

Debug exceptions other than Software Breakpoint Instruction exceptions are enabled only if both the OS Lock is unlocked and EDPRSR.DLK is 0.

Software Breakpoint Instruction exceptions are enabled regardless of the state of the OS Lock and EDPRSR.DLK.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G2.6 Summary of permitted routing and enabling of debug exceptions

Behavior is as follows:

**Software Breakpoint Instruction exceptions**

These are always enabled, regardless of the current Privilege level and Security state. Table G2-8 shows the routing of these. In the table, n/a means not applicable.

**Table G2-8 Routing of Software Breakpoint Instruction exceptions**

| Current Security state | HDCR.TDE[a]: | Target when enabled from: | | |
|---|---|---|---|---|
| | | PL0 | PL1 | PL2 |
| Secure | X | Secure Abort mode[b] | Secure Abort mode[b] | n/a |
| Non-secure | 0 | Non-secure Abort mode | Non-secure Abort mode | Hyp mode |
| | 1 | Hyp mode | Hyp mode | Hyp mode |

a. If EL2 is not implemented, behavior is as if the value of this bit is 0. Otherwise, if the value of HCR.TGE is 1, HDCR.TDE is treated as being 1 other than for a direct read of HDCR.

b. If EL3 is implemented and is using AArch32, Secure Abort mode is at EL3. Otherwise, Secure Abort mode is at EL1.

**All other debug exceptions**

The enabling and permitted routing is controlled by all of the following:

- SDCR.SPD.
- SDER.SUIDEN.
- HDCR.TDE.
- The IMPLEMENTATION DEFINED authentication interface.

Table G2-9 shows the valid combinations of the values of SDCR.SPD, SDER.SUIDEN, HDCR.TDE, and, in the *Auth* column, the input from the IMPLEMENTATION DEFINED authentication interface described by the pseudocode function `AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()`, see *Pseudocode description of AArch32 Self-Hosted Secure Privileged Invasive Debug Enabled* on page K2-5521. For each combination, the table shows where debug exceptions are enabled from and where they are taken to.

In the table, n/a means not applicable and a dash, -, means that debug exceptions are disabled from that Exception level.

**Table G2-9 Breakpoint, Watchpoint, and Vector Catch exceptions**

| Debug state | Lock[a] | Current Security state | SPD[b] | Auth[c] | SUIDEN | TDE[d] | Target AArch32 mode when enabled from: | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | PL0 | PL1 | PL2 |
| Yes | X | X | `0bXX` | X | X | X | - | - | - |
| No | 1 | X | `0bXX` | X | X | X | - | - | - |
| No | 0 | Secure | `0b00` | TRUE | 0 | X | - | - | n/a |
| No | 0 | Secure | `0b00` | TRUE | 1 | X | Secure Abort mode[e] | - | n/a |
| No | 0 | Secure | `0b00` | FALSE | X | X | Secure Abort mode[e] | Secure Abort mode[e] | n/a |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G2-9 Breakpoint, Watchpoint, and Vector Catch exceptions (continued)**

| Debug state | Lock[a] | Current Security state | SPD[b] | Auth[c] | SUIDEN | TDE[d] | Target AArch32 mode when enabled from: | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | PL0 | PL1 | PL2 |
| No | 0 | Secure | 0b10 | X | 0 | X | - | - | n/a |
| No | 0 | Secure | 0b10 | X | 1 | X | Secure Abort mode[e] | - | n/a |
| No | 0 | Secure | 0b11 | X | X | X | Secure Abort mode[e] | Secure Abort mode[e] | n/a |
| No | 0 | Non-secure | 0bXX | X | X | 0 | Non-secure Abort mode | Non-secure Abort mode | - |
| No | 0 | Non-secure | 0bXX | X | X | 1 | Hyp mode | Hyp mode | - |

a. The value of (OSLSR_EL1.OSLK OR EDPRSR.DLK).
b. If EL3 is not implemented, behavior is as if this is 0b11.
c. See the text that introduces this table for an explanation of the *Auth* on page G2-3892 column. An entry of TRUE indicates that the authentication mechanism permits the debug exceptions to be taken to their default target PE mode.
d. If HCR.TGE is 1, this bit is treated as being 1 other than for a direct read of HDCR. If EL2 is not implemented, behavior is as if TDE is 0.
e. If EL3 is implemented and is using AArch32, Secure Abort mode is at EL3. Otherwise, Secure Abort mode is at EL1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G2.7 Pseudocode description of debug exceptions

`DebugFault()` returns a `FaultRecord()` that indicates that a memory access has generated a debug exception:

```
// AArch32.DebugFault()
// ====================

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_Debug, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

The `Abort()` function processes `FaultRecord()`, as described in *Abort exceptions* on page G3-3985, and generates:

- Data Abort exceptions for watchpoints.
- Prefetch Abort exceptions for all other debug exceptions.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G2.8 Software Breakpoint Instruction exceptions

This section describes Software Breakpoint Instruction exceptions in an AArch32 translation regime.

**Note**

A T32 or A32 `BKPT` instruction executed at EL0 can generate a Software Breakpoint Instruction exception that is taken to an Exception level that is using AArch64.

For more information about the handling of these exceptions see *Software Breakpoint Instruction exceptions* on page D2-1650.

It contains the following subsections:

- *About Software Breakpoint Instruction exceptions*.
- *Breakpoint instruction in the A32 and T32 instruction sets*.
- *BKPT instructions as the first instruction in an IT block* on page G2-3896.
- *Exception syndrome information and preferred return address* on page G2-3896.
- *Pseudocode description of Software Breakpoint Instruction exceptions* on page G2-3897.

### G2.8.1 About Software Breakpoint Instruction exceptions

A *breakpoint* is a debug event that results from the execution of an instruction, based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint*.
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

*Software Breakpoint Instruction exceptions*, that this section describes, are software breakpoints. *Breakpoint exceptions* on page G2-3898 describes hardware breakpoints.

There is no enable control for Software Breakpoint Instruction exceptions. They are always enabled, and cannot be masked.

A Software Breakpoint Instruction exception is generated whenever a breakpoint instruction is committed for execution, regardless of all of the following:

- The current Exception level.
- The current Security state.
- Whether the *debug target Exception level*, $EL_D$, is using AArch64 or AArch32.

**Note**

- $EL_D$ is the Exception level that debug exceptions are targeting. See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-3889.
- Debuggers using breakpoint instructions must be aware of the ARMv8 rules for concurrent modification and execution of instructions. See *Concurrent modification and execution of instructions* on page B2-82.

### G2.8.2 Breakpoint instruction in the A32 and T32 instruction sets

The breakpoint instruction, in both instruction sets, is:

- `BKPT #<immediate>`

For details of the instruction encoding, see *BKPT* on page F6-2637.

#### About whether the BKPT instruction is conditional

In the T32 instruction set, `BKPT` instructions are always unconditional.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In the A32 instruction set:

- If the condition code field is AL, the BKPT instruction is unconditional.
- If the condition code field is anything other than AL, behavior is CONSTRAINED UNPREDICTABLE, and is one of the following:
  - The instruction is UNDEFINED.
  - The instruction is treated as a NOP instruction.
  - The instruction is executed unconditionally.
  - The instruction is executed conditionally.

### G2.8.3 BKPT instructions as the first instruction in an IT block

If the first instruction in an IT block is a T32 BKPT instruction, then in an implementation that supports the ITD control, if ITD field that applies to the current Exception level is:

**0** The BKPT instruction generates a Software Breakpoint Instruction exception.

**1** The combination of IT instruction and BKPT instruction is UNDEFINED. Either the IT instruction or the BKPT instruction generates an Undefined Instruction exception.

In such an implementation, to ensure consistent behavior when making the first instruction in one or more IT blocks a BKPT instruction, the debugger must replace the IT instruction.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

The ITD control fields are:

**HSCTLR.ITD** Applies to execution at EL2 when EL2 is using AArch32.

**SCTLR.ITD** Applies to execution at EL0 or EL1 when EL1 is using AArch32.

**SCTLR_EL1.ITD**
Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

**Note**

T32 BKPT instructions are always unconditional, even when they are inside an IT block. See:

- *Disabling or enabling PL0 and PL1 use of AArch32 deprecated functionality* on page G1-3849.
- *Disabling or enabling EL2 use of AArch32 deprecated functionality* on page G1-3857.

### G2.8.4 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information*.
- *Preferred return address* on page G2-3897.

#### Exception syndrome information

The PE takes a Software Breakpoint Instruction exception as either:

- A Prefetch Abort exception, if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp Trap exception, if it is taken to PL2 because either HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

**PL1 Abort mode**

The PE sets all of the following:

- DBGDSCRext.MOE to 0b0011, to indicate a Software Breakpoint Instruction exception.
- IFSR.FS to the code for a debug event, 0b00010.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The IFAR with an UNKNOWN value.

**PL2 Hyp mode**

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-10.
- Sets DBGDSCRext.MOE to `0b0011`, to indicate a Software Breakpoint Instruction exception.
- Sets the HIFAR to an UNKNOWN value.

**Table G2-10 Information recorded in the HSR**

| HSR field | Information recorded |
|---|---|
| *Exception Class*, EC | The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, `0x20`. |
| *Instruction Length*, IL | The PE sets this to:<br>• 0 for a T32 `BKPT` instruction.<br>• 1 for an A32 `BKPT` instruction. |
| *Instruction Specific Syndrome*, ISS | **ISS[24:10]** RES0.<br>**ISS[9]** *External Abort type* (EA). The PE sets this to 0.<br>**ISS[8:6]** RES0.<br>**ISS[5:0]** *Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`. |

**Note**

For information about how debug exceptions can be routed to PL2, see *Routing debug exceptions on page G2-3887*.

### Preferred return address

The preferred return address is the address of the breakpoint instruction, not the next instruction. This is different to the behavior of other exception-generating instructions, like `SVC`.

## G2.8.5 Pseudocode description of Software Breakpoint Instruction exceptions

`AArch32.SoftwareBreakpoint()` generates a Prefetch Abort exception that is taken from AArch32 state.

```
// AArch32.SoftwareBreakpoint()
// ============================

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);

    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH;           // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;

    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G2.9 Breakpoint exceptions

This section describes Breakpoint exceptions in stage 1 of an AArch32 translation regime.

It contains the following subsections:

- *About Breakpoint exceptions*.
- *Breakpoint types and linking of breakpoints* on page G2-3899.
- *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-3906.
- *Breakpoint instruction address comparisons* on page G2-3907.
- *Context comparisons* on page G2-3912.
- *Using breakpoints* on page G2-3912.
- *Exception syndrome information and preferred return address* on page G2-3917.
- *Pseudocode description of Breakpoint exceptions taken from AArch32 state* on page G2-3918.

### G2.9.1 About Breakpoint exceptions

A *breakpoint* is a debug event that results from the execution of an instruction, based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

*Breakpoint exceptions* are generated by *Breakpoint debug events*. Breakpoint debug events are generated by hardware Breakpoints. Software Breakpoints are described in *Software Breakpoint Instruction exceptions* on page G2-3895.

An implementation can include between 2-16 hardware breakpoints. DBGDIDR.BRPs shows how many are implemented.

To use an implemented hardware breakpoint, a debugger programs the following registers for the breakpoint:

- The *Breakpoint Control Register*, DBGBCR<n>. This contains controls for the breakpoint, for example an enable control.
- The *Breakpoint Value Register*, DBGBVR<n>. This holds a value used for breakpoint matching, that is one of:
  - An instruction virtual address.
  - A Context ID.
- If EL2 is implemented, the *Breakpoint Extended Value Register*, DBGBXVR<n>, that holds a VMID value used for breakpoint matching.

These registers are numbered, so that:

- DBGBCR1, DBGBVR1, and DBGBXVR1 are for breakpoint number one.
- DBGBCR2, DBGBVR2, and DBGBXVR2 are for breakpoint number two.
- …
- …
- DBGBCR<n>, DBGBVR<n>, and DBGBXVR<n> are for breakpoint number <n>.

A debugger can link a breakpoint that is programmed with an address and a breakpoint that is programmed with anything other than an address together, so that a Breakpoint debug event is only generated if both breakpoints match.

For each instruction in the program flow, all of the breakpoints are tested. When a breakpoint is tested, it generates a Breakpoint debug event if all of the following are true:

- The breakpoint is enabled. That is, the breakpoint enable control for it, DBGBCR<n>.E, is 1.
- The conditions specified in the DBGBCR<n> are met.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The comparisons with the values held in one or both of the DBGBVR<n> and DBGBXVR<n>, as applicable, are successful.
- If the breakpoint is linked to another breakpoint, the comparisons made by that other breakpoint are also successful.
- The instruction is committed for execution.

If all of these conditions are met, the breakpoint generates the Breakpoint debug event regardless of the following:

- Whether the instruction passes its condition code check.
- The instruction type.

If halting is allowed and EDSCR.HDE is 1, Breakpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are

- Enabled, Breakpoint debug events generate Breakpoint exceptions
- Disabled, Breakpoint debug events are ignored.

**Note**

The remainder of this Breakpoint exceptions section, including all subsections, describes breakpoints as generating Breakpoint exceptions.

However, the behavior described also applies if breakpoints are causing entry to Debug state.

*The debug exception enable controls* on page G2-3886 describes the enable controls for Breakpoint debug events.

### G2.9.2 Breakpoint types and linking of breakpoints

Each implemented breakpoint is one of the following:

- A *context-aware* breakpoint. This is a breakpoint that can be programmed to generate a Breakpoint exception on any one of the following:
  - An instruction address match.
  - An instruction address mismatch.
  - A Context ID match, with the value held in the CONTEXTIDR.
  - A VMID match, with the value held in the VTTBR.
  - Both a Context ID match and a VMID match.
- A breakpoint that is not context-aware. These can only be programmed to generate a Breakpoint exception on an instruction address match or an instruction address mismatch.

DBGDIDR.CTX_CMPs shows how many of the implemented breakpoints are context-aware breakpoints. At least one implemented breakpoint must be context-aware. The context-aware breakpoints are the highest numbered breakpoints.

Any breakpoint that is programmed to generate a Breakpoint exception on an instruction address match or mismatch is categorized as an *Address breakpoint*. Breakpoints that are programmed to match on anything else are categorized as *Context breakpoints*.

When a debugger programs a breakpoint to be an Address or a Context breakpoint, it must also program that breakpoint so that it is either:

- Used in isolation. In this case the breakpoint is called an *Unlinked breakpoint*.
- Enabled for linking to another breakpoint. In this case the breakpoint is called a *Linked breakpoint*.

By linking an Address breakpoint and a Context breakpoint together, the debugger can create a breakpoint pair that only generates a Breakpoint exception if the PE is in a particular context when an instruction address match or mismatch occurs. For example, a debugger might:

1. Program breakpoint number one to be a *Linked Address Match breakpoint*.
2. Program breakpoint number five to be a *Linked Context ID Match breakpoint*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

3. Link these two breakpoints together. A Breakpoint exception is only generated if both the instruction address matches and the Context ID matches.

The *Breakpoint Type* field for a breakpoint, DBGBCR<n>.BT, controls the breakpoint type and whether the breakpoint is enabled for linking. If BT[0] is 1, the breakpoint is enabled for linking.

Figure G2-1 shows all of the possible breakpoint types that stage 1 of an AArch32 translation regime supports, and their associated BT field values.

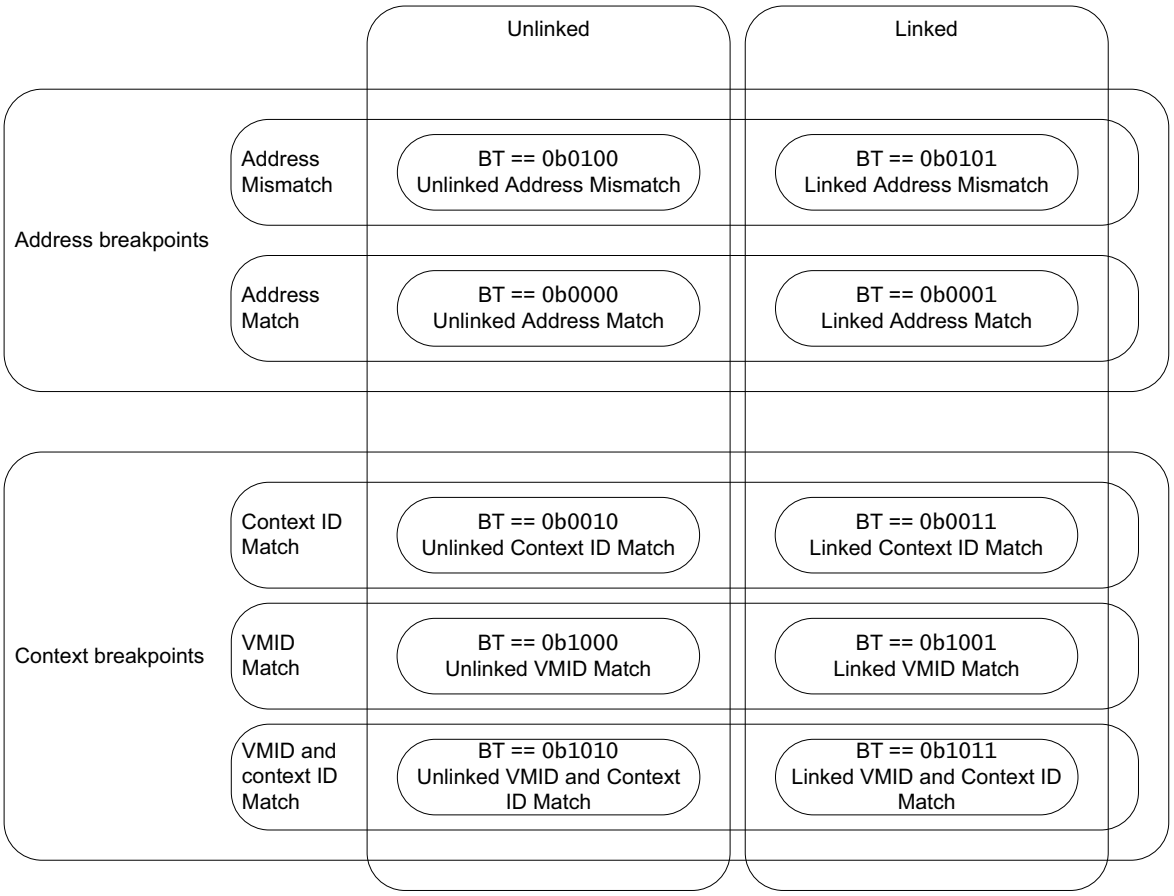



**Figure G2-1 Breakpoint types and their associated BT field values**

Address breakpoints can be programmed to generate Breakpoint exceptions on addresses that are halfword-aligned but not word-aligned. This makes it possible to breakpoint on T32 instructions. See *Specifying the halfword-aligned address that an Address breakpoint matches on* on page G2-3908.

## Rules for linking breakpoints

The rules for breakpoint linking are as follows:

- Only Linked breakpoint types can be linked.
- Any type of Linked Address breakpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, DBGBCR<n>.LBN, for the Linked Address breakpoint specifies the particular Linked Context breakpoint that the Linked Address breakpoint links to, and:
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Address breakpoint define the execution conditions that the breakpoint pair generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-3906.
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

- Linked Context breakpoint types can only be linked to. The LBN field for Context breakpoints is therefore ignored.
- Linked Address breakpoints cannot link to watchpoints. The LBN field can therefore only specify another breakpoint.
- If a Linked Address breakpoint links to a breakpoint that is not context-aware, the behavior of the Linked Address breakpoint is CONSTRAINED UNPREDICTABLE. See *Other usage constraints for Address breakpoints* on page G2-3916.
- If a Linked Address breakpoint links to an Unlinked Context breakpoint, the Linked Address breakpoint never generates any Breakpoint exceptions.
- Multiple Linked Address breakpoints can link to a single Linked Context breakpoint.

  **Note**

  Multiple Linked watchpoints can also link to a single Linked Context breakpoint. *Watchpoint exceptions* on page G2-3923 describes watchpoints.

These rules mean that a single Linked Context breakpoint might be linked to by all, or any combination of, the following:

- Multiple Linked Address Match breakpoints.
- Multiple Linked Address Mismatch breakpoints.
- Multiple Linked watchpoints.

It is also possible that a Linked Context breakpoint might have no breakpoints or watchpoints linked to it.

Figure G2-2 on page G2-3902 shows an example of permitted breakpoint and watchpoint linking.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

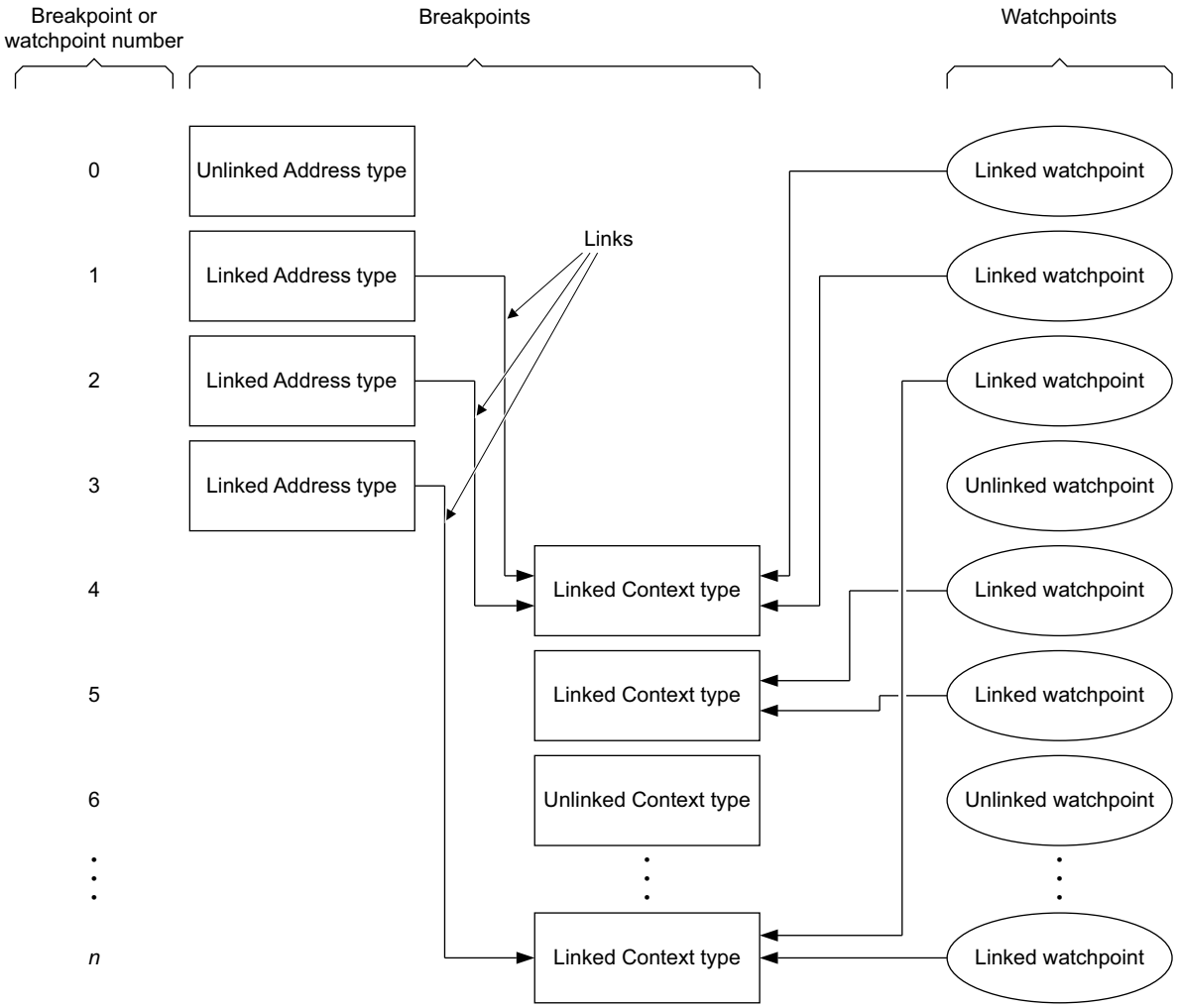



**Figure G2-2 The role of linking in Breakpoint and Watchpoint exception generation**

In Figure G2-2, each Linked Address breakpoint can only generate a Breakpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful. Similarly, each Linked watchpoint can only generate a Watchpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful.

## Breakpoint types defined by DBGBCRn.BT

The following list provides more detail about each breakpoint type:

**`0b0000`, Unlinked Address Match breakpoint**

Generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-3906.
- A successful address match, as described in *Breakpoint instruction address comparisons* on page G2-3907.

DBGBCR<n>.LBN for this breakpoint is ignored.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**0b0001, Linked Address Match breakpoint**

Generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>.{SSC, HMC, PMC} for this breakpoint. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-3906.
- A successful address match defined by this breakpoint, as described in *Breakpoint instruction address comparisons* on page G2-3907.
- A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

DBGBCR<n>.LBN for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

**0b0010, Unlinked Context ID Match breakpoint**

BT == 0b0010 is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-3906.
- A successful Context ID match, as described in *Context comparisons* on page G2-3912.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored

**0b0011, Linked Context ID Match breakpoint**

BT == 0b0011 is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page G2-3907.
  - A successful Context ID match defined by this breakpoint, as described in *Context comparisons* on page G2-3912.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page G2-3926.
  - A successful Context ID match defined by this breakpoint, as described in *Context comparisons* on page G2-3912.

DBGBCR<n>.{LBN, SSC, HMC, BAS PMC} for this breakpoint are ignored.

**0b0100, Unlinked Address Mismatch breakpoint**

Generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-3906.
- A successful address mismatch, as described in *Breakpoint instruction address comparisons* on page G2-3907.

DBGBCR<n>.LBN for this breakpoint is ignored.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**0b0101, Linked Address Mismatch breakpoint**

Generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-3906.
- A successful address mismatch defined by this breakpoint, as described in *Breakpoint instruction address comparisons* on page G2-3907.
- A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

DBGBCR<n>.LBN for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

**0b1000, Unlinked VMID Match breakpoint**

BT == 0b1000 is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-3906.
- A successful VMID match, as described in *Context comparisons* on page G2-3912.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored.

**0b1001, Linked VMID Match breakpoint**

BT == 0b1000 is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address Match breakpoint that links to this breakpoint. See *Breakpoint instruction address comparisons* on page G2-3907.
  - A successful VMID match defined by this breakpoint, as described in *Context comparisons* on page G2-3912.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page G2-3926.
  - A successful VMID match defined by this breakpoint, as described in *Context comparisons* on page G2-3912.

DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

**0b1010, Unlinked Context ID and VMID Match breakpoint**

BT == 0b1010 is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For context-matching breakpoints, generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page G2-3906.
- A successful Context ID match.
- A successful VMID match.

*Context comparisons* on page G2-3912 describes the requirements for a successful Context ID match and a successful VMID match.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored.

**`0b1011`, Linked Context ID and VMID Match breakpoint**

BT == `0b1011` is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-matching breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on all of the following:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see *Breakpoint instruction address comparisons* on page G2-3907.
  - A successful Context ID match defined by this breakpoint.
  - A successful VMID match defined by this breakpoint.
- Generation of a Watchpoint exception depends on all of the following:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see *Watchpoint data address comparisons* on page G2-3926.
  - A successful Context ID match defined by this breakpoint.
  - A successful VMID match defined by this breakpoint.

*Context comparisons* on page G2-3912 describes the requirements for a successful Context ID match and a successful VMID match by this breakpoint.

DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

**Note**

See *Reserved DBGBCR<n>.BT values* on page G2-3915 for the behavior of breakpoints programmed with reserved BT values.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G2.9.3 Execution conditions for which a breakpoint generates Breakpoint exceptions

Each breakpoint can be programmed so that it only generates Breakpoint exceptions for certain execution conditions. For example, a breakpoint might be programmed to generate Breakpoint exceptions only when the PE is executing at PL0 in Secure state.

DBGBCR<n>.{SSC, HMC, PMC} define the execution conditions the breakpoint generates Breakpoint exceptions for, as follows:

**Security State Control, SSC**

Controls whether the breakpoint generates Breakpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

**Note**

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the breakpoint is set.

**Higher Mode Control, HMC, and Privileged Mode Control, PMC**

HMC and PMC together control which AArch32 modes the breakpoint generates Breakpoint exceptions in.

Table G2-11 shows the valid combinations of the values of HMC, SSC, and PMC, and for each combination shows which Privilege levels breakpoints generate Breakpoint exceptions in.

In the table:

**Y or -** Means that a breakpoint programmed with the values of HMC, SSC and PMC shown in that row:
- **Y** Can generate Breakpoint exceptions in AArch32 modes at that Privilege level.
- **-** Cannot generate Breakpoint exceptions in AArch32 modes at that Privilege level.

**Res** Means that the combination of HMC, SSC, and PMC is reserved. See *Reserved DBGBCR<n>.{HMC, SSC, PMC} values* on page G2-3915.

**Table G2-11 Summary of breakpoint HMC, SSC, and PMC encodings**

| HMC | SSC | PMC | Security state the breakpoint is programmed to match in | PL2[a] | PL1 | PL0 | Implementation | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | No EL3 | No EL2 and no EL3 |
| 0 | 00 | 00 | Both | - | Y[b] | Y[b] | - | - |
| 0 | 00 | 01 | | - | Y | - | - | - |
| 0 | 00 | 10 | | - | - | Y | - | - |
| 0 | 00 | 11 | | - | Y | Y | - | - |
| 0 | 01 | 00 | Non-secure | - | Y[b] | Y[b] | Res | Res |
| 0 | 01 | 01 | | - | Y | - | Res | Res |
| 0 | 01 | 10 | | - | - | Y | Res | Res |
| 0 | 01 | 11 | | - | Y | Y | Res | Res |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G2-11 Summary of breakpoint HMC, SSC, and PMC encodings (continued)**

| HMC | SSC | PMC | Security state the breakpoint is programmed to match in | PL2[a] | PL1 | PL0 | Implementation | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | No EL3 | No EL2 and no EL3 |
| 0 | 10 | 00 | Secure | - | Y[b] | Y[b] | Res | Res |
| 0 | 10 | 01 | | - | Y | - | Res | Res |
| 0 | 10 | 10 | | - | - | Y | Res | Res |
| 0 | 10 | 11 | | - | Y | Y | Res | Res |
| 1 | 00 | 01 | Both | Y | Y | - | - | Res |
| 1 | 00 | 11 | | Y | Y | Y | - | Res |
| 1 | 01 | 01 | Non-secure | Y | Y | - | Res | Res |
| 1 | 01 | 11 | | Y | Y | Y | Res | Res |
| 1 | 10 | 01 | Secure | - | Y | - | Res | Res |
| 1 | 10 | 11 | | - | Y | Y | Res | Res |
| 1 | 11 | 00 | Non-secure | Y | - | - | - | Res |

a. Debug exceptions are not generated at PL2 using AArch32. This means that these combinations of HMC, SSC, and PMC are only relevant if breakpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PMC that generate Breakpoint exceptions at PL2 using AArch32.

b. Only in User, System and Supervisor modes.

All combinations of HMC, SSC, and PMC that this table does not show are reserved. See *Reserved HMC, SSC, and PMC combinations* on page G2-3915.

## G2.9.4 Breakpoint instruction address comparisons

Address comparisons are made for each instruction in the program flow. The following subsections describe the criteria for a successful address comparison, for:

- *Address Match breakpoints*.
- *Address Mismatch breakpoints* on page G2-3908.

### Address Match breakpoints

An address match comparison is successful if both:

- Bits [31:2] of the current instruction virtual address are equal to DBGBVR<n>[31:2].
- The word or halfword selected by DBGBCR<n>.BAS matches. That is, either:
  - DBGBCR<n>.BAS is programmed with `0b0011` or `0b1111`, and the instruction is at a word-aligned address.
  - DBGBCR<n>.BAS is programmed with `0b1100`, and the instruction is not at a word-aligned address.

  See *Specifying the halfword-aligned address that an Address breakpoint matches on* on page G2-3908.

**Note**

DBGBVR<n>[1:0] are RES0 and are ignored.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Address Mismatch breakpoints

An address mismatch comparison is successful if either:

- Bits [31:2] of the current instruction virtual address are not equal to DBGBVR<n>[31:2].
- The word or halfword selected by DBGBCR<n>.BAS does not match. That is, either:
  - DBGBCR<n>.BAS is programmed with `0b0011` or `0b1111`, and the instruction is not at a word-aligned address.
  - DBGBCR<n>.BAS is programmed with `0b1100`, and the instruction is at a word-aligned address.

  See *Specifying the halfword-aligned address that an Address breakpoint matches on*.

**Note**

- DBGBVR<n>[1:0] are RES0 and are ignored.
- Address Mismatch breakpoints can be used to single-step through code. See *Using an Address Mismatch breakpoint to single-step an instruction* on page G2-3913.

### Specifying the halfword-aligned address that an Address breakpoint matches on

For an Address breakpoint, a debugger can use the *Byte Address Selection* field, DBGBCR<n>.BAS, so that the address comparison is successful on one of:

- The whole word starting at address DBGBVR<n>[31:2]:00.
- The halfword starting at address DBGBVR<n>[31:2]:00.
- The halfword starting at address ((DBGBVR<n>[31:2]:00) + 2).

This makes it possible to breakpoint on T32 instructions.

**Note**

The address programmed into the DBGBVR<n> must be word-aligned.

DBGBCR<n>.BAS can be used in both Address Match breakpoints and Address Mismatch breakpoints, as follows:

- For an Address Match breakpoint, DBGBCR<n>.BAS selects which halfword-aligned address the breakpoint must generate a Breakpoint exception for. This means that an address comparison is successful only if both of the following match:
  - The instruction address held in bits [31:2] of the DBGBVR<n>.
  - The halfword defined by the BAS field.

  That is, a successful address comparison = DBGBVR<n>[31:2]match AND BAS match.
- For an Address Mismatch breakpoint, DBGBCR<n>.BAS selects which halfword-aligned address the breakpoint must not generate a Breakpoint exception for. This means that an address comparison is successful if either or both of the following do not match:
  - The instruction address held in bits [31:2] of the DBGBVR<n>.
  - The halfword defined by the BAS field.

  That is, a successful address comparison = NOT (DBGBVR<n>[31:2] match AND BAS match).

The following subsections show the supported BAS values:

- *Using the BAS field in Address Match breakpoints* on page G2-3909.
- *Using the BAS field in Address Mismatch breakpoints* on page G2-3910.

For Context breakpoints, DBGBCR<n>.BAS is RES1 and is ignored.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Using the BAS field in Address Match breakpoints

The supported BAS values are:

`0b0000` This value is reserved. Behavior is a CONSTRAINED UNPREDICTABLE choice of:

- The breakpoint is disabled.
- The breakpoint behaves as if BAS is `0b0011`, `0b1100`, or `0b1111`.

`0b0011` The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for all of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, ARM recommends that a debugger uses this BAS value only for T32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

`0b1100` The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b10`.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for both of the following:

- 32-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
- 16-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 or A32 instruction starting at a word-aligned address.

`0b1111` The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for all of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, ARM recommends that a debugger uses this BAS value only for A32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on a 32-bit T32 instruction or a 16-bit T32 instruction at the halfword-aligned address ((DBGBVR<n>[31:2]:00) + 2).

All other BAS values are reserved. For these reserved other values, DBGBCR<n>.BAS[3,1] ignore writes and read the same values as DBGBCR<n>[2,0] respectively. This means that the smallest instruction size a debugger can program breakpoints to match on is a halfword.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Figure G2-3 shows a summary of when breakpoints programmed with particular BAS values generate Breakpoint exceptions.

The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example:

- Row 2 shows that a breakpoint with a BAS value of `0b1100` generates Breakpoint exceptions for 16-bit T32 instructions starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) + 2).
- Row 6 shows that a breakpoint with a BAS value of either `0b0011` or `0b1111` generates Breakpoint exceptions for A32 instructions. A32 instructions are always at word-aligned addresses.

In the figure:

**Yes** Means that the breakpoint generates a Breakpoint exception.

**No** Means that the breakpoint does not generate a Breakpoint exception.

**UNP** Means that is it CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See *Other usage constraints for Address breakpoints* on page G2-3916.

| | Instruction set | Size | Location of instruction[a] -2 | -1 | 0 | +1 | +2 | +3 | +4 | +5 | BAS[3:0] 0b0011 | 0b1100 | 0b1111 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Row 1 | T32 | 16-bit | | | ■ | ■ | | | | | Yes | No | Yes |
| Row 2 | | 16-bit | | | | | ■ | ■ | | | No | Yes | UNP |
| Row 3 | T32 | 32-bit | ■ | ■ | ■ | ■ | | | | | UNP | No | UNP |
| Row 4 | | 32-bit | | | ■ | ■ | ■ | ■ | | | Yes | UNP | Yes |
| Row 5 | | 32-bit | | | | | ■ | ■ | ■ | ■ | No | Yes | UNP |
| Row 6 | A32 | 32-bit | | | ■ | ■ | ■ | ■ | | | Yes | UNP | Yes |

a. 0 means the word-aligned address held in the DBGBVRn. The other locations are as follows:
- -2 means ((DBGBVRn[31:2]:00) - 2).
- -1 means ((DBGBVRn[31:2]:00) - 1).
- ...
- ...
- +5 means ((DBGBVRn[31:2]:00) + 5).

The solid areas show the location of the instruction.

**Figure G2-3 Summary of BAS field meanings for Address Match breakpoints**

### *Using the BAS field in Address Mismatch breakpoints*

An Address Mismatch breakpoint generates Breakpoint exceptions for all instructions committed for execution, except the instruction whose address the breakpoint is programmed to match.

The supported BAS values are:

`0b0000` The breakpoint ignores the address held in the DBGBVR<n> and generates Breakpoint exceptions for all instruction addresses.

`0b0011` The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:
- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for any of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, ARM recommends that a debugger uses this BAS value only for T32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

`0b1100` The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b10`.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for either of the following:

- 32-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
- 16-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 or A32 instruction at a word-aligned address.

`0b1111` The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<n>[31:2].
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for any of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, ARM recommends that a debugger uses this BAS value only for A32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<n>[31:2]:00) - 2).

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on a 32-bit T32 instruction or a 16-bit T32 instruction at the halfword-aligned address ((DBGBVR<n>[31:2]:00) + 2).

All other BAS values are reserved. For these reserved other values, DBGBCR<n>.BAS[3,1] ignore writes and read the same values as DBGBCR<n>[2,0] respectively. This means that the smallest instruction size that a breakpoint can never generate a Breakpoint exception for is a halfword.

Figure G2-4 on page G2-3912 shows a summary of when breakpoints programmed with particular BAS values generate Breakpoint exceptions.

The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example:

- Row 1 shows that a breakpoint with a BAS value of `0b1100` generates Breakpoint exceptions for 16-bit T32 instructions starting at the word-aligned address held in the DBGBVR<n>.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Row 5 shows that a breakpoint with a BAS value of `0b0011` generates Breakpoint exceptions for 32-bit T32 instructions starting at the halfword-aligned address immediately after the word aligned address held in the DBGBVR<n>.

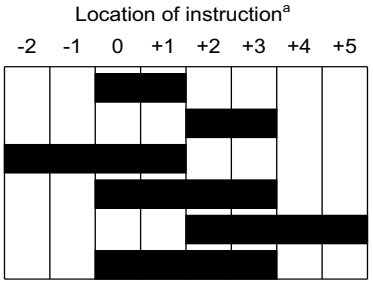
In the figure:

**Yes** Means that the breakpoint does generate a Breakpoint exception.

**No** Means that the breakpoint does not generate a Breakpoint exception.

**UNP** Means that is it CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See *Other usage constraints for Address breakpoints* on page G2-3916

.



| | Instruction set | Size | Location of instruction[a] | | | | | | | | BAS[3:0] | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | -2 | -1 | 0 | +1 | +2 | +3 | +4 | +5 | 0b0000 | 0b0011 | 0b1100 | 0b1111 |
| Row 1 | T32 | 16-bit | | | ■ | ■ | | | | | Yes | No | Yes | No |
| Row 2 | | 16-bit | | | | | ■ | ■ | | | Yes | Yes | No | UNP |
| Row 3 | T32 | 32-bit | ■ | ■ | ■ | ■ | | | | | Yes | UNP | Yes | UNP |
| Row 4 | | 32-bit | | | ■ | ■ | ■ | ■ | | | Yes | No | UNP | No |
| Row 5 | | 32-bit | | | | | ■ | ■ | ■ | ■ | Yes | Yes | No | UNP |
| Row 6 | A32 | 32-bit | | | ■ | ■ | ■ | ■ | | | Yes | No | UNP | No |

a. 0 means the word-aligned address held in the DBGBVRn. The other locations are as follows:
- -2 means ((DBGBVRn[31:2]:00) - 2).
- -1 means ((DBGBVRn[31:2]:00) - 1).
- ...
- ...
- +5 means ((DBGBVRn[31:2]:00) + 5).

The solid areas show the location of the instruction.

**Figure G2-4 Summary of BAS field meanings for Address Mismatch breakpoints**

## G2.9.5 Context comparisons

A context comparison is successful if, depending on the breakpoint type set by DBGBCR<n>.BT, one of the following is true:

- The current Context ID value is equal to DBGBVR<n>[31:0].
- The current VMID value is equal to DBGBXVR<n>[7:0].
- The current Context ID value is equal to DBGBVR<n>[31:0], and the current VMID value is equal to DBGBXVR<n>[7:0].

Context breakpoints do not generate Breakpoint exceptions when execution is in EL2.

The following Context breakpoint types do not generate Breakpoint exceptions in Secure state:
- VMID Match breakpoints.
- VMID and Context ID Match breakpoints.

**Note**

- For all Context breakpoints, DBGBCR<n>.BAS is RES1 and is ignored.
- For Linked Context breakpoints, DBGBCR<n>.{LBN, SSC, HMC, PMC} are RES0 and are ignored.

## G2.9.6 Using breakpoints

This section contains the following:
- *Using an Address Mismatch breakpoint to single-step an instruction* on page G2-3913.
- *ITD control effects on address breakpoints on the first instruction in an IT block* on page G2-3914.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Usage constraints* on page G2-3914.

### Using an Address Mismatch breakpoint to single-step an instruction

In execution conditions that an Address Mismatch breakpoint matches, defined by DBGBCR<n>.{LBN, SSC, PMC}, the breakpoint generates Breakpoint exceptions for all instructions committed for execution, except the instruction whose address the breakpoint is programmed with. Figure G2-5 shows an example of Address Mismatch breakpoint operation, for an Address Mismatch breakpoint programmed with address `0x1014`.

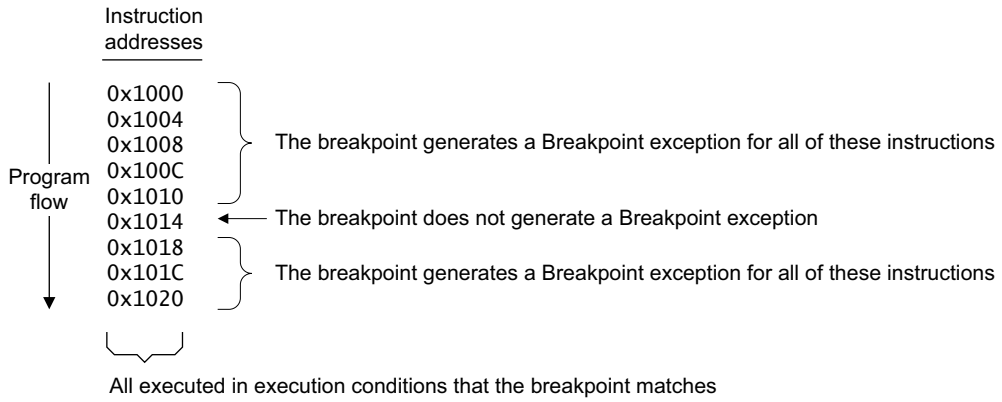



**Figure G2-5 Operation of an Address Mismatch breakpoint**

This means that an Address Mismatch breakpoint can be used to single-step an instruction.

In the example shown in Figure G2-5:

- If the target of a branch is an instruction other than the instruction at address `0x1014`, the breakpoint generates a Breakpoint exception when the instruction is committed for execution.
- If the target of a branch is the instruction at address `0x1014`, the PE executes the instruction at `0x1014` and the breakpoint does not generate a Breakpoint exception until the instruction at address `0x1018` is committed for execution. The instruction at address `0x1014` is therefore single-stepped.

  However, if the instruction at `0x1014` generates a synchronous exception, or if the PE takes an asynchronous exception while the instruction is being stepped, the breakpoint is evaluated again after taking the exception. This means that behavior is as follows:

  — If the exception handler executes in execution conditions that the breakpoint matches, the breakpoint generates a Breakpoint exception for the exception vector, because the exception vector is not address `0x1014`. This means that software execution steps into the exception.

  — If the exception handler executes in execution conditions that the breakpoint does not match, the breakpoint does not generate any Breakpoint exceptions after the PE has taken the exception, until the exception handler completes and executes an exception return instruction. The effect is to step over the exception. Whether the instruction is stepped again depends on whether the target of the exception return instruction is the instruction at `0x1014` or the instruction at `0x1018`.

  If the instruction at `0x1014` is single-stepped and branches to itself, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception after the PE has executed the branch.

This means that an instruction is only single-stepped if it is the target of a branch instruction and its address matches the address the breakpoint is programmed for. In the example shown in Figure G2-5, this is `0x1014`.

Usually this branch instruction is an exception return instruction that changes PE mode, branching from a PE mode in which the breakpoint does not generate a Breakpoint exception. A branch instruction that does not change PE mode would itself generate a Breakpoint exception. However, it might be a branch-to-self instruction as described above.

Because Address Mismatch breakpoints can single-step instructions, the behavior of an address mismatch Breakpoint exception is similar to the behavior of an AArch64 Software Step exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- The example shown in Figure G2-5 on page G2-3913 assumes an A32 instruction. The same behavior applies for both 32-bit and 16-bit T32 instructions.
- Software Step exceptions are the highest priority exception. Breakpoint exceptions are lower priority. See *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

### ITD control effects on address breakpoints on the first instruction in an IT block

In an implementation that supports the ITD control, if the value of the ITD field that applies to the current Exception level is 1, all of the following are true:

- An `IT` instruction can only be used to apply to one 16-bit T32 instruction.
- Only certain combinations of an `IT` instruction and second single 16-bit T32 instruction are permitted.
- For a permitted combination, it is IMPLEMENTATION DEFINED whether the implementation treats the combination as:
  - A pair of 16-bit instructions.
  - One 32-bit instruction.

If the implementation treats the combination as one 32-bit instruction, then as described in *Other usage constraints for Address breakpoints* on page G2-3916, an Address breakpoint might not generate a Breakpoint exception for an address match only on the second halfword of the instruction.

For this reason, if the ITD bit associated with the current Exception level is 1, ARM recommends that a debugger that wants to program a breakpoint to match on the second T32 instruction programs it to match on the `IT` instruction instead.

However, if returning from an exception whose preferred return address is the address of the second T32 instruction, then because the debugger is aware that the implementation has treated the combination as a pair of 16-bit instructions, the debugger is permitted to program the breakpoint to match on the second T32 instruction.

The ITD control fields are:

**HSCTLR.ITD** Applies to execution at EL2 when EL2 is using AArch32.

**SCTLR.ITD** Applies to execution at EL0 or EL1 when EL1 is using AArch32.

**SCTLR_EL1.ITD**
Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0, and therefore the information in this section does not apply to such an implementation.

**Note**

Programming the breakpoint to match on the second T32 instruction might be necessary when using an Address Mismatch breakpoint for single stepping.

### Usage constraints

See the following:

- *Reserved DBGBCR<n>.BT values* on page G2-3915.
- *Reserved DBGBCR<n>.{HMC, SSC, PMC} values* on page G2-3915.
- *Reserved DBGBCR<n>.BAS values* on page G2-3916.
- *Reserved DBGBCR<n>.LBN values* on page G2-3916.
- *Other usage constraints for Address breakpoints* on page G2-3916.
- *Other usage constraints for Context breakpoints* on page G2-3917.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Reserved DBGBCR<n>.BT values*

Table G2-12 shows when particular DBGBCR<n>.BT values are reserved.

**Table G2-12 Reserved BT values**

| BT value | Breakpoint type | Reserved |
|---|---|---|
| 0b001x | Context ID Match | For non context-aware breakpoints. |
| 0b010x | Address Mismatch | If EDSCR.HDE is 1 and halting is allowed. |
| 0b011x | - | Always. |
| 0b100x | VMID Match | For non context-aware breakpoints, or if EL2 is not implemented. |
| 0b101x | Context ID and VMID Match | |
| 0b11xx | - | Always. |

If an enabled breakpoint is programmed with one of these reserved BT values:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BT value that is not reserved, other than for a direct read of DBGBCR<n>.
- For a direct read of DBGBCR<n>, if the reserved BT value:
  - Has no function for any execution conditions, the value read back is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the BT value so that the breakpoint functions for the other execution conditions.

The behavior of breakpoints with reserved BT values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

### *Reserved DBGBCR<n>.{HMC, SSC, PMC} values*

Table G2-13 shows when particular combinations of DBGBCR<n>.{HMC, SSC, PMC} are reserved in stage 1 of an AArch32 translation regime.

**Table G2-13 Reserved HMC, SSC, and PMC combinations**

| HMC, SSC, and PMC combination | Reserved |
|---|---|
| All combinations with SSC set to 0b01 or 0b10. | When EL3 is not implemented and EL2 is implemented. |
| Any combination where HMC or SSC is nonzero. | When both of EL2 and EL3 are not implemented. |
| Combinations not included in Table G2-11 on page G2-3906. | Always |

For all breakpoints except Linked Context breakpoints, if an enabled breakpoint is programmed with one of these reserved combinations:

- If the reserved combination has a function for other execution conditions:
  - The breakpoint must behave as if it is disabled.
  - A direct read of DBGBCR<n>.{HMC, SSC, PMC} returns the values written. This means that software can save and restore the combination so that the breakpoint can function for the other execution conditions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If the reserved combination does not have a function for other execution conditions:
  - The breakpoint must behave either as if it is programmed with a combination that is not reserved or as if it is disabled.
  - A direct read of DBGBCR<n>.{HMC, SSC, PMC} returns UNKNOWN values.

Linked Context breakpoints ignore the values of HMC, SSC, and PMC. See *Other usage constraints for Context breakpoints* on page G2-3917.

The behavior of breakpoints with reserved combinations of HMC, SSC, and PMC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

#### *Reserved DBGBCR<n>.BAS values*

For all Context breakpoints, DBGBCR<n>.BAS is RES1 and is ignored.

For all Address breakpoints:

- The BAS field values `0bxx01` and `0b01xx` are reserved. A breakpoint programmed with `0bxx01` or `0b01xx` must behave as if it is programmed with `0bxx11` or `0b11xx` respectively.
- The BAS field values `0bxx10` and `0b10xx` are reserved. A breakpoint programmed with `0bxx10` or `0b10xx` must behave as if it is programmed with `0bxx00` or `0b00xx` respectively.
- The BAS field value `0b0000` is reserved. A breakpoint programmed with `0b0000` must behave either as if it is disabled, or programmed with `0b0011`, `0b1100`, or `0b1111`.

#### *Reserved DBGBCR<n>.LBN values*

A Linked Address breakpoint must link to a context-aware breakpoint. For a Linked Address breakpoint, any DBGBCR<n>.LBN value that is not for a context-aware breakpoint is reserved.

#### *Other usage constraints for Address breakpoints*

- For all Address breakpoints:
  - DBGBVR<n>[1:0] are RES0 and are ignored.
  - The DBGBXVR<n> is ignored.
- For Address Match breakpoints:
  - For 32-bit instructions, if a breakpoint matches on the address of the second halfword but not the address of the first halfword, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception.
  - If DBGBCR<n>.BAS is `0b1111`, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception for a T32 instruction starting at address ((DBGBVR<n>[31:2]:00) + 2). For T32 instructions, ARM recommends that the debugger programs the BAS field with either `0b0011` or `0b1100`.
- For Address Mismatch breakpoints:
  - The constraints are the same as those described in *For Address Match breakpoints*, except that if DBGBCR<n>.BAS is programmed with `0b0000`, the breakpoint matches on all addresses, even for the address held in the DBGBVR<n>.

  That is, if DBGBCR<n>.BAS is programmed with `0b0000`, the Address Mismatch breakpoint ignores the address held in the DBGBVR<n>.

For all Unlinked Address breakpoints, DBGBCR<n>.LBN reads UNKNOWN and its value is ignored.

For Linked Address breakpoints:

- If a Linked Address breakpoint links to a breakpoint that is not implemented, or that is not context-aware, behavior is CONSTRAINED UNPREDICTABLE. The Linked Address breakpoint behaves as if it is either:
  - Disabled, and DBGBCR<n>.LBN for it reads UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Links to an UNKNOWN context-aware breakpoint. In this case, it generates Breakpoint exceptions and DBGBCR<n>.LBN indicates which context-aware breakpoint it has linked to.

- If a Linked Address breakpoint that links to a breakpoint that is implemented and that is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

#### *Other usage constraints for Context breakpoints*

For all Context breakpoints:

- Any bits of DBGBVR<n> and DBGBXVR<n> that are not used to specify Context ID or VMID are RES0 and are ignored.

  **Note**

  This means that for Context ID Match breakpoints, the DBGBXVR<n> is RES0 and is ignored, and for VMID Match breakpoints, the DBGBVR<n> is RES0 and is ignored.

- DBGBCR<n>.LBN reads UNKNOWN and its value is ignored.

For Linked Context breakpoints:

- DBGBCR<n>.{LBN, SSC, HMC, PMC} are RES0 and are ignored.
- If no Linked Address breakpoints or Linked Watchpoints link to a Linked Context breakpoint, the Linked Context breakpoint does not generate any Breakpoint exceptions.

### G2.9.7 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information*.
- *Preferred return address* on page G2-3918.

#### Exception syndrome information

The PE takes a Breakpoint exception as either:

- A Prefetch Abort exception, if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

**Abort mode**

The PE sets all of the following:

- DBGDSCRext.MOE to `0b0001`, to indicate a Breakpoint exception.
- IFSR.FS to the code for a debug event, `0b00010`.
- The IFAR with an UNKNOWN value.

**Hyp mode**

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-14 on page G2-3918.
- Sets DBGDSCRext.MOE to `0b0001`, to indicate a Breakpoint exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Sets the HIFAR to an UNKNOWN value.

**Table G2-14 Information recorded in the HSR**

| HSR field | Information recorded |
|---|---|
| *Exception Class*, EC | The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, `0x20`. |
| *Instruction Length*, IL | The PE sets this to 1. |
| *Instruction Specific Syndrome*, ISS | **ISS[24:10]** RES0.<br>**ISS[9]** *External Abort type* (EA). The PE sets this to 0.<br>**ISS[8:6]** RES0.<br>**ISS[5:0]** *Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`. |

**Note**

For information about how debug exceptions can be routed to PL2, see *Routing debug exceptions on page G2-3887*.

### Preferred return address

The preferred return address of a Breakpoint exception is the address of the instruction that was not executed because the PE took the Breakpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

## G2.9.8 Pseudocode description of Breakpoint exceptions taken from AArch32 state

`AArch32.BreakpointValueMatch()` returns a pair of results:

- A result for Address Match and Context breakpoints.
- A result for Address Mismatch breakpoints.

```
// AArch32.BreakpointValueMatch()
// ==============================
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n, bits(32) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //    matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.

    // If a non-existant breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(DBGDIDR.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(DBGDIDR.BRPs));
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking.)
    if DBGBCR[n].E == '0' then return (FALSE,FALSE);

    context_aware = (n >= UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    type = DBGBCR[n].BT;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if (type IN {'011x','11xx'} ||                                  // Reserved
          (type == '010x' && HaltOnBreakpointOrWatchpoint()) ||     // Address mismatch
          (type != '0x0x' && !context_aware) ||                     // Context matching
          (type == '1xxx' && !HaveEL(EL2))) then                    // VMID match
        (c, type) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = type == '0x0x';
    mismatch   = type == '010x';
    match_vmid = type == '10xx';
    match_cid  = type == 'x01x';
    linked     = type == 'xxx1';

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return (FALSE,FALSE);

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return (FALSE,FALSE);

    // Do the comparison.
    if match_addr then
        byte = UInt(vaddress<1:0>);
        assert byte IN {0,2};                     // "vaddress" is halfword aligned.
        byte_select_match = (DBGBCR[n].BAS<byte> == '1');
        BVR_match = vaddress<31:2> == DBGBVR[n]<31:2> && byte_select_match;
    elsif match_cid then
        BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
    if match_vmid then
        vmid = (if ELUsingAArch32(EL2) then VTTBR.VMID else VTTBR_EL2.VMID);
        BXVR_match = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                      vmid == DBGBXVR[n]<7:0>);

    match = (!match_vmid || BXVR_match) && (!(match_addr || match_cid) || BVR_match);
    return (match && !mismatch, !match && mismatch);
```

`AArch32.StateMatch()` tests the values in DBGBCR<n>.{SSC, HMC, PMC} and, if the breakpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

For a watchpoint, `AArch32.StateMatch()` tests the values in DBGWCR<n>.{SSC, HMC, PAC} and, if the watchpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

```
// AArch32.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreakpnt, boolean ispriv)
    // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
    // "linked" is TRUE if this is a linked breakpoint/watchpoint type.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.

    // If parameters are set to a reserved type, behaves either as disabled or not a reserved-type
    if ((HMC:SSC:PxC) IN {'011xx','100x0','101x0','110x0','11101','1111x'} ||         // Reserved
          (HMC == '0' && PxC == '00' && !isbreakpnt) ||                               // Usr/Svc/Sys
          (SSC IN {'01','10'} && !HaveEL(EL3)) ||                                     // No EL3
          (HMC:SSC != '000' && !HaveEL(EL3) && !HaveEL(EL2))) then                    // No EL3/EL2
        (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    PL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
    PL2_match = HaveEL(EL2) && HMC == '1';
    PL1_match = PxC<0> == '1';
    PL0_match = PxC<1> == '1';
    SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';

    if SSU_match then
        priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
    else
        case PSTATE.EL of
            when EL3, EL1  priv_match = if ispriv then PL1_match else PL0_match;
            when EL2       priv_match = PL2_match;
            when EL0       priv_match = PL0_match;

    case SSC of
        when '00'  security_state_match = TRUE;         // Both
        when '01'  security_state_match = !IsSecure();  // Non-secure only
        when '10'  security_state_match = IsSecure();   // Secure only
        when '11'  security_state_match = TRUE;         // Both

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));
        last_ctx_cmp = UInt(DBGDIDR.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(32) UNKNOWN;
        linked_to = TRUE;
        (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

`AArch32.BreakpointMatch()` tests a committed instruction against all breakpoints.

```
// AArch32.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIDR.BRPs);

    enabled = DBGBCR[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then                 // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool();
        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool();

    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool();
        if !value_mismatch then value_mismatch = ConstrainUnpredictableBool();

    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
```

`AArch32.CheckBreakpoint()` generates a FaultRecord that `AArch32.Abort` raises a Breakpoint exception for if all of the following are true:

- DBGDSCRext.MDBGen is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-3889.
- All of the conditions required for Breakpoint exception generation are met. See *About Breakpoint exceptions* on page G2-3898.

**Note**

`AArch32.CheckBreakpoint()` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

```
// AArch32.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

    match = FALSE;
    mismatch = FALSE;

    for i = 0 to UInt(DBGDIDR.BRPs)
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_Breakpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The `Abort()` function processes the Faultrecord object returned by `CheckBreakpoint`, as described in *Abort exceptions on page G3-3985*. If there is a Breakpoint exception, the `Abort()` function generates a Prefetch Abort exception.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G2.10 Watchpoint exceptions

This section describes Watchpoint exceptions in stage 1 of an AArch32 translation regime.

It contains the following subsections:

- *About Watchpoint exceptions*.
- *Watchpoint types and linking of watchpoints* on page G2-3924.
- *Execution conditions for which a watchpoint generates Watchpoint exceptions* on page G2-3925.
- *Watchpoint data address comparisons* on page G2-3926.
- *Determining the memory location that caused a Watchpoint exception* on page G2-3930.
- *Watchpoint behavior on other instructions* on page G2-3931.
- *Usage constraints* on page G2-3931.
- *Exception syndrome information and preferred return address* on page G2-3933.
- *Pseudocode description of Watchpoint exceptions taken from AArch32 state* on page G2-3934.

### G2.10.1 About Watchpoint exceptions

A *watchpoint* is a debug event that results from the execution of an instruction, based on a data address. Watchpoints are also known as *data breakpoints*.

A watchpoint operates as follows:

1. A debugger programs the watchpoint with a data address, or a data address range.
2. The watchpoint generates a *Watchpoint debug event* on an access to the address, or any address in the address range.

A watchpoint never generates a Watchpoint debug event on an instruction fetch.

An implementation can include between 2-16 watchpoints. In an implementation, DBGDIDR.WRPs shows how many are implemented.

To use an implemented watchpoint, a debugger programs the following registers for the watchpoint:

- The *Watchpoint Control Register*, DBGWCR<n>. This holds control information for the watchpoint, for example an enable control.
- The *Watchpoint Value Register*, DBGWVR<n>. This holds the data virtual address used for watchpoint matching.

The registers are numbered, so that:

- DBGWCR1 and DBGWVR1 are for watchpoint number one.
- DBGWCR2 and DBGWVR2 are for watchpoint number two.
- …
- …
- DBGWCRn and DBGWVRn are for watchpoint number n.

A watchpoint can:

- Be programmed to generate Watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
- Link to a *Linked Context breakpoint*, so that a Watchpoint debug event is only generated if the PE is in a particular context when the address match occurs.

A single watchpoint can be programmed to match on one or more address bytes. A watchpoint generates a Watchpoint debug event on an access to any byte that it is watching. The number of bytes a watchpoint is watching is either:

- One to eight bytes, provided that these bytes are contiguous and that they are all in the same naturally-aligned doubleword. A debugger uses the *Byte Address Select* field, DBGWCR<n>.BAS, to select the bytes. See *Programming a watchpoint with eight bytes or fewer* on page G2-3927.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Eight bytes to 2GB, provided that both of the following are true:
  - The number of bytes is a power-of-two.
  - The range starts at an address that is aligned to the range size.

  A debugger uses the *MASK* field, DBGWCR<n>.MASK, to program a watchpoint with eight bytes to 2GB. See *Programming a watchpoint with eight or more bytes* on page G2-3929.

A debugger must use either the BAS field or the MASK field. If it uses both, whether the watchpoint generates Watchpoint exceptions is CONSTRAINED UNPREDICTABLE. See *Programming dependencies of the BAS and MASK fields* on page G2-3932.

For each memory access, all of the watchpoints are tested. When a watchpoint is tested, it generates a Watchpoint debug event if all of the following are true:

- The watchpoint is enabled. That is, the watchpoint enable control for it, DBGWCR<n>.E, is 1.
- The conditions specified in the DBGWCR<n> are met.
- The comparison with the address held in the DBGWVR<n> is successful.
- If the watchpoint links to a Linked Context breakpoint, the comparison or comparisons made by the Linked Context breakpoint are successful. See  on page G2-3902 shows this. See also *Context comparisons* on page G2-3912.
- The instruction that initiates the memory access is committed for execution.
- The instruction that initiates the memory access passes its condition code check.

If halting is allowed and EDSCR.HDE is 1, Watchpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:
- Enabled, Watchpoint debug events generate Watchpoint exceptions.
- Disabled, Watchpoint debug events are ignored.

**Note**

The remainder of this Watchpoint Exceptions section, including all subsections, describes watchpoints as generating Watchpoint exceptions.

However, the behavior described also applies if watchpoints are causing entry to Debug state.

*The debug exception enable controls* on page G2-3886 describes the enable controls for Watchpoint debug events.

### G2.10.2 Watchpoint types and linking of watchpoints

When a debugger programs a watchpoint, it must program that watchpoint so that it is either:
- Used in isolation. In this case the watchpoint is called an *Unlinked watchpoint*.
- Enabled for linking to a Linked Context breakpoint. In this case the watchpoint is called a *Linked watchpoint*.

When a Linked watchpoint links to a Linked Context breakpoint, the Linked watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs. For example, a debugger might:

1. Program watchpoint number one with a data address.
2. Program breakpoint number five to be a *Linked VMID Match breakpoint*.
3. Link the watchpoint and the breakpoint together. A Watchpoint exception is only generated if both the data address matches and the VMID matches.

The *Watchpoint Type* field for a watchpoint, DBGWCR<n>.WT, controls whether the watchpoint is enabled for linking. If DBGWCR<n>.WT is 1, the watchpoint is enabled for linking.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Rules for linking watchpoints

The rules for watchpoint linking are as follows:

- Only Linked watchpoints can be linked.
- A Linked watchpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, DBGWCR<n>.LBN, for the Linked watchpoint specifies the particular Linked Context breakpoint that the Linked watchpoint links to, and:
  - DBGWCR<n>.WT.{SSC, HMC, PAC} for the Linked watchpoint define the execution conditions that the watchpoint generates Watchpoint exceptions for. See *Execution conditions for which a watchpoint generates Watchpoint exceptions*.
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- A Linked watchpoint cannot link to another watchpoint. The LBN field can therefore only specify a breakpoint.
- If a Linked watchpoint links to a breakpoint that is not context-aware, the behavior of the Linked watchpoint is CONSTRAINED UNPREDICTABLE. See *Usage constraints* on page G2-3931
- If a Linked watchpoint links to an Unlinked Context breakpoint, the Linked watchpoint never generates any Watchpoint exceptions.
- Multiple Linked watchpoints can link to a single Linked Context breakpoint.

**Note**

Multiple Address breakpoints can also link to a single Linked Context breakpoint. *Breakpoint exceptions* on page G2-3898 describes breakpoints.

Figure G2-2 on page G2-3902 shows an example of permitted watchpoint linking.

## G2.10.3 Execution conditions for which a watchpoint generates Watchpoint exceptions

Each watchpoint can be programmed so that it only generates Watchpoint exceptions for certain execution conditions. For example, a watchpoint might be programmed to generate Watchpoint exceptions only when the PE is executing at PL0 in Secure state.

DBGWCR<n>.{SSC, HMC, PAC} define the execution conditions a watchpoint generates Watchpoint exceptions for, as follows:

**Security State Control, SSC**

Controls whether the watchpoint generates Watchpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

**Note**

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the watchpoint is set.

**Higher Mode Control, HMC, and Privileged Access Control, PAC**

HMC and PAC together control which Privilege level the watchpoint generates Watchpoint exceptions in.

**Note**

PAC controls which access privilege the watchpoint matches. This means that if the PE is executing an unprivileged load/store instruction at PL1, the data access might trigger a watchpoint that is programmed to match on PL0 accesses.

Table G2-15 on page G2-3926 shows the valid combinations of HMC, SSC, and PAC, and for each combination shows which Privilege levels watchpoints generate Watchpoint exceptions in.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In the table:

**Y or -** Means that a watchpoint programmed with the values of HMC, SSC, and PAC shown in that row:

**Y** Can generate Watchpoint exceptions at that Privilege level.

**-** Cannot generate Watchpoint exceptions at that Privilege level.

**Res** Means that the combination of HMC, SSC, and PAC is reserved. See *Reserved DBGWCR<n>.{HMC, SSC, PAC} values* on page G2-3932.

**Table G2-15 Summary of watchpoint HMC, SSC, and PAC encodings**

| HMC | SSC | PAC | Security state the watchpoint is programmed to match in | PL2[a] | PL1 | PL0 | Implementation | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | No EL3 | No EL2 and no EL3 |
| 0 | 00 | 01 | Both | - | Y | - | - | - |
| 0 | 00 | 10 | | - | - | Y | - | - |
| 0 | 00 | 11 | | - | Y | Y | - | - |
| 0 | 01 | 01 | Non-secure | - | Y | - | Res | Res |
| 0 | 01 | 10 | | - | - | Y | Res | Res |
| 0 | 01 | 11 | | - | Y | Y | Res | Res |
| 0 | 10 | 01 | Secure | - | Y | - | Res | Res |
| 0 | 10 | 10 | | - | - | Y | Res | Res |
| 0 | 10 | 11 | | - | Y | Y | Res | Res |
| 1 | 00 | 01 | Both | Y | Y | - | - | Res |
| 1 | 00 | 11 | | Y | Y | Y | - | Res |
| 1 | 01 | 01 | Non-secure | Y | Y | - | Res | Res |
| 1 | 01 | 11 | | Y | Y | Y | Res | Res |
| 1 | 10 | 01 | Secure | - | Y | - | Res | Res |
| 1 | 10 | 11 | | - | Y | Y | Res | Res |
| 1 | 11 | 00 | Non-secure | Y | - | - | - | Res |

a. Debug exceptions are not generated at PL2 using AArch32. This means that these combinations of HMC, SSC, and PAC are only relevant if watchpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PAC that generate Watchpoint exceptions at PL2 using AArch32.

All combinations of HMC, SSC, and PAC that this table does not show are reserved. See *Reserved DBGWCR<n>.{HMC, SSC, PAC} values* on page G2-3932.

### G2.10.4 Watchpoint data address comparisons

An address comparison is successful if bits [31:2] of the current data virtual address are equal to DBGWVR<n>[31:2], taking into account all of the following:

- The size of the access. See *Size of the data access* on page G2-3927.
- The bytes selected by DBGWVR<n>.BAS. See *Programming a watchpoint with eight bytes or fewer* on page G2-3927.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Any address ranges indicated by DBGWVR<n>.MASK. See *Programming a watchpoint with eight or more bytes* on page G2-3929.

**Note**

DBGWVR<n>[1:0] are RES0 and are ignored.

### Size of the data access

Because watchpoints can be programmed to generate Watchpoint exceptions on individual bytes, the size of each access must be taken into account. See Example G2-1.

**Example G2-1**

1. A debugger programs a watchpoint to generate Watchpoint exceptions only when the byte at address `0x1009` is accessed.
2. The PE accesses the unaligned doubleword starting at address `0x1003`.

In this scenario, the watchpoint must generate a Watchpoint exception.

The size of data accesses initiated by `DCIMVAC` instructions is an IMPLEMENTATION DEFINED size that is both:

- From the inclusive range between:
  - The size that CTR.DminLine defines.
  - 2KB.
- A power-of-two.

The lowest address accessed by a `DCIMVAC` instruction is the address supplied to the instruction, rounded down to the nearest multiple of the access size initiated by that instruction.

The highest address accessed is (size - 1) bytes above the lowest address accessed.

See also, *Watchpoint behavior on accesses by DCIMVAC instructions* on page G2-3931.

### Programming a watchpoint with eight bytes or fewer

The Byte Address Select field, DBGWCR<n>.BAS, selects which bytes in the doubleword starting at the address contained in the DBGWVR<n> the watchpoint generates Watchpoint exceptions for.

If the address programmed into the DBGWVR<n> is:

- Doubleword-aligned:
  - All eight bits of DBGWCR<n>.BAS are used, and the descriptions given in Table G2-16 apply.
- Word-aligned but not doubleword-aligned:
  - Only DBGWCR<n>.BAS[3:0] are used, and the descriptions given in Table G2-17 on page G2-3928 apply. In this case, DBGWCR<n>.BAS[7:4] are RES0.

**Table G2-16 Supported BAS values when the DBGWVRn address alignment is doubleword**

| BAS value | Description |
|---|---|
| `0b00000000` | Watchpoint never generates a Watchpoint exception |
| BAS[0] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:000 is accessed |
| BAS[1] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:001 is accessed |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G2-16 Supported BAS values when the DBGWVRn address alignment is doubleword**

| BAS value | Description |
|---|---|
| BAS[2] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:010 is accessed |
| BAS[3] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:011 is accessed |
| BAS[4] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:100 is accessed |
| BAS[5] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:101 is accessed |
| BAS[6] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:110 is accessed |
| BAS[7] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:3]:111 is accessed |

**Table G2-17 Supported BAS values when the DBGWVRn address alignment is word**

| BAS value[a] | Description |
|---|---|
| 0b00000000 | Watchpoint never generates a Watchpoint exception |
| BAS[0] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:2]:00 is accessed |
| BAS[1] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:2]:01 is accessed |
| BAS[2] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:2]:10 is accessed |
| BAS[3] == 1 | Generates a Watchpoint exception if byte at address DBGWVR<n>[31:2]:11 is accessed |

a. DBGWCR<n>.BAS[7:4] are RES0.

If the BAS field is programmed with more than one byte, the bytes that it is programmed with must be contiguous. For watchpoint behavior when its BAS field is programmed with non-contiguous bytes, see *Other usage constraints* on page G2-3933.

When programming the BAS field with anything other than 0b11111111, a debugger must also program DBGWCR<n>.MASK to be 0b00000. See *Programming dependencies of the BAS and MASK fields* on page G2-3932.

A watchpoint generates a Watchpoint exception whenever a watched byte is accessed, even if:

- The access size is smaller or larger than the address region being watched.
- The access is misaligned, and the base address of the access is not in the doubleword or word of memory addressed by the DBGWVR<n>[31:3]. See Example G2-1 on page G2-3927.

The following are some example configurations of the BAS field:

- To program a watchpoint to generate a Watchpoint exception on the byte at address 0x1003, program:
  - DBGWVR<n> with 0x1000.
  - DBGWCR<n>_EL1.BAS to be 0b00001000.
- To program a watchpoint to generate a Watchpoint exception on the bytes at addresses 0x2003, 0x2004 and 0x2005, program:
  - DBGWVR<n> with 0x2000.
  - DBGWCR<n>_EL1.BAS to be 0b00111000.
- If the address programmed into the DBGWVR<n> is doubleword-aligned:
  - To generate a Watchpoint exception when any byte in the word starting at the doubleword-aligned address is accessed, program DBGWCR<n>.BAS to be 0b00001111.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— To generate a Watchpoint exception when any byte in the word starting at address DBGWVR<n>[31:3]:100 is accessed, program DBGWCR<n>.BAS to be 0b11110000.

**Note**

ARM deprecates programming a DBGWVR<n> with an address that is not doubleword-aligned.

### Programming a watchpoint with eight or more bytes

A debugger can use the *MASK* field, DBGWCR<n>.MASK, to program a single watchpoint with a data address range. The data address range must meet all of the following criteria:

- It is a size that is both:
  - — A power-of-two.
  - — A minimum of eight bytes.
  - — A maximum of 2GB.
- It starts at an address that is aligned to the size.

The MASK field specifies the number of least significant data address bits that must be masked. Up to 31 least significant bits can be masked:

| | | |
|---|---|---|
| **MASK** | 0b00000 | No bits are masked. |
| | 0b00001 | Reserved. |
| | 0b00010 | Reserved. |
| | 0b00011 | Three least significant bits are masked. |
| | 0b00100 | Four least significant bits are masked. |
| | 0b00101 | Five least significant bits are masked. |
| | … | … |
| | 0b11111 | 31 least significant bits are masked. |

If *n* least significant address bits are masked, the watchpoint generates a Watchpoint exception on all of the following:

- Address DBGWVR<n>[31:*n*]:000…
- Address DBGWVR<n>[31:*n*]:111…
- Any address between these two addresses.

For example, if the four least significant address bits are masked, Watchpoint exceptions are generated for all addresses between DBGWVR<n>[31:4]:0000 and DBGWVR<n>[31:4]:1111, including these addresses.

**Note**

- The most significant bit cannot be masked. This means that the full address cannot be masked.
- For watchpoint behavior when its MASK field is programmed with a reserved value, see *Reserved DBGWCR<n>.MASK values* on page G2-3933.

When masking address bits, a debugger must both:

- Program DBGWCR<n>.BAS to be 0b11111111. See *Programming dependencies of the BAS and MASK fields* on page G2-3932.
- In the DBGWVR<n>, set the masked address bits to 0. For watchpoint behavior when any of the masked address bits are not 0, see *Other usage constraints* on page G2-3933.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G2.10.5 Determining the memory location that caused a Watchpoint exception

On a Watchpoint exception, the PE records an address in a *Fault Address Register* that the debugger can use to determine the memory location that triggered the watchpoint.

The Fault Address Register (FAR) used is either:

- DFAR, if the exception is taken to PL1.
- HDFAR, if the exception is taken to PL2.

In cases where one instruction triggers multiple watchpoints, only one address is recorded.

On entering Debug state on a Watchpoint debug event, the PE records the address in the EDWAR.

For more information, see the subsections that follow. These are:

- *Address recorded for Watchpoint exceptions generated by instructions other than Data Cache instructions*.
- *Address recorded for Watchpoint exceptions generated by Data Cache instructions*.

#### Address recorded for Watchpoint exceptions generated by instructions other than Data Cache instructions

The address recorded must be both:

- From the inclusive range between:
  - The lowest address accessed by the memory access that triggered the watchpoint.
  - The highest *watchpointed address* accessed by the memory access. A watchpointed address is an address that the watchpoint is watching.
- Within a naturally-aligned block of memory that is all of the following:
  - A power-of-two size.
  - No larger than 2KB.

    **Note**

    It must also be no larger than the block size used by the A64 DC ZVA instruction.
    There is no architectural means to discover this block size from AArch32 state.

  - Contains a watchpointed address accessed by the memory access.

  The size of the block is IMPLEMENTATION DEFINED. There is no architectural means of discovering the size.

**Example G2-2 Address recorded for a watchpoint programmed on `0x8019`**

A debugger programs a watchpoint to generate a Watchpoint debug event on any access to the byte `0x8019`.

An A32 load multiple instruction then loads nine registers starting from address `0x8004` upwards. This triggers the watchpoint.

If the `DC ZVA` block size is:

- 32 bytes, the address that the PE records must be between `0x8004` and `0x8019` inclusive.
- 16 bytes, the address that the PE records must be between `0x8010` and `0x8019` inclusive.

#### Address recorded for Watchpoint exceptions generated by Data Cache instructions

The address recorded is the address passed to the instruction. This means that the address recorded might be higher than the address of the location that triggered the watchpoint.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G2.10.6 Watchpoint behavior on other instructions

Under normal operating conditions, the following do not generate Watchpoint exceptions:

- Instruction cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.
- Preload instructions.
- All data cache maintenance instructions except DCIMVAC.

However, the debug architecture allows for IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to enable checking for Watchpoint debug events on an IMPLEMENTATION DEFINED subset of these instructions. Whether a watchpoint treats the instruction as a load or a store, and the access size of instruction cache, address translation, and TLB operations are IMPLEMENTATION DEFINED.

See also:

- *Watchpoint behavior on accesses by Store-Exclusive instructions*.
- *Watchpoint behavior on accesses by DCIMVAC instructions*.

#### Watchpoint behavior on accesses by Store-Exclusive instructions

If a watchpoint matches on a data access caused by a Store-Exclusive instruction, then:

- If the store fails because an exclusive monitor does not permit it, it is IMPLEMENTATION DEFINED whether the watchpoint generates a Watchpoint exception.
- Otherwise, the watchpoint generates a Watchpoint exception.

#### Watchpoint behavior on accesses by DCIMVAC instructions

It is IMPLEMENTATION DEFINED whether DCIMVAC operations can generate Watchpoint exceptions. If they can, they are treated as data stores. This means that for a watchpoint to match on an access caused by a DCIMVAC instruction, the debugger must program DBGWCR<n>.LSC to be one of the following:

**10** Match on data stores only.

**11** Match on data stores and data loads.

**Note**

For the size of data accesses performed by DCIMVAC instructions, see *Watchpoint data address comparisons* on page G2-3926. The size of all data accesses must be considered because watchpoints can be programmed to match on individual bytes.

### G2.10.7 Usage constraints

See the following:

- *Reserved DBGWCR<n>.{HMC, SSC, PAC} values* on page G2-3932.
- *Reserved DBGWCR<n>.LBN values* on page G2-3932.
- *Programming dependencies of the BAS and MASK fields* on page G2-3932.
- *Reserved DBGWCR<n>.BAS values* on page G2-3933.
- *Reserved DBGWCR<n>.MASK values* on page G2-3933.
- *Other usage constraints* on page G2-3933.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Reserved DBGWCR<n>.{HMC, SSC, PAC} values

Table G2-18 shows when particular combinations of DBGWCR<n>.{HMC, SSC, PAC} are reserved.

**Table G2-18 Reserved HMC, SSC, and PAC combinations**

| HMC, SSC, and PMC combination | Reserved |
|---|---|
| All combinations with SSC set to `0b01` or `0b10`. | When EL3 is not implemented and EL2 is implemented. |
| All combinations where HMC or SSC is nonzero, and all combinations with PAC set to `0b00`. | When both of EL2 and EL3 are not implemented. |
| Combinations not included in Table G2-15 on page G2-3926. | Always |

If an enabled watchpoint is programmed with one of these reserved combinations:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with a combination that is not reserved, other than for a direct read of DBGWCR<n>.
- For a direct read of DBGWCR<n>, if the reserved combination:
  - Has no function for any execution conditions, the value read back for each of HMC, SSC, and PMC is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the combination so that the watchpoint functions for the other execution conditions.

The behavior of watchpoints with reserved combinations of HMC, SSC, and PAC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

### Reserved DBGWCR<n>.LBN values

A Linked watchpoint must link to a context-aware breakpoint. For a Linked watchpoint, any DBGWCR<n>.LBN value that is not for a context-aware breakpoint is reserved.

### Programming dependencies of the BAS and MASK fields

When programming a watchpoint, a debugger must use either:

- The MASK field, to program the watchpoint with an address range that can be eight bytes to 2GB.
- The BAS field, to select which bytes in the doubleword or word starting at the address contained in the DBGWVR<n> the watchpoint must generate Watchpoint exceptions for.

If the debugger uses the:

- MASK field, it must program BAS to be `0b11111111`, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be `0b00000`, so that the MASK field does not indicate any address ranges.

If the debugger uses both of these fields, then behavior of the watchpoint is CONSTRAINED UNPREDICTABLE. Either:

- The watchpoint treats the MASK field as if it is programmed with `0b00000`. In this case, the watchpoint is programmed with a single address and it generates Watchpoint exceptions for the bytes that the BAS field indicates.
- For each byte in the masked region, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Reserved DBGWCR<n>.BAS values

If DBGWVR<n>[2] is 1, DBGWCR<n>.BAS[7:4] are RES0 and are ignored.

### Reserved DBGWCR<n>.MASK values

If DBGWCR<n>.MASK is programmed with a reserved value, the watchpoint must behave as if it is either:

- Disabled.
- Programmed with an UNKNOWN value that is not reserved, that might be `0b00000`.

### Other usage constraints

For all watchpoints:

- DBGWVR<n>[1:0] are RES0 and are ignored.
- If DBGWCR<n>.BAS is programmed with non-contiguous bytes of memory, it is CONSTRAINED UNPREDICTABLE whether the Watchpoint generates a Watchpoint exception for each byte in the doubleword or word of memory addressed by the DBGWCR<n>.
- If DBGWVR<n>.MASK is nonzero, and any masked bits of DBGWVR<n> are not 0, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception when the unmasked bits match.
- A watchpoint never generates any Watchpoint exceptions if DBGWCR<n>.LSC is `0b00`.

For Unlinked watchpoints, DBGBCR<n>.LBN reads UNKNOWN and its value is ignored.

For Linked watchpoints:

- If a Linked watchpoint links to a breakpoint that is not implemented, or that is not context-aware, behavior is CONSTRAINED UNPREDICTABLE. The Linked watchpoint behaves as if it is either:
  - Disabled, and DBGBCR<n>.LBN for it reads UNKNOWN.
  - Links to an UNKNOWN context-aware breakpoint. In this case, it generates Watchpoint exceptions and DBGBCR<n>.LBN indicates which context-aware breakpoint it has linked to.
- If a Linked watchpoint links to a breakpoint that is implemented and is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

## G2.10.8 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information*.
- *Preferred return address* on page G2-3934.

### Exception syndrome information

The PE takes a Watchpoint exception as either:

- A Data Abort exception, if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

**Abort mode**

The PE sets all of the following:

- DBGDSCRext.MOE to `0b1010`, to indicate a Watchpoint exception.
- DFSR.CM to indicate whether a cache maintenance instruction caused the exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- DFSR.WnR to indicate whether the exception was generated on a read instruction or a write instruction.
- DFAR to an address that the debugger can use to determine the memory location that triggered the watchpoint. See *Determining the memory location that caused a Watchpoint exception* on page G2-3930.

In addition, if using the:

- Short-descriptor format, the PE sets DFSR.FS to the code for a debug event, `0b00010`, and DFSR.Domain to an UNKNOWN value.
- Long-descriptor format, the PE sets DFSR.STATUS to the code for a debug event, `0b100010`.

**Hyp mode**

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-19.
- Sets DBGDSCRext.MOE to `0b1001`, to indicate a Watchpoint exception.
- Sets the HDFAR to an address that the debugger can use to determine the memory location that triggered the watchpoint. See *Determining the memory location that caused a Watchpoint exception* on page G2-3930.

**Table G2-19 Information recorded in the HSR**

| HSR field | Information recorded |
|---|---|
| *Exception Class*, EC | The PE sets this to the code for a Data Abort exception routed to Hyp mode, `0x24`. |
| *Instruction Length*, IL | The PE sets this to 1. |
| *Instruction Specific Syndrome*, ISS | **ISV[24]** *Instruction Syndrome Valid* (ISV). The PE sets this to 0.<br>**ISS[23:10]** RES0.<br>**ISS[9]** *External Abort type* (EA). The PE sets this to 0.<br>**ISS[8]** *Cache Maintenance* (CM). The PE sets this to indicate whether a cache maintenance instruction caused the exception.<br>**ISS[7]** RES0.<br>**ISS[6]** *Write not Read* (WnR). The PE sets this to indicate whether the exception was generated on a read instruction or a write instruction.<br>**ISS[5:0]** *Data Fault Status Code* (DFSC). The PE sets this to the code for a debug exception, `0b100010`. |

**Note**

For information about how debug exceptions can be routed to PL2, see *Routing debug exceptions* on page G2-3887.

### Preferred return address

The preferred return address of a Watchpoint exception is the address of the instruction that was not executed because the PE took the Watchpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

## G2.10.9 Pseudocode description of Watchpoint exceptions taken from AArch32 state

`AArch32.WatchpointByteMatch()` tests an individual byte accessed by an operation.

```
// AArch32.WatchpointByteMatch()
// =============================
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)

    bottom = if DBGWVR[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool();
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));  MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                    // Not contiguous
            byte_select_match = ConstrainUnpredictableBool();
            bottom = 3;                                       // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;           // Disabled
            when Constraint_NONE      mask = 0;               // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<31:mask> == DBGWVR[n]<31:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool();
    else
        WVR_match = vaddress<31:bottom> == DBGWVR[n]<31:bottom>;

    return WVR_match && byte_select_match;
```

AArch32.StateMatch() tests the values in DBGWCR<n>.{HMC, SSC, PAC}, and if the watchpoint is Linked, also tests the Linked Context breakpoint that the watchpoint links to. AArch32.StateMatch() is given in the Breakpoint exceptions section. See page G2-3919.

AArch32.WatchpointMatch() tests the value in DBGWVR<n>.

```
// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
                                boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIDR.WRPs);

    // "ispriv" is FALSE for LDRT/STRT instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, DBGWCR[n].LBN, isbreakpnt, ispriv);

    ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return value_match && state_match && ls_match && enabled;
```

AArch32.CheckWatchpoint() generates a FaultRecord that AArch64.Abort raises a Watchpoint exception for if all of the following are true:

- DBGDSCRext.MDBGen is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-3889.
- All of the conditions required for Watchpoint exception generation are met. See *About Watchpoint exceptions* on page G2-3923.

**Note**

AArch32.CheckWatchpoint() might halt the PE and cause it to enter Debug state. External debug uses Debug state.

```
// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = PSTATE.EL != EL0 && !(PSTATE.EL == EL1 && acctype == AccType_UNPRIV);

    for i = 0 to UInt(DBGDIDR.WRPs)
        match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elsif match && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

The Abort() function processes the Faultrecord object returned by CheckWatchpoint, as described in *Abort exceptions* on page G3-3985. If there is a Watchpoint exception, the Abort() function generates a Data Abort exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G2.11 Vector Catch exceptions

ARM deprecates the use of vector catch.

This section describes Vector Catch exceptions in stage 1 of an AArch32 translation regime. It contains the following:


- *About Vector Catch exceptions*.
- *Exception vectors that Vector Catch exceptions can be enabled for* on page G2-3938.
- *Generation of Vector Catch exceptions* on page G2-3940.
- *Usage constraints* on page G2-3942.
- *Exception syndrome information and preferred return address* on page G2-3942.
- *Pseudocode description of Vector Catch exceptions* on page G2-3943.


### G2.11.1 About Vector Catch exceptions

Whenever the PE takes an exception, execution is forced to an address that is the *exception vector* for that exception. Vector catch permits a debugger to trap exceptions based on the exception vector, or based on the exception type associated with the exception vector, as follows:

- If the *address-matching* form of vector catch is implemented, the debugger can trap exceptions based on the exception vector.
- If the *exception-trapping* form of vector catch is implemented, the debugger can trap exceptions based on the exception type associated with the exception vector.

The ARMv8-A architecture supports only these two forms of vector catch. Only one form can be implemented, and which is implemented is IMPLEMENTATION DEFINED. The DBGDEVID indicates which form is implemented.

Regardless of the form of vector catch implemented, a debugger enables Vector Catch exceptions for exception vectors or types by programming the DBGVCR. This register contains *vector catch enable bits*. Each of these bits corresponds to a different vector. When a debugger sets a vector catch enable bit to 1, Vector Catch exceptions are enabled for the corresponding exception vector or type.

**Note**

EL2 using AArch64 or EL3 using AArch64 can enable Vector Catch exceptions for vectors by programming the DBGVCR32_EL2. The DBGVCR32_EL2 is architecturally mapped to the DBGVCR.

When Vector Catch exceptions are enabled for an exception vector, this is called an *enabled vector catch*. The set of exception vectors that Vector Catch exceptions are enabled for is called the *enabled vector catch set*.

If the form of vector catch implemented is the:

**Address-matching form:**

The PE compares the virtual address of each instruction in the program flow with a subset of the enabled vector catch set.

If an address match occurs, a Vector Catch exception is generated when the instruction that caused the match is committed for execution.

**Exception-trapping form**

Whenever the PE takes an exception, if the vector the exception is taken to is included in a subset of the enabled vector catch set, a Vector Catch exception is generated.

The Vector Catch exception is generated as part of entry to the exception, and must be taken before the PE either executes any instructions or takes any further exceptions.

The addresses that comprise the subset depend on whether EL3 is implemented and, for the:

- Address-matching form, the current Security state.
- Exception-trapping form, the Security state that the exception is handled in.

See *Generation of Vector Catch exceptions* on page G2-3940.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G2-20 summarizes the differences between the address-matching and exception-trapping forms.

**Table G2-20 Differences in behavior of the address-matching and exception-trapping forms of vector catch**

| Address-matching | Exception-trapping |
|---|---|
| An enabled vector catch generates a Vector Catch exception when an instruction that is fetched from the vector is committed for execution.<br>This means that spurious Vector Catch exceptions might occur, where the Vector Catch exception does not result from an exception entry, but is instead caused by a branch to the vector.<br>A branch to the vector might occur, for example, on a return from a nested exception or when simulating an exception entry. | An enabled vector catch generates a Vector Catch exception immediately after the PE takes the exception that is associated with the vector.<br>This means that Vector Catch exceptions always result from exception entry, and not from branches to exception vectors. |
| A Vector Catch exception is generated as a result of an instruction fetch. This means that the Vector Catch exception has a priority relative to the other synchronous exceptions that result from an instruction fetch.<br>*Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 describes this prioritization. | A Vector Catch exception is generated as a result of an exception entry. This means that the Vector Catch exception is part of the exception that caused the Vector Catch exception. Therefore, the Vector Catch exception has no priority associated with it.<br>For this reason, Vector Catch exceptions are outside the scope of the prioritization that *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 describes. |
| A Vector Catch exception can be preempted by another exception.<br>If this happens, the Vector Catch exception is generated again when the exception handler branches back to the vector. | Vector Catch exceptions must be taken before other exceptions. |
| A Vector Catch exception can be generated as a result of an instruction fetch executed in any AArch32 mode except Hyp mode, including User mode. | Because a Vector Catch exception is generated as the result of an exception entry, the Vector Catch exception is only generated when the PE is in the AArch32 exception handling mode. |
| If HCR.TGE is 1, Vector Catch exceptions can be generated for User mode instruction fetches from Non-secure PL1 vectors. | If HCR.TGE is 1, Vector Catch exceptions are never generated in Non-secure state, because:<br>• Exceptions are routed away from Non-secure PL1 vectors, to PL2.<br>• The architecture does not provide vector catch enable bits for the Hyp exception vectors. |

Depending on the implementation, some vector catch enable bits in the DBGVCR might be RES0. For example, if EL3 is not implemented or is implemented but is using AArch64, Monitor mode is not implemented, and so the enable bits for exception vectors for exceptions taken to Monitor mode are RES0. See *Exception vectors that Vector Catch exceptions can be enabled for* for the vector catch enable bits that exist for different implementations.

*The debug exception enable controls* on page G2-3886 describes the enable controls for Vector Catch exceptions.

### G2.11.2 Exception vectors that Vector Catch exceptions can be enabled for

When the PE takes an exception, the exception vector is contained in a *vector table* at the Privilege level the exception is taken to.

Depending on the Security state and AArch32 mode the exception is taken to, when the exception is taken, the vector table used is the table that contains one of:

- *Local exception vectors*.
- *Non-secure Local exception vectors*.
- *Secure Local exception vectors*.
- *Hyp exception vectors*.
- *Monitor exception vectors*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G2-21 shows which vector tables are implemented for different implementations. In the table:

- A dash, -, means that the Exception level is not implemented.
- 64 means that the Exception level is using AArch64.
- 32 means that the Exception level is using AArch32.

**Table G2-21 Vector tables implemented for different implementations**

| Implementation | | | | Vector table or tables implemented |
|---|---|---|---|---|
| EL0 | EL1 | EL2 | EL3 | |
| 32 | 32 | - | - | Local exception vectors. |
| | | 64 | - | Non-secure Local exception vectors. |
| | | 32 | - | Non-secure Local exception vectors.<br>Hyp exception vectors. |
| | | - | 64 | Secure Local exception vectors.<br>Non-secure Local exception vectors. |
| | | - | 32 | Secure Local exception vectors.<br>Non-secure Local exception vectors.<br>Monitor exception vectors. |
| | | 64 | 64 | Secure Local exception vectors.<br>Non-secure Local exception vectors. |
| | | 32 | 64 | Secure Local exception vectors.<br>Non-secure Local exception vectors.<br>Hyp exception vectors. |
| | | 32 | 32 | Secure Local exception vectors.<br>Non-secure Local exception vectors.<br>Hyp exception vectors.<br>Monitor exception vectors. |

For example, in an AArch32-only implementation that includes EL0, EL1, and EL3, when the PE takes an exception to Monitor mode, it uses the vector table containing Monitor exception vectors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The tables that follow show the vectors that Vector Catch exceptions can be enabled for, and their corresponding vector catch enable bits in the DBGVCR:

- Table G2-22 shows the Local exception vectors, Secure Local exception vectors, and Non-secure Local exception vectors that Vector Catch exceptions can be enabled for.
- Table G2-23 shows the Monitor exception vectors that Vector Catch exceptions can be enabled for.

The ARMv8-A architecture does not provide vector catch enable bits for the Hyp exception vectors.

**Table G2-22 Local exception vectors, Secure Local exception vectors, and Non-secure Local exception vectors that Vector Catch exceptions can be enabled for**

| Vector catch enable bit | | Exception type | Local exception vectors | |
|---|---|---|---|---|
| Local or Secure Local exception vectors | Non-secure Local exception vectors | | Normal. SCTLR.V is 0.[a] | High. SCTLR.V is 1. |
| SF | NSF | FIQ interrupt | VBAR + 0x0000001C | 0xFFFF001C |
| SI | NSI | IRQ interrupt | VBAR + 0x00000018 | 0xFFFF0018 |
| SD | NSD | Data Abort | VBAR + 0x00000010 | 0xFFFF0010 |
| SP | NSP | Prefetch Abort | VBAR + 0x0000000C | 0xFFFF000C |
| SS | NSS | Supervisor Call | VBAR + 0x00000008 | 0xFFFF0008 |
| SU | NSU | Undefined Instruction | VBAR + 0x00000004 | 0xFFFF0004 |

a. If EL3 is implemented and is using AArch32, VBAR is banked. The Secure Local exception vectors use $VBAR_S$ and the Non-secure Local Exception vectors use $VBAR_{NS}$.

**Table G2-23 Monitor exception vectors that Vector Catch exceptions can be enabled for**

| Vector catch enable bit | Exception type | Monitor exception vectors |
|---|---|---|
| MF | FIQ interrupt | MVBAR + 0x0000001C |
| MI | IRQ interrupt | MVBAR + 0x00000018 |
| MD | Data Abort | MVBAR + 0x00000010 |
| MP | Prefetch Abort | MVBAR + 0x0000000C |
| MS | Secure Monitor Call | MVBAR + 0x00000008 |

**Note**

There is no Vector catch enable bit for Monitor trap exceptions.

### G2.11.3 Generation of Vector Catch exceptions

How Vector Catch exceptions are generated depends on which form is implemented:

- *Address-matching form* on page G2-3941.
- *Exception-trapping form* on page G2-3941.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Address-matching form

The PE compares the virtual address of each instruction in the program flow is with some or all of the addresses in the enabled vector catch set, as follows:

- If EL3 is not implemented, the enabled vector catch set contains only Local exception vectors. The PE compares the virtual address of each instruction in the program flow, including those executed at EL0, with all addresses in the enabled vector catch set.
- If EL3 is implemented, the enabled vector catch set might contain one or more of the following:
  - Monitor exception vectors, if EL3 is using AArch32.
  - Secure Local exception vectors.
  - Non-secure Local exception vectors.

  In this case, Table G2-24 shows which addresses, in the enabled vector catch set, the virtual address of each instruction in the program flow is compared with.

**Table G2-24 Comparisons made if the implementation includes EL3**

| EL3 is using | For exceptions taken to: | |
|---|---|---|
| | Secure PL1 modes | Non-secure PL1 modes |
| AArch64 | Secure Local exception vectors | Non-secure Local exception vectors |
| AArch32 | Secure Local exception vectors and Monitor exception vectors | |

For example, for exceptions taken to a Secure PL1 mode when EL3 is using AArch64, the virtual address of each instruction in the program flow is compared with each Secure Local exception vector in the enabled vector catch set.

For each instruction in the program flow, the PE tests for any possible Vector Catch exceptions before executing the instruction. If a match occurs, a Vector Catch exception is generated when the instruction is committed for execution, regardless of all of the following:

- Whether the instruction passes its condition code check.
- Whether the instruction is executed as part of exception entry.
- If EL2 is implemented, what HCR.{IMO, FMO, AMO} are set to.
- If EL3 is implemented, what SCR.{IRQ, FIQ, EA} are set to.

### Exception-trapping form

When the PE takes an exception, it tests whether the exception is by branching to an exception vector in a subset of the enabled vector catch set, as follows:

- If EL3 is not implemented, the enabled vector catch set contains only Local exception vectors. The PE tests whether the exception is by branching to any address in the enabled vector catch set.
- If EL3 is implemented, the enabled vector catch set might contain one or more of the following:
  - Monitor exception vectors, if EL3 is using AArch32.
  - Secure Local exception vectors.
  - Non-secure Local exception vectors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In this case, the PE tests whether the exception is by branching to a vector in one of the subsets that Table G2-25 shows. In the table, n/a means not applicable.

**Table G2-25 Subsets that the PE tests within if EL3 is implemented**

| EL3 is using | For exceptions taken to: | | |
|---|---|---|---|
| | Monitor mode | Other Secure PL1 modes | Non-secure PL1 modes |
| AArch64 | n/a | Secure Local exception vectors | Non-secure Local exception vectors |
| AArch32 | Monitor exception vectors | | |

For example, for an exception taken to a Secure PL1 mode when EL3 is using AArch64, the PE tests whether the exception is by branching to any of the Secure Local exception vectors in the enabled vector address set.

If the exception is by branching to a vector in the subset, a Vector Catch exception is generated as part of exception entry. That is, a Vector Catch exception is generated instead of the exception handler executing its first instruction.

## G2.11.4 Usage constraints

See the following subsections:

- *Usage constraints that apply to both forms of vector catch*.
- *Usage constraints that apply only to the address-matching form*.

### Usage constraints that apply to both forms of vector catch

For Vector Catch exceptions enabled for either the Prefetch Abort exception vector or the Data Abort exception vector, if one of these exception types is taken to the Exception level that debug exceptions are targeting, behavior is CONSTRAINED UNPREDICTABLE. Either:

- Vector catch is ignored, therefore a Vector Catch exception is not generated.
- Vector catch generates a Prefetch Abort debug exception. For Vector Catch exceptions enabled for the Prefetch Abort exception vector, the PE might enter a recursive loop of Prefetch Abort exceptions causing Vector Catch exceptions and Vector Catch exceptions causing Prefetch Abort exceptions.

**Note**

The Exception level that debug exceptions are targeting is called the *debug target Exception level*, $EL_D$. *Routing debug exceptions* on page G2-3887 describes how $EL_D$ is derived.

### Usage constraints that apply only to the address-matching form

Exception vectors are at word-aligned addresses, and:

- It is CONSTRAINED UNPREDICTABLE whether an enabled vector catch generates a Vector Catch exception for a 32-bit T32 instruction starting at the halfword-aligned address immediately prior to the vector address.
- T32 instructions that start at the halfword-aligned address immediately after the exception vector do not generate Vector Catch exceptions.

For the address-matching form, Vector Catch exceptions have the same priority as Breakpoint exceptions. If a single instruction causes both a Vector Catch exception and a Breakpoint exception, it is CONSTRAINED UNPREDICTABLE which of these debug exceptions the PE takes.

## G2.11.5 Exception syndrome information and preferred return address

See the following:

- *Exception syndrome information* on page G2-3943.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Preferred return address*.

### Exception syndrome information

The PE takes a Vector Catch exception as either:

- A Prefetch Abort exception, if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

**PL1 Abort mode**

The PE sets all of the following:

- IFSR.FS to the code for a debug event, `0b00010`.
- DBGDSCRext.MOE to `0b0101`, to indicate a Vector Catch exception.
- The IFAR with an UNKNOWN value.

**PL2 Hyp mode**

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See Table G2-26.
- Sets DBGDSCRext.MOE to `0b0101`, to indicate a Vector Catch exception.
- Sets the HIFAR to an unknown value.

**Table G2-26 Information recorded in the HSR**

| HSR field | Information recorded |
|---|---|
| *Exception Class*, EC | The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, `0x20`. |
| *Instruction Length*, IL | The PE sets this to 1. |
| *Instruction Specific Syndrome*, ISS | **ISS[24:10]** RES0.<br>**ISS[9]** *External Abort type* (EA). The PE sets this to 0.<br>**ISS[8:6]** RES0.<br>**ISS[5:0]** *Instruction Fault Status Code* (IFSC). The PE sets this to the code for a debug exception, `0b100010`. |

**Note**

For information about how debug exceptions can be routed to PL2, see *Routing debug exceptions on page G2-3887*.

### Preferred return address

The preferred return address of a Vector Catch exceptions is the address of the instruction that was not executed because the PE took the Vector Catch exception instead.

This means that the preferred return address is the exception vector. This is true regardless of whether the address-matching form or the exception trapping form is implemented.

## G2.11.6 Pseudocode description of Vector Catch exceptions

The `AArch32.VCRMatch()` pseudocode function checks whether the instruction at `address` generates a Vector Catch exception. It therefore shows the address-matching form of vector catch.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// AArch32.VCRMatch()
// ==================

boolean AArch32.VCRMatch(bits(32) vaddress)

    if UsingAArch32() && ELUsingAArch32(EL1) && IsZero(vaddress<1:0>) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && !IsSecure() then
                match_word<UInt(vaddress<4:2>) + 24> = '1';     // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1';      // Secure vectors (or no EL3)
        if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() && vaddress<31:5> == MVBAR<31:5> then
            match_word<UInt(vaddress<4:2>) + 8> = '1';          // Monitor vectors

        // Mask out bits not corresponding to vectors.
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
        elsif !ELUsingAArch32(EL3) then
            mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
        else
            mask = '11011110':'00000000':'11011100':'11011110';

        match_word = match_word AND DBGVCR AND mask;
        match = !IsZero(match_word);

        // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
        if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
            match = ConstrainUnpredictableBool();
    else
        match = FALSE;

    return match;
```

The `AArch32.CheckVectorCatch()` pseudocode function uses `VCRMatch()` to test whether the instruction generates a Vector Catch exception, and if `VCRMatch()` returns TRUE it generates that event.

```
// AArch32.CheckVectorCatch()
// ==========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// Vector Catch can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = AArch32.VCRMatch(vaddress);
    if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
        match = ConstrainUnpredictableBool();

    if match && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_VectorCatch;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

The `Abort()` function processes the Faultrecord object returned by `CheckVectorCatch`, as described in *Abort exceptions* on page G3-3985. If there is a Vector Catch exception, the `Abort()` function generates a Prefetch Abort exception.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G2.12 Synchronization and debug exceptions

The behavior of debug depends on all of the following:

- The state of the external debug authentication interface.
- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.

If a change is made to any of these, the effect of that change on debug exception generation cannot be relied on until after a *Context Synchronization Operation* (CSO) has occurred.

For any instructions executed between the time when the change is made and the time when the next CSO occurs, it is CONSTRAINED UNPREDICTABLE whether debug uses the state of the PE before the change, or the state of the PE after the change.

**Example G2-3**

1. Software changes DBGDSCRext.MDBGen from 0 to 1.
2. An instruction is executed, that would cause a Breakpoint exception if self-hosted debug uses the state of the PE after the change.
3. A CSO occurs.

In this case, it is CONSTRAINED UNPREDICTABLE whether the instruction generates a Breakpoint exception.

**Example G2-4**

1. Software unlocks the OS lock.
2. The PE executes some instructions.
3. A CSO occurs.

During the time when the PE is executing some instructions, step 2, it is CONSTRAINED UNPREDICTABLE whether debug exceptions other than Software Breakpoint Instruction exceptions can be generated.

**Note**

- See *Context synchronization operation* for the definition of this term.
- Some register updates are self-synchronizing. Others require an explicit CSO. For more information, see both:
  - *Synchronization requirements for System registers* on page D7-1908.
  - *Accessing PSTATE fields* on page G1-3768.
  - *Synchronization of changes to the external debug registers* on page H8-4946.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G2.12.1 State and mode changes without explicit context synchronization operations

Most changes to the Privilege level, and the Security state if EL3 is implemented, happen as a result of operations that are an explicit CSO. This is because taking an exception and returning from an exception are both explicit CSOs, and the Privilege level and Security state can only change as a result of taking or returning from an exception.

However, some Security state and AArch32 mode changes can happen because of operations that are not an explicit CSO. These are:

- AArch32 mode changes caused by `MSR` and `CPS` instructions. A mode change might be to a mode at a lower Privilege level.
- If EL3 is using AArch32, a Security state change caused by a direct write to the SCR in a privileged mode other than Monitor mode, to set SCR.NS to 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter G3
# The AArch32 System Level Memory Model

This chapter provides a system level view of the general features of the memory system. It contains the following sections:


- *About the memory system architecture* on page G3-3948.
- *Address space* on page G3-3949.
- *Mixed-endian support* on page G3-3950.
- *AArch32 cache and branch predictor support* on page G3-3952.
- *System register support for IMPLEMENTATION DEFINED memory features* on page G3-3975.
- *External aborts* on page G3-3976.
- *Memory barrier instructions* on page G3-3978.
- *Pseudocode description of general memory system instructions* on page G3-3979.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G3.1 About the memory system architecture

The ARM architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that permits implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system see *Memory hierarchy* on page E2-2358.

### G3.1.1 Form of the memory system architecture

The ARMv8 A-profile architecture includes a *Virtual Memory System Architecture* (VMSA), described in Chapter G4 *The AArch32 Virtual Memory System Architecture*.

### G3.1.2 Memory attributes

*Memory types and attributes* on page E2-2382 describes the memory attributes, including how different memory types have different attributes. Each location in memory has a set of memory attributes, and the translation tables define the virtual memory locations, and the attributes for each location.

Table G3-1 shows the memory attributes that are visible at the system level.

**Table G3-1 Memory attribute summary**

| Memory type | Shareability | Cacheability |
|---|---|---|
| Device[a] | Outer Shareable | Non-cacheable. |
| Normal | One of:<br>• Non-shareable.<br>• Inner Shareable.<br>• Outer Shareable. | One of:<br>• Non-cacheable[b].<br>• Write-Through Cacheable.<br>• Write-Back Cacheable. |

a. Takes additional attributes, see *Device memory* on page E2-2387.
b. See also *Cacheability, cache allocation hints, and cache transient hints* on page G3-3954.

For more information on Cacheability and Shareability see *The Cacheability and Shareability memory attributes* on page E2-2359, *Non-shareable Normal memory* on page E2-2384, and *Caches and memory hierarchy* on page E2-2358.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G3.2 Address space

The ARMv8 architecture is designed to support a wide range of applications with different memory requirements. It supports a range of *physical address* (PA) sizes, and provides associated control and identification mechanisms. For more information, see *About VMSAv8-32* on page G4-3992.

### G3.2.1 Address space overflow or underflow

This subsection describes address space overflow or underflow:

#### Instruction address space overflow

When a PE performs a normal, sequential execution of instructions, it calculates:

```
(address_of_current_instruction) + (size_of_executed_instruction)
```

This calculation is performed after each instruction to determine which instruction to execute next.

If the address calculation performed after executing an A32 or T32 instruction overflows `0xFFFF FFFF`, the program counter becomes UNKNOWN.

If the PE executes an instruction for which the instruction address, size, and alignment mean that it contains the bytes `0xFFFFFFFF` and `0x00000000`, the bytes that apparently from `0x00000000` onwards come from an UNKNOWN address.

#### Data address space overflow and underflow

If the PE executes a load or store instruction for which the computed address, total access size, and alignment mean that it accesses bytes `0xFFFFFFFF` and `0x00000000`, then the bytes that apparently come from `0x00000000` onwards come from UNKNOWN addresses.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G3.3 Mixed-endian support

Table G3-2 shows the endianness of explicit data accesses and translation table walks.

**Table G3-2 Endianness support**

| Exception level | Explicit data accesses | Stage 1 translation table walks | Stage 2 translation table walks |
|---|---|---|---|
| EL0 | PSTATE.E | SCTLR(S/NS).EE | HSCTLR.EE |
| EL1 | PSTATE.E | SCTLR(S/NS).EE | HSCTLR.EE |
| EL2 | PSTATE.E | HSCTLR.EE | N/A |
| EL3 | PSTATE.E | SCTLR(S).EE | N/A |

ARMv8 provides the following options for endianness support:

- All Exception levels support mixed-endianness:
  - SCTLR(S/NS).EE, HSCTLR.EE, and PSTATE.E are RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only little-endianness:
  - SCTLR(S/NS).EE and HSCTLR.EE are RES0. PSTATE.E is RW when in EL0 and RES0 when in EL1, EL2, or EL3. SPSR.E is also RES0 when not returning to EL0.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only big-endianness:
  - SCTLR(S/NS).EE and HSCTLR.EE are RES1. PSTATE.E is RW when in EL0 and RES1 when in EL1, EL2, or EL3. SPSR.E is also RES1 when not returning to EL0.
- All Exception levels support only little-endianness:
  - Each of SCTLR(S/NS).EE, HSCTLR.EE, PSTATE.E, and SPSR.E is RES0.
- All Exception levels support only big-endianness:
  - Each of SCTLR(S/NS).EE, HSCTLR.EE, PSTATE.E, and SPSR.E is RES1.

If mixed endian support is implemented for an Exception level using AArch32, endianness is controlled by PSTATE.E. For exception returns to AArch32 state, PSTATE.E is copied from SPSR_ELx.E. If the target Exception level supports only little-endian accesses, SPSR_ELx.E is RES0. If the target Exception level supports only big-endian accesses, SPSR_ELx.E is RES1.

**Note**

- When using AArch32, ARM deprecates PSTATE.E having a different value from the equivalent System register EE bit when in EL1, EL2 or EL3. The use of the SETEND instruction is also deprecated.
- If the higher Exception levels are using AArch64, the corresponding registers are:
  - SCTLR_EL1 for SCTLR(NS).
  - SCTLR_EL2 for HSCTLR.
  - SCTLR_EL3 for SCTLR(S).

The BigEndian() function determines whether the current Exception level and Execution state is using big-endian data:

```
// BigEndian()
// ===========

boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bigend = (SCTLR_EL1.E0E != '0');
else
    bigend = (SCTLR[].EE != '0');
return bigend;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G3.4 AArch32 cache and branch predictor support

Cache support includes:

- Cache identification. See *Cache identification* on page G3-3953.
- Write-through and Write-back attributes, and cache allocation hints and cache transient hints. See *Cacheability, cache allocation hints, and cache transient hints* on page G3-3954.
- Caches and reset. See *Behavior of caches at reset* on page G3-3955.
- Enabling and disabling caches. See *Cache enabling and disabling in AArch32 state* on page G3-3955.
- Cache and branch predictor maintenance. See *AArch32 cache and branch predictor maintenance instructions* on page G3-3961.

Additional information relating to caches is provided in the following subsections:

- *The ARMv8 cache maintenance functionality* on page G3-3957.
- *AArch32 cache and branch predictor maintenance instructions* on page G3-3961.
- *Cache lockdown* on page G3-3972.
- *System level caches* on page G3-3974.

See also Chapter G4 *The AArch32 Virtual Memory System Architecture*.

**Note**

- Branch predictors typically use a form of cache to hold branch target data. Therefore, they are included in this section.
- In the instruction mnemonics, MVA is a synonym for VA.

### G3.4.1 General behavior of the caches

When a memory location is marked with a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. The following non-exhaustive list of factors might be involved:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- A memory location present in the cache remains in the cache.
- A memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
  - A particular implementation.
  - Some memory attributes.
- An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.
- A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

For more information, see *The interaction of cache lockdown with cache maintenance instructions* on page G3-3972.

- If a memory location is marked as cacheable for the translation regime at either the current Exception level or at a higher Exception level, then there is no mechanism that can guarantee that the memory location cannot be allocated to an enabled cache at any time.

  Any application must assume that any memory location with such access permissions and cacheability attributes can be allocated to any enabled cache at any time.

- If the cache is disabled, it is guaranteed that no new allocation of memory locations into the cache occurs.

- If the cache is enabled, it is guaranteed that no memory location that does not have a Cacheable attribute is allocated into the cache.

- If the cache is enabled, it is guaranteed that no memory location is allocated to the cache if it is not marked as cacheable in both:
  — The translation regime at the current Exception level.
  — The translation regime at a higher Exception level.

- For data accesses, any memory location that is marked as Normal Inner Shareable or Normal Outer Shareable is guaranteed to be coherent with all masters in its Shareability domain.

- Any memory location is not guaranteed to remain incoherent with the rest of memory.

- The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by an observer in the Shareability domain of that memory location. The maximum size of the memory that can be overwritten is called the *Cache Write-back Granule*. In some implementations the CTR identifies the Cache Write-back Granule, see *CTR, Cache Type Register* on page G6-4276.

- The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer, if it was previously visible to that observer.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to its size.

### G3.4.2 Cache identification

The ARMv8 cache identification consists of a set of registers that describe the implemented caches that are under the control of the PE:

- A single Cache Type Register defines:
  — The minimum line length of any of the instruction caches affected by the instruction cache maintenance instructions.
  — The minimum line length of any of the data or unified caches, affected by the data cache maintenance instructions.
  — The cache indexing and tagging policy of the Level 1 instruction cache.

  For more information, see *CTR, Cache Type Register* on page G6-4276.

- A single Cache Level ID Register defines:
  — The type of cache implemented at each cache level, up to the maximum of seven levels.
  — The *Level of Unification Inner Shareable* (LoUIS), *Level of Coherence* (LoC) and the *Level of Unification* (LoU) for the caches. See *Terms used in describing the maintenance instructions* on page G3-3957 for a definition of these terms.
  — An optional ISB field to indicate the cacheable boundary between inner and outer domains.

  For more information, see *CLIDR, Cache Level ID Register* on page G6-4257.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A single Cache Size Selection Register selects the cache level and cache type of the current Cache Size Identification Register, see *CSSELR, Cache Size Selection Register* on page G6-4274.
- For each implemented cache, across all the levels of caching, a Cache Size Identification Register defines:
  - Whether the cache supports Write-Through, Write-Back, Read-Allocate and Write-Allocate.
  - The number of sets, associativity and line length of the cache. See *Terms used in describing the maintenance instructions* on page G3-3957 for a definition of these terms.

  For more information, see *CCSIDR, Current Cache Size ID Register* on page G6-4254.

To determine the cache topology associated with a PE:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache. This register also provides the size of the smallest cache lines used for the instruction caches, and for the data and unified caches. These values are used in cache maintenance instructions.
2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache type fields, for cache levels 1 to 7. Scanning these fields, starting from Level 1, identifies the instruction, data or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are defined. The Cache Level ID Register also specifies the Level of Unification (LoU) and the Level of Coherence (LoC) for the cache implementation.
3. For each cache identified at stage 2:
   - Write to the Cache Size Selection Register to select the required cache. A cache is identified by its level, and whether it is:
     - An instruction cache.
     - A data or unified cache.
   - Read the Cache Size ID Register to find details of the cache.

### G3.4.3 Cacheability, cache allocation hints, and cache transient hints

Cacheability only applies to Normal memory, and is defined independently for Inner and Outer cache locations.

As described in *Memory types and attributes* on page E2-2382, the memory attributes include a cacheability attribute that is one of:

- Non-cacheable.
- Write-Through cacheable.
- Write-Back cacheable.

Cacheability attributes other than Non-cacheable can be complemented by a *cache allocation hint*. This is an indication to the memory system of whether allocating a value to a cache is likely to improve performance. A *cache transient hint* provides a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. These hints are used to limit cache pollution to a part of a cache, such as to a subset of ways.

The following cache allocation hints can be used in ARMv8:

- Read-Allocate, Transient Read-Allocate, or no Read-Allocate.
- Write-Allocate, Transient Write-Allocate, or no Write-Allocate.

**Note**

A Cacheable location with both no Read-Allocate and no Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply for a location that is Cacheable, no Read-Allocate, no Write-Allocate.

The architecture does not require an implementation to make any use of cache allocation hints. This means an implementation might not make any distinction between memory locations with attributes that differ only in their cache allocation hint.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G3.4.4 Behavior of caches at reset

In ARMv8:

- All caches reset to IMPLEMENTATION DEFINED states that might be UNKNOWN.
- All caches are disabled at reset.
- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before it is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, and the routine must be documented clearly as part of the documentation of the device.
- If an implementation permits cache hits when the cache is disabled the cache initialization routine must:
  - Provide a mechanism to ensure the correct initialization of the caches.
  - Be documented clearly as part of the documentation of the device.

  In particular, if an implementation permits cache hits when the cache is disabled and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.
- ARM recommends that whenever an invalidation routine is required, it is based on the ARMv8 cache maintenance instructions.

Similar rules apply to:

- Branch predictor behavior, see *Behavior of the branch predictors at reset* on page G3-3965.
- TLB behavior, see *TLB behavior at reset* on page G4-4059.

### G3.4.5 Cache enabling and disabling in AArch32 state

When a data cache or unified cache is disabled for a translation regime, as determined by SCTLR.C or HSCTLR.C, data accesses and translation table walks from that translation regime to all Normal memory types behave as Non-cacheable for all levels of data caches and unified caches.

For the PL1&0 translation regime:

- When the value of SCTLR.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the PL1&0 stage 1 translation tables are Non-cacheable.
- When the value of HCR2.CD is 1:
  - All stage 2 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the PL1&0 stage 2 translation tables are Non-cacheable.

  **Note**
  - The stage 1 and stage 2 cacheability attributes are combined as described in *Combining the Cacheability attribute* on page G4-4056.
  - The SCTLR.C bit has no effect on the EL2 and EL3 translation regimes.
  - The HCR2.CD bit affects only stage 2 of the Non-secure PL1&0 translation regime.
- When the value of HCR.DC is 1, the stage 1 PL1&0 translation regime is cacheable regardless of the value of SCTLR.C.

For the EL2 translation regime:

- When the value of HSCTLR.C is 0, all data accesses to Normal memory using the EL2 translation regime are Non-cacheable. This means all accesses made by the EL2 translation table walks are Non-cacheable.

  **Note**

  The HSCTLR.C bit has no effect on the PL1&0 and EL3 translation regimes.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the EL3 translation regime:

- When the value of SCTLR.C is 0, all data accesses to Normal memory using the EL3 translation regime are Non-cacheable. This means all accessed made by the EL3 translation table walks are Non-cacheable.

  **Note**

  The SCTLR.C bit has no effect on the PL1&0 and EL2 translation regimes.

The effect of the SCTLR.C or HSCTLR.C and HCR2.CD bits is reflected in the result of the address translation instructions in the PAR.

Disabling the instruction cache for a translation regime, as determined by the SCTLR.I or HSCTLR.I, makes instruction accesses to all Normal memory types behave as Non-cacheable for all levels of instruction or unified cache.

For the PL1&0 translation regime:

- When SCTLR.I == 0, all instruction accesses to Normal memory are made Non-cacheable at the first stage of translation.
- When HCR2.ID == 1, all instruction accesses to Normal memory are made Non-cacheable at the second stage of translation.

  **Note**

  — The stage 1 and stage 2 cacheability attributes are combined as described in *Combining the Cacheability attribute* on page G4-4056.

  — The SCTLR.I bit has no effect on the EL2 and EL3 translation regimes.

  — The HCR2.ID bit affects only stage 2 of the Non-secure PL1&0 translation regime.

- If HCR2.DC is set to 1, then the Non-secure stage 1 PL1&0 translation regime is cacheable regardless of the value of SCTLR.I.

For the EL2 translation regime:

- When the value of the HSCTLR.I bit is 0, all instruction accesses to Normal memory using the EL2 translation regime are Non-cacheable.

  **Note**

  The HSCTLR.I bit has no effect on the PL1&0 and EL3 translation regimes.

For the EL3 translation regime:

- When the value of the SCTLR.I bit is 0, all instruction accesses to Normal memory using the EL3 translation regime are Non-cacheable.

  **Note**

  The SCTLR.I bit has no effect on the PL1&0 and EL2 translation regimes

In addition, for the Secure EL1, EL2, and EL3 translation regimes, when the value of SCTLR.M or HSCTLR.M is 0, indicating that the stage 1 translations are disabled for that translation regime, SCTLR.I or HSCTLR.I has the following effect:

- If SCTLR.I ==0 or HSCTLR.I == 0, instruction accesses to Normal memory from stage 1 of that translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If SCTLR.I == 1 or HSCTLR.I == 1, instruction accesses to Normal memory from stage 1 of that translation regime are Outer Shareable, Inner Write-Through, Outer Write-Through.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the Non-secure EL1 translation regime, when the value of SCTLR.M is 0, indicating that the stage 1 translations are disabled for that translation regime, and HCR_EL2.DC == 0, the SCTLR.I bit has the following effect:

- If SCTLR.I == 0 or HSCTLR.I == 0, instruction accesses to Normal memory from stage 1 of that translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If SCTLR.I == 1 or HSCTLR.I == 1, instruction accesses to Normal memory from stage 1 of that translation regime are Outer Shareable, Inner Write-Through, Outer Write-Through.

**Note**

- This means that the architecturally required effect of SCTLR.I or HSCTLR.I is limited to its effect on caching instruction accesses in unified caches.
- This specification can give rise to different cacheability attributes between instruction and data accesses to the same location. Where this occurs, the measures for mismatch memory attributes described in *Mismatched memory attributes* on page E2-2392 must be followed to manage the corresponding loss of coherency.

## G3.4.6 The ARMv8 cache maintenance functionality

The following sections give general information about the ARMv8 cache maintenance functionality:

- *Terms used in describing the maintenance instructions*.
- *The ARMv8 abstraction of the cache hierarchy* on page G3-3960.

The following sections describe the AArch32 state cache maintenance instructions for ARMv8:

- *AArch32 instruction cache maintenance instructions (IC*)* on page G3-3962.
- *AArch32 data cache maintenance instructions (DC*)* on page G3-3963.

### Terms used in describing the maintenance instructions

Cache maintenance instructions are defined to act on particular memory locations. Instructions can be defined:

- By the address of the memory location to be maintained, referred to as operating *by VA*.
- By a mechanism that describes the location in the hardware of the cache, referred to as operating *by set/way*.

In addition, for instruction caches and branch predictors, there are instructions that invalidate all entries.

The following subsections define the terms used in the descriptions of the cache maintenance instructions:

- *Terminology for cache maintenance instruction operating by virtual address, VA*.
- *Terminology for cache maintenance instructions operating by set/way* on page G3-3958.
- *Terminology for Clean, Invalidate, and Clean and Invalidate instructions* on page G3-3958.

#### *Terminology for cache maintenance instruction operating by virtual address, VA*

In a VMSA implementation, the addresses used by the PE are VAs. When all applicable stages of translation are disabled, the VA is identical to the PA.

**Note**

For more information about memory system behavior when MMUs are disabled, see *The effects of disabling address translation stages on VMSAv8-32 behavior* on page G4-4000.

For the cache maintenance instruction, any instruction described as operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, the current system *Address Space IDentifier*, ASID.
- The current Security state.
- Whether the instruction was performed from Hyp mode, or from Non-secure EL1 state.
- For an instruction executed from a Non-secure EL1 state, the *Virtual Machine IDentifier*, VMID.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a data or unified cache maintenance instruction by VA, the operation cannot generate a Data Abort exception for a Domain fault or a Permission fault, except for the Permission fault cases described in:

- *Data cache maintenance instructions (DC*)* on page D3-1717.
- *Stage 2 fault on a stage 1 translation table walk* on page G4-4085.

For an instruction cache maintenance instruction by VA:

- It is IMPLEMENTATION DEFINED whether the operation can generate a Data Abort exception for a Translation fault or an Access flag fault.
- The operation cannot generate a Data Abort exception for a Domain fault or a Permission fault, except for the Permission fault case described in *Stage 2 fault on a stage 1 translation table walk* on page G4-4085.

For more information about these faults, see *MMU faults in AArch32 state* on page G4-4086.

### Terminology for cache maintenance instructions operating by set/way

Cache maintenance instruction that operate by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an instruction works on. These parameters are:

**Level** The cache level of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED and can be determined from the Cache Level ID register. See *CLIDR, Cache Level ID Register* on page G6-4257.

In the ARM architecture, the lower numbered levels are those closest to the PE. See *Memory hierarchy* on page E2-2358.

**Set** Each level of a cache is split up into a number of *sets*. Each set is a set of locations in a cache level to which an address can be assigned. Usually, the set number is an IMPLEMENTATION DEFINED function of an address.

In the ARM architecture, sets are numbered from 0.

**Way** The associativity of a cache is the number of locations in a set to which a specific address can be assigned. The *way* number specifies one of these locations.

In the ARM architecture, ways are numbered from 0.

**Note**

Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, ARM expects that most portable software will use only the cache maintenance instructions by set/way as single steps in a routine to perform maintenance on the entire cache.

### Terminology for Clean, Invalidate, and Clean and Invalidate instructions

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a PE that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.
2. Updates to memory locations by other observers that can access memory might not be visible to a PE that accesses a cache. This can occur when the cache contains an old, or *stale*, copy of the memory location that has been updated.

The *Clean* and *Invalidate* instructions address these two issues. The definitions of these instructions are:

**Clean** A cache clean instruction ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the instruction is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the instruction is performed, for example to the Point of Unification.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the Shareability domain of that memory location.

**Invalidate** A cache invalidate instruction ensures that updates made visible by observers that access memory at the point to which the invalidate is defined, are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate instruction that have been written by observers that access the cache, if those updates have not been cleaned from the cache since they were made.

If the address of an entry on which the invalidate instruction operates does not have a Normal Cacheable attribute, or if the cache is disabled, then an invalidate instruction also ensures that this address is not present in the cache.

**Note**

Entries for addresses with a Normal Cacheable attribute can be allocated to an enabled cache at any time, and so the cache invalidate instruction cannot ensure that the address is not present in an enabled cache.

**Clean and Invalidate**

A cache *clean and invalidate* instruction behaves as the execution of a clean instruction followed immediately by an invalidate instruction. Both instructions are performed to the same location.

The points to which a cache maintenance instruction can be defined differ depending on whether the instruction operates by VA or by set/way:

- For instructions operating by set/way, the point is defined to be to the next level of caching. For the All operations, the point is defined as the Point of Unification for each location held in the cache.
- For instruction operating by VA, two conceptual points are defined:

  **Point of Coherency (PoC)**

  For a particular VA, the PoC is the point at which all agents that can access memory are guaranteed to see the same copy of a memory location. In many cases, this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherence between memory system agents.

  **Note**

  The presence of system caches can affect the definition of the point of coherency as described in *System level caches* on page G3-3974.

  **Point of Unification (PoU)**

  The PoU for a PE is the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. In many cases, the Point of Unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

  The PoU for an Inner Shareable Shareability domain is the point by which the instruction and data caches and the translation table walks of all the PEs in that Inner Shareable Shareability domain are guaranteed to see the same copy of a memory location. Defining this point permits self-modifying software to ensure future instruction fetches are associated with the modified version of the software by using the standard correctness policy of:

  1. Clean data cache entry by address.
  2. Invalidate instruction cache entry by address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following fields in the CLIDR relate to these conceptual points:

**LoC, Level of Coherence**

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Coherency. The LoC value is a cache level, so, for example, if LoC contains the value 3:

- A clean to the Point of Coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.
- Level 4 cache is the first level that does not have to be maintained.

If the LoC field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Coherency.

If the LoC field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Coherency.

**LoUU, Level of Unification, uniprocessor**

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the PE. As with LoC, the LoUU value is a cache level.

If the LoUU field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification.

If the LoUU field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

**LoUIS, Level of Unification, Inner Shareable**

In any implementation:

- This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable Shareability domain. As with LoC, the LoUIS value is a cache level.
- If the LoUIS field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable Shareability domain.
- If the LoUIS field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

For more information, see *CLIDR, Cache Level ID Register* on page G6-4257.

## The ARMv8 abstraction of the cache hierarchy

The following subsections describe the ARMv8 abstraction of the cache hierarchy:

- *Cache maintenance instructions that operate by address*.
- *Cache maintenance instructions that operate by set/way* on page G3-3961.

### *Cache maintenance instructions that operate by address*

The address-based cache maintenance instructions are described as operating by VA. Each of these instructions is always qualified as being either:

- Performed to the Point of Coherency.
- Performed to the Point of Unification.

See *Terms used in describing the maintenance instructions* on page G3-3957 for definitions of Point of Coherency and Point of Unification, and more information about possible meanings of VA.

*AArch32 cache and branch predictor maintenance instructions* on page G3-3961 lists the address-based maintenance instructions.

The CTR holds minimum line length values for:

- The instruction caches.
- The data and unified caches.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These values support efficient invalidation of a range of addresses, because this value is the most efficient address stride to use to apply a sequence of address-based maintenance instructions to a range of addresses.

For the Invalidate data or unified cache line by VA instruction, the Cache Write-back Granule field of the CTR defines the maximum granule that a single invalidate instruction can invalidate. This meaning of the Cache Write-back Granule is in addition to its defining the maximum size that can be written back.

#### *Cache maintenance instructions that operate by set/way*

*AArch32 cache and branch predictor maintenance instructions* lists the set/way-based maintenance instructions. Some encodings of these instructions include a required field that specifies the cache level for the instruction:

- A clean instruction cleans from the level of cache specified through to at least the next level of cache, moving further from the PE.
- An invalidate instruction invalidates only at the level specified.

### G3.4.7 AArch32 cache and branch predictor maintenance instructions

The instruction and data cache maintenance instructions have the same functionality in AArch32 state and in AArch64 state. Table G3-3 shows these system instructions. Instructions that take an argument include Rt in the instruction description.

AArch32 state also provides branch predictor maintenance instructions.

**Note**

- In Table G3-3 the Point of Unification is the Point of Unification of the PE executing the cache maintenance instruction.
- In AArch32 state, all of the maintenance instructions are available from EL1 or higher.
- In AArch64 state, branch predictors are always invisible to software, and therefore AArch64 state does not provide any branch predictor maintenance instructions.

**Table G3-3 AArch32 System instructions for cache maintenance**

| Register | Instruction |
|---|---|
| Instruction cache maintenance instructions | |
| ICIALLUIS | Invalidate all to Point of Unification, Inner Shareable |
| ICIALLU | Invalidate all to Point of Unification |
| ICIMVAU, Rt | Invalidate by virtual address to Point of Unification |
| Data cache maintenance instructions | |
| DCIMVAC, Rt | Invalidate by virtual address to Point of Coherency |
| DCISW, Rt | Invalidate by set/way |
| DCCMVAC, Rt | Clean by virtual address to Point of Coherency |
| DCCSW, Rt | Clean by set/way |
| DCCMVAU, Rt | Clean by virtual address to Point of Unification |
| DCCIMVAC, Rt | Clean and invalidate by virtual address to Point of Coherency |
| DCCISW, Rt | Clean and invalidate by set/way |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G3-3 AArch32 System instructions for cache maintenance (continued)**

| Register | Instruction |
|---|---|
| Branch prediction maintenance instructions | |
| BPIMVA, Rt | Invalidate the virtual address from the branch predictors |
| BPIALLIS, Rt | Invalidate all entries from branch predictors, Inner Shareable |
| BPIALL, Rt | Invalidate all entries from branch predictors |

A DSB or DMB instruction intended to ensure the completion of cache or branch predictor maintenance instructions must have an access type of both loads and stores.

The following subsections give more information about these instructions:

- *AArch32 instruction cache maintenance instructions (IC*)*.
- *AArch32 data cache maintenance instructions (DC*)* on page G3-3963.
- *Branch predictors* on page G3-3963.
- *General requirements for the scope of cache and branch predictor maintenance instructions* on page G3-3965.
- *Effects of instructions that operate by virtual address to the Point of Coherency* on page G3-3966.
- *Effects of instructions that operate by virtual address but not to the Point of Coherency* on page G3-3966.
- *Effects of All and set/way maintenance instructions* on page G3-3967.
- *Effects of virtualization and security on the AArch32 cache maintenance instructions* on page G3-3967.
- *Boundary conditions for cache maintenance instructions* on page G3-3969.
- *Ordering of cache and branch predictor maintenance instructions* on page G3-3969.
- *Performing cache maintenance instructions* on page G3-3971.

### AArch32 instruction cache maintenance instructions (IC*)

Where an address argument for these instructions is required, it takes the form of a 32-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

All instruction cache maintenance instructions can execute in any order relative to other instruction cache maintenance instructions, data cache maintenance instructions, and loads and stores, unless a DSB is executed between the instructions.

An instruction cache maintenance instruction can complete at any time after it is executed, but is only guaranteed to be complete, and its effects visible to other observers, following a DSB instruction executed by the PE that executed the cache maintenance instruction.

See also *Ordering of cache and branch predictor maintenance instructions* on page G3-3969.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### AArch32 data cache maintenance instructions (DC*)

Where an address argument for these instructions is required, it takes the form of a 32-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

Data cache maintenance instructions that take a set/way/level argument take a 32-bit register.

A data or unified cache invalidation by virtual address instruction performed in a Non-secure EL1 mode must not change data in any location for which the stage 2 translation permissions do not permit write access. Where such a permission violation occurs, it is IMPLEMENTATION DEFINED whether:

- A stage 2 Permission fault is generated for the DCIMVAC operation.
- The DCIMVAC operation is upgraded to DCCIMVAC.

DCIMVAC and DCISW at EL1 is performed by the PE as clean and invalidate, that is DCCIMVAC or DCCISW if all of the following apply:

- EL2 is implemented.
- PL1&0 stage two address translation is enabled, meaning either:
  - EL2 is using AArch32 and the value of HCR.VM is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.VM is 1.
- EL3 is not implemented, or either:
  - EL3 is using AArch32 and the value of SCR.NS is 1.
  - EL3 is using AArch64 and the value of SCR_EL3.NS is 1.

**Note**

Similarly, DCIMVAC and DCISW at EL1 must be performed as clean and invalidate, that is DCCIMVAC and DCCISW at EL1 when EL1 is using AArch32, if all of the following apply:

- EL2 is implemented.
- EL2 is using AArch32 and HCR.VM is set to the value of 1, or EL2 is using AArch64 and HCR_EL2.VM is set to the value of 1.
- EL3 is using AArch32 and SCR.NS is set to the value of 1, or EL3 is using AArch64 and SCR.NS is set to the value of 1, or EL3 is not implemented.

If a data cache maintenance by set/way instruction specifies a set, way, or level argument that is larger than the value supported by the implementation then the instruction is CONSTRAINED UNPREDICTABLE, see *Out of range values of the Set/Way/Index fields in cache maintenance instructions* on page K1-5432 or the instruction description.

If a memory fault that sets FAR for the translation regime applicable for the cache maintenance instruction is generated from a data cache maintenance instruction, the FAR holds the address specified in the register argument of the instruction.

See also *Ordering of cache and branch predictor maintenance instructions* on page G3-3969.

### Branch predictors

In AArch32 state it is IMPLEMENTATION DEFINED whether branch prediction is architecturally visible. This means that under some circumstances software must perform branch predictor maintenance to avoid incorrect execution caused by out-of-date entries in the branch predictor. For example, to ensure correct operation it might be necessary to invalidate branch predictor entries on a change to instruction memory, or a change of instruction address mapping. For more information, see *Specific requirements for branch predictor maintenance instructions* on page G3-3964.

An invalidate all operation on the branch predictor ensures that any location held in the branch predictor has no functional effect on execution. An invalidate branch predictor by VA instruction operates on the address of the branch instruction, but can affect other branch predictor entries.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The architecture does not make visible the range of addresses in a branch predictor to which the invalidate operation applies. This means the address used in the invalidate by VA operation must be the address of the branch to be invalidated.

If branch prediction is architecturally visible, an instruction cache invalidate all operation also invalidates all branch predictors.

See also *Ordering of cache and branch predictor maintenance instructions* on page G3-3969.

### *Specific requirements for branch predictor maintenance instructions*

If, for a given translation regime and a given ASID and VMID as appropriate, the instructions at any virtual address change, then branch predictor maintenance instructions must be performed to invalidate entries in the branch predictor, to ensure that the change is visible to subsequent execution. This maintenance is required when writing new values to instruction locations. It can also be required as a result of any of the following situations that change the translation of a virtual address to a physical address, if, as a result of the change to the translation, the instructions at the virtual addresses change:

- For any translation regime other than the Non-secure PL1&0 translation regime, enabling or disabling stage 1 translations.
- For the Non-secure PL1&0 translation regime:
  - When stage 2 translations are enabled, enabling or disabling stage 1 translations unless accompanied by a change of VMID.
  - When stage 2 translations are disabled, enabling or disabling stage 1 translations.
  - Enabling or disabling stage 2 translations.
- Writing new mappings to the translation tables.
- Any change to the TTBR0, TTBR1, or TTBCR registers, unless:
  - For a change to the Secure PL1&0 translation regime, the change is accompanied by a change to the ASID.
  - For a change to the stage 1 translations of the Non-secure PL1&0 translation regime, the change is accompanied by a change to the ASID or a change to the VMID.
- Any change to the VTTBR or VTCR registers, unless accompanied by a change to the VMID.

**Note**

Invalidation is not required if the changes to the translations are such that the instructions associated with the non-faulting translations of a virtual address, for a given translation regime and a given ASID and VMID, as appropriate, remain unchanged throughout the sequence of changes to the translations. Examples of translation changes to which this applies are:

- Changing a valid translation to a translation that generates an MMU fault.
- Changing a translation that generates an MMU fault to a valid translation.

Failure to invalidate entries might give UNPREDICTABLE results, caused by the execution of old branches. For more information, see *Ordering of cache and branch predictor maintenance instructions* on page G3-3969.

**Note**

- In ARMv8, there is no requirement to use the branch predictor maintenance operations to invalidate the branch predictor after:
  - Changing the ContextID or VMID.
  - A cache maintenance instruction that is identified as also flushing the branch predictors, see *AArch32 cache and branch predictor maintenance instructions* on page G3-3961.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Cache maintenance instructions, functional group* on page G4-4169 shows the branch predictor maintenance operations in a VMSA implementation.

#### *Behavior of the branch predictors at reset*

In ARMv8:

- If branch predictors are not architecturally invisible:
  - The branch predictors reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
  - The branch predictors are disabled at reset.
- An implementation can require the use of a specific branch predictor initialization routine to invalidate the branch predictor storage array before it is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, but the routine must be documented clearly as part of the documentation of the device.
- ARM recommends that whenever an invalidation routine is required, it is based on the ARMv8 branch predictor maintenance operations.

Similar rules apply:

- To cache behavior, see *Behavior of caches at reset* on page G3-3955.
- To TLB behavior, see *TLB behavior at reset* on page G4-4059.

### General requirements for the scope of cache and branch predictor maintenance instructions

The ARMv8 specification of the cache maintenance and branch predictor instructions describes what each instruction is guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements described in *General behavior of the caches* on page G3-3952, *Behavior of caches at reset* on page G3-3955, and *Preloading caches* on page E2-2361.

This means that as a side-effect of a cache maintenance instruction:

- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

As a side-effect of a branch predictor maintenance instruction, any entry in the branch predictor might be invalidated.

**Note**

ARM recommends that, for best performance, such side-effects are kept to a minimum. ARM strongly recommends that the side-effects of operations performed in Non-secure state do not have a significant performance impact on execution in Secure state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Effects of instructions that operate by virtual address to the Point of Coherency

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of other PEs in the Shareability domain described by the Shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of all PEs in the Outer Shareable Shareability domain of the PE on which the instruction is operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the Point of Coherency.

**Table G3-4 PEs affected by cache maintenance instructions to the Point of Coherency**

| Shareability | PEs affected | Effective to |
|---|---|---|
| Non-shareable | The PE performing the operation | The Point of Coherency of the entire system |
| Inner Shareable | All PEs in the same Inner Shareable Shareability domain as the PE performing the operation | The Point of Coherency of the entire system |
| Outer Shareable | All PEs in the same Outer Shareable Shareability domain as the PE performing the operation | The Point of Coherency of the entire system |

### Effects of instructions that operate by virtual address but not to the Point of Coherency

The following instruction operate by virtual address but not to the Point of Coherency:

- Clean data or unified cache line by MVA to the Point of Unification, DCCMVAU.
- Invalidate instruction cache line by MVA to Point of Unification, ICIMVAU.
- Invalidate by MVA from branch predictors, BPIMVA.

For these instructions, Table G3-5 shows how, for a VA in a Normal or Device memory location, the Shareability attribute of the VA determines the minimum set of PEs affected, and the point to which the instruction must be effective.

**Table G3-5 PEs affected by cache maintenance instructions to the Point of Unification**

| Shareability | PEs affected | Effective to |
|---|---|---|
| Non-shareable | The PE executing the instruction | The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, on the PE executing the instruction |
| Inner Shareable<br>or<br>Outer Shareable | All PEs in the same Inner Shareable Shareability domain as the PE executing the instruction | The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, of all PEs in the same Inner Shareable Shareability domain as the PE executing the instruction |

**Note**

The set of PEs guaranteed to be affected is never greater than the PEs in the Inner Shareable Shareability domain containing the PE executing the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Effects of All and set/way maintenance instructions

The ICIALLU, BPIALL and DC* set/way instructions apply only to the caches and branch predictors of the PE that performs the instruction. If the branch predictors are architecturally-visible, ICIALLU also performs a BPIALL operation.

The ICIALLUIS and BPIALLIS instructions can affect the caches and branch predictors of all PEs in the same Inner Shareable Shareability domain as the PE that performs the instruction. If the branch predictors are architecturally-visible, ICIALLUIS also performs a BPIALLIS operation. These instructions have an effect to the Point of Unification of instruction cache fills, data cache fills, and write-backs, and translation table walks, of all PEs in the same Inner Shareable Shareability domain.

**Note**

The possible presence of system caches, as described in *System level caches* on page G3-3974, means architecture does not guarantee that all levels of cache can be maintained using set/way instructions.

### Effects of virtualization and security on the AArch32 cache maintenance instructions

Each Security state has its own physical address space, and therefore cache entries are associated with physical address space. In addition, cache maintenance and branch predictor instructions performed in Non-secure state have to take account of:

- Whether the instruction was performed at EL1 or at EL2.
- For instructions that operate by VA, the current VMID.

Table G3-6 shows the effects of virtualization and security on these maintenance instructions.

**Table G3-6 Effects of virtualization and security on the AArch32 cache maintenance instructions**

| Cache maintenance instructions | Security state | Specified entry |
|---|---|---|
| Data or unified cache maintenance instructions | | |
| Invalidate, Clean, or Clean and Invalidate by VA: DCIMVAC, DCCMVAC, DCCMVAU, DCCIMVAC | Either | All lines that hold the PA that, in the current translation regime, are mapped to by the combination of all of:<br>• The specified VA.<br>• For an instruction executed at EL1, the current ASID if the location is mapped to by a non-global page.<br>• For an instruction executed at Non-secure EL1, the current VMID[a]. |
| Invalidate, Clean, or Clean and Invalidate by set/way: DCISW, DCCSW, DCCISW | Non- secure | Line specified by set/way provided that the entry comes from the Non-secure PA space. |
| | Secure | Line specified by set/way regardless of the PA space that the entry has come from. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G3-6 Effects of virtualization and security on the AArch32 cache maintenance instructions (continued)**

| Cache maintenance instructions | Security state | Specified entry |
|---|---|---|
| Instruction cache maintenance instructions | | |
| Invalidate by VA: ICIMVAU | Either | All lines corresponding to the specified VA[b] in the current translation regime and:<br>• For an instruction executed at EL1 or EL0, the current ASID.<br>• For an instruction executed at Non-secure EL1 or Non-secure EL0, the current VMID[a]. |
| Invalidate All: ICIALLU, ICIALLUIS | | • Can invalidate any unlocked entry in the instruction cache.<br>• Are required to invalidate any entries relevant to the software component that executed it. The Non-secure and Secure descriptions give more information:<br>**Non-secure**<br>An instruction executed at EL1 must operate on all instruction cache lines that contain entries associated with the current virtual machine, meaning any entry with the current VMID.[a]<br>An instruction executed at EL2 must operate on all instruction cache lines that contain entries that can be accessed from Non-secure state.<br>**Secure** The instruction must invalidate all instruction cache lines. |
| Branch predictor operations | | |
| Invalidate by VA: BPIMVA | Either | All lines that, in the current translation regime, are mapped to by the combination of: all of:<br>• The specified VA.<br>• For an instruction executed at EL1 or EL0, the current ASID if the location is mapped to by a non-global page.<br>• For an instruction executed at Non-secure EL1 or EL0, the current VMID[a]. |
| Invalidate all: BPIALL, BPIALLIS | | • Can invalidate any unlocked entry in the instruction cache.<br>• Are required to invalidate any entries relevant to the software component that executed it. The Non-secure and Secure descriptions give more information. |

a. Dependencies on the VMID apply even when HCR_EL2.VM is set to 0. However, VTTBR_EL2.VMID resets to zero, meaning there is a valid VMID from reset.

b. The type of instruction cache used affects the interpretation of the specified entries in this table such that:
• For a PIPT instruction cache, the cache maintenance applies to all entries whose physical address corresponds to the specified address.
• For a VIVT instruction cache, the cache maintenance applies to entries whose virtual tag and physical address corresponds to the specified address.
• For an ASID and VMID tagged VIVT instruction cache, the cache maintenance applies to entries whose virtual address corresponds to the specified address.
For information of types of instruction cache, see *Instruction caches* on page G4-4074.

For locked entries and entries that might be locked, the behavior of cache maintenance instructions described in *The interaction of cache lockdown with cache maintenance instructions* on page G3-3972 applies.

With an implementation that generates aborts if entries are locked or might be locked in the cache, when the use of lockdown aborts is enabled, these aborts can occur on any cache maintenance instructions.

In an implementation that includes EL2:

- The architecture does not require cache cleaning when switching between virtual machines. Cache invalidation by set/way must not present an opportunity for one virtual machine to corrupt state associated with a second virtual machine. To ensure this requirement is met, Non-secure clean by set/way operations can be upgraded to clean and invalidate by set/way.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The AArch32 Data cache invalidate instructions DCIMVAC and DCISW, perform a cache clean as well as a cache invalidate, meaning they operate as DCCIMVAC and DCCISW respectively, if both of the following apply:
  - The value of HCR.VM is 1.
  - The instruction is executed in Non-secure state, or EL3 is not implemented.
- The AArch32 Data cache invalidate by set/way instruction DCISW performs a cache clean as well as a cache invalidate, meaning it operates as DCCISW, if both of the following apply:
  - The value of HCR.SWIO is 1.
  - The instruction is executed in Non-secure state, or EL3 is not implemented.
- When the value of HCR.FB is 1, TLB and instruction cache invalidate instructions executed in the Non-secure EL1 Exception level are broadcast across the Inner Shareable domain. When Non-secure EL1 is using AArch32, this applies to the TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL, and ICIALLU instructions. This means the instruction is upgraded to the corresponding Inner Shareable instruction, for example ICIALLU is upgraded to ICIALLUIS, and BPIALL is upgraded to BPIALLIS.

For more information about the cache maintenance instructions, see *The ARMv8 cache maintenance functionality* on page G3-3957, *AArch32 cache and branch predictor maintenance instructions* on page G3-3961, and Chapter G4 *The AArch32 Virtual Memory System Architecture*.

### Boundary conditions for cache maintenance instructions

Cache maintenance instructions operate on the caches when the caches are enabled or when they are disabled.

For address-based cache maintenance instructions, the instructions operate on the caches regardless of the memory type and cacheability attributes marked for the memory address in the VMSA translation table entries. This means that the effects of the cache maintenance instructions can apply regardless of:

- Whether the address accessed:
  - Is Normal memory or Device memory.
  - Has the Cacheable attribute or the Non-cacheable attribute.
- Any applicable domain control of the address accessed.
- The access permissions for the address accessed, other than the effect of the stage two write permission on data or unified cache invalidation instructions.

### Ordering of cache and branch predictor maintenance instructions

The following rules describe the effect of the memory order model on the cache and branch predictor maintenance instructions:

- All cache and branch predictor maintenance instructions that do not specify an address execute, relative to each other, in program order.

  All cache and branch predictor instructions that specify an address:
  - Execute in program order relative to all cache and branch predictor operations that do not specify an address.
  - Execute in program order relative to all cache and branch predictor operations that specify the same address.
  - Can execute in any order relative to cache and branch predictor operations that specify a different address.
- Where a cache maintenance or branch predictor instruction appears in program order before a change to the translation tables, the architecture guarantees that the cache or branch predictor maintenance instruction uses the translations that were visible before the change to the translation tables.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Where a change of the translation tables appears in program order before a cache maintenance or branch predictor instruction, software must execute the sequence outlined in *Ordering and completion of TLB maintenance instructions* on page G4-4064 before performing the cache or branch predictor maintenance instruction, to ensure that the maintenance operation uses the new translations.
- A `DMB` instruction causes the effect of all data or unified cache maintenance instructions appearing in program order before the `DMB` to be visible to all explicit load and store operations appearing in program order after the `DMB`.

  Also, a `DMB` instruction ensures that the effects of any data or unified cache maintenance instruction appearing in program order before the `DMB` are observable by any observer in the same required Shareability domain before any data or unified cache maintenance or explicit memory operations appearing in program order after the `DMB` are observed by the same observer. Completion of the `DMB` does not guarantee the visibility of all data to other observers. For example, all data might not be visible to a translation table walk, or to instruction fetches.
- A `DSB` is required to guarantee the completion of all cache maintenance instruction that appear in program order before the `DSB` instruction.
- A context synchronization operation is required to guarantee the effects of any branch predictor maintenance operation. This means a context synchronization operation causes the effect of all completed branch predictor maintenance operations appearing in program order before the context synchronization operation to be visible to all instructions after the context synchronization operation.

  **Note**

  See *Context synchronization operation* in the *Glossary* for the definition of this term.

  This means that, if a branch instruction appears after an invalidate branch predictor operation and before any context synchronization operation, it is UNPREDICTABLE whether the branch instruction is affected by the invalidate. Software must avoid this ordering of instructions, because it might cause UNPREDICTABLE behavior.
- Any data or unified cache maintenance instruction by VA must be executed in program order relative to any explicit load or store on the same PE to an address covered by the VA of the cache instruction if that load or store is to Normal Cacheable memory. The order of memory accesses that result from the cache maintenance instruction, relative to any other memory accesses to Normal Cacheable memory, are subject to the memory ordering rules. For more information, see *Memory ordering* on page E2-2372.

  Any data or unified cache maintenance instruction by VA can be executed in any order relative to any explicit load or store on the same PE to an address covered by the VA of the cache maintenance instruction if that load or store is not to Normal Cacheable memory.
- There is no restriction on the ordering of data or unified cache maintenance instruction by VA relative to any explicit load or store on the same PE where the address of the explicit load or store is not covered by the VA of the cache instruction. Where the ordering must be restricted, a `DMB` instruction must be inserted to enforce ordering.
- There is no restriction on the ordering of a data or unified cache maintenance instruction by set/way relative to any explicit load or store on the same PE. Where the ordering must be restricted, a `DMB` instruction must be inserted to enforce ordering.
- Software must execute a context synchronization operation after the completion of an instruction cache maintenance instruction, to guarantee that the effect of the maintenance instruction is visible to any instruction fetch.

A `DSB` or `DMB` instruction intended to ensure the completion of cache maintenance instructions or branch predictor instructions must have an access type of both loads and stores.

The scope of instruction cache maintenance depends on the type of the instruction cache. For more information see *Instruction caches* on page G4-4074.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Example G3-1 Cache cleaning operations for self-modifying code**

The sequence of cache cleaning operations for a line of self-modifying code on a uniprocessor system is:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line
; instead of STR for a 16-bit instruction.
       STR Rt, [Rn]
       DCCMVAU Rn          ; Clean data cache by MVA to point of unification (PoU)
       DSB                 ; Ensure visibility of the data cleaned from cache
       ICIMVAU Rn          ; Invalidate instruction cache by MVA to PoU
       BPIMVA  Rn          ; Invalidate branch predictor by MVA to PoU
       DSB                 ; Ensure completion of the invalidations
       ISB                 ; Synchronize the fetched instruction stream
```

## Performing cache maintenance instructions

To ensure all cache lines in a block of address space are maintained through all levels of cache ARM strongly recommends that software:

- For data or unified cache maintenance, uses the CTR.DMinLine value to determine the loop increment size for a loop of data cache maintenance by VA instructions.
- For instruction cache maintenance, uses the CTR.IMinLine value to determine the loop increment size for a loop of instruction cache maintenance by VA instructions.

### *Example code for cache maintenance instructions*

The cache maintenance instructions by set/way can be used to clean or invalidate, or both, the entirety of one or more levels of cache attached to a PE. However, unless all PEs attached to the caches regard all memory locations as Non-cacheable, it is not possible to prevent locations being allocated into the cache during such a sequence of the cache maintenance instructions.

**Note**

Since the set/way instructions are performed only locally, there is no guarantee of the atomicity of cache maintenance between different PEs, even if those different PEs are each executing the same cache maintenance instructions at the same time. Since any cacheable line can be allocated into the cache at any time, it is possible for a cache line to migrate from an entry in the cache of one PE to the cache of a different PE in a way that means the cache line is not affected by set/way based cache maintenance. Therefore, ARM strongly discourages the use of set/way instructions to manage coherency in coherent systems. The expected use of the cache maintenance instructions that operate by set/way is limited to the cache maintenance associated with the powerdown and powerup of caches, if this is required by the implementation.

The limitations of cache maintenance by set/way mean maintenance by set/way does not happen on multiple PEs, and cannot be made to happen atomically for each address on each PE. Therefore in multiprocessor or multithreaded systems, the use of cache maintenance by set/way to clean, or clean and invalidate, the entire cache for coherency management with very large buffers or with buffers with unknown address can fail to provide the expected coherency results because of speculation by other PEs, or possibly by other threads. The only way that these instructions can be used in this way is to first ensure that all PEs that might cause speculative accesses to caches that need to be maintained are not capable of generating speculative accesses. This can be achieved by ensuring that those PEs have no memory locations marked as cacheable. Such an approach can have very large system performance effects, and ARM advises implementers to use hardware coherency mechanisms in systems where this will be an issue.

*System level caches* on page G3-3974 refers to other limitations of cache maintenance by set/way.

The following example code for cleaning a data or unified cache to the Point of Coherency illustrates a generic mechanism for cleaning the entire data or unified cache to the Point of Coherency.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    MRC p15, 1, R0, c0, c0, 1   ; Read CLIDR into R0
    ANDS R3, R0, #0x07000000
    MOV R3, R3, LSR #23         ; Cache level value (naturally aligned)
    BEQ Finished
    MOV R10, #0
Loop1
    ADD R2, R10, R10, LSR #1    ; Work out 3 x cache level
    MOV R1, R0, LSR R2          ; bottom 3 bits are the Cache type for this level
    AND R1, R1, #7              ; get those 3 bits alone
    CMP R1, #2
    BLT Skip                    ; no cache or only instruction cache at this level
    MCR p15, 2, R10, c0, c0, 0  ; write CSSELR from R10
    ISB                         ; ISB to sync the change to the CCSIDR
    MRC p15, 1, R1, c0, c0, 0   ; read current CCSIDR to R1
    AND R2, R1, #7              ; extract the line length field
    ADD R2, R2, #4              ; add 4 for the line length offset (log2 16 bytes)
    MOV R4, #0x3FF
    ANDS R4, R4, R1, LSR #3     ; R4 is the max number on the way size (right aligned)
    CLZ R5, R4                  ; R5 is the bit position of the way size increment
    MOV R9, R4                  ; R9 working copy of the max way size (right aligned)
Loop2
    MOV R7, #0x00007FFF
    ANDS R7, R7, R1, LSR #13    ; R7 is the max number of the index size (right aligned)
Loop3
    ORR R11, R10, R9, LSL R5    ; factor in the way number and cache number into R11
    ORR R11, R11, R7, LSL R2    ; factor in the index number
    MCR p15, 0, R11, c7, c10, 2 ; DCCSW, clean by set/way
    SUBS R7, R7, #1             ; decrement the index
    BGE Loop3
    SUBS R9, R9, #1             ; decrement the way number
    BGE Loop2
Skip
    ADD R10, R10, #2            ; increment the cache number
    CMP R3, R10
    DSB                         ; ensure completion of previous cache maintenance instruction
    BGT Loop1
Finished
```

Similar approaches can be used for all cache maintenance instructions.

### G3.4.8 Cache lockdown

The concept of an entry locked in a cache is allowed, but not architecturally defined. How lockdown is achieved is IMPLEMENTATION DEFINED and might not be supported by:

- An implementation.
- Some memory attributes.

An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

#### The interaction of cache lockdown with cache maintenance instructions

The interaction of cache lockdown and cache maintenance instructions is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance instruction on a locked cache line must comply with the following general rules:

- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by set/way, DCCSW.
  - Cache invalidate by set/way, DCISW.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Cache clean and invalidate by set/way, DCISW.
— Instruction cache invalidate all, ICIALLU and ICIALLUIS.

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is not invalidated from the cache.
2. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the fault status code defined for this purpose. See *Data Abort exception* on page G1-3821.

This permits a usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked.

The effect of the following instructions is IMPLEMENTATION DEFINED:

- Cache clean by virtual address, DCCMVAC and DCCMVAU.
- Cache invalidate by virtual address, DCIMVAC.
- Cache clean and invalidate by virtual address, DCCIMVAC.

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is invalidated from the cache. For the clean and invalidate instructions, the entry must be cleaned before it is invalidated.
2. If the instruction specified an invalidation, a locked entry is not invalidated from the cache. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the fault status code defined for this purpose. See DFSR or HSR.

In an implementation that includes EL2, if HCR.TIDCP is set to 1, any exception relating to lockdown of an entry associated with Non-secure memory is routed to EL2.

**Note**

An implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other permitted alternatives for the locked entries.

ARM recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use architecturally-defined instructions. This minimizes the number of customized instructions required.

In addition, an implementation that uses an abort to handle cache maintenance instructions for entries that might be locked must provide a mechanism that ensures that no entries are locked in the cache.

The reset setting of the cache must be that no cache entries are locked.

### *Additional cache functions for the implementation of lockdown*

An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED space. See *IMPLEMENTATION DEFINED registers, functional group* on page G4-4179.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G3.4.9 System level caches

The system level architecture might define further aspects of the software view of caches and the memory model that are not defined by the ARMv8 architecture. These aspects of the system level architecture can affect the requirements for software management of caches and coherency. For example, a system design might introduce additional levels of caching that cannot be managed using the architecturally-defined maintenance instructions. Such caches are referred to as *system caches*.

Conceptually, three classes of system cache can be envisaged:

1. System caches which lie before the point of coherency and cannot be managed by cache maintenance instructions. Such systems fundamentally undermine the concept of cache maintenance instructions operating to the point of coherency, as they imply the use of non-architecture mechanisms to manage coherency. The use of such systems in the ARM architecture is explicitly prohibited.

2. System caches which lie before the point of coherency and can be managed by cache maintenance by address instructions that apply to the point of coherency, but cannot be managed by cache maintenance by set/way instructions. Where maintenance of the entire system cache must be performed, as is the case for power management, it must be performed using non-architectural mechanisms.

3. System caches which lie beyond the point of coherency and so are invisible to software. The management of such caches is outside the scope of architecture.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G3.5 System register support for IMPLEMENTATION DEFINED memory features

The VMSAv8-32 defines the following registers for describing IMPLEMENTATION DEFINED features of the memory system:

- The TCM Type Register,TCMTR, in CP15 c0, must be implemented. The following conditions apply to this register:
  - — If no TCMs are implemented, the TCMTR indicates zero-size TCMs.
  - — If bits[31:29] are `0b100`, the format of the rest of the register is IMPLEMENTATION DEFINED. This value indicates that the implementation includes TCMs that do not follow the legacy usage model. Other fields in the register might give more information about the TCMs.
- The CP15 c9 encoding space with `<CRm>` = {0-2, 5-7} is IMPLEMENTATION DEFINED for all values of `<opc2>` and `<opc1>`. This space is reserved for branch predictor, cache and TCM functionality, for example maintenance, override behaviors and lockdown.

  For more information, see *VMSAv8-32 CP15 c9 register summary* on page G4-4144.
- In a VMSAv8 implementation, part of the CP15 c10 encoding space is IMPLEMENTATION DEFINED and reserved for TLB functionality, see *TLB lockdown* on page G4-4059.
- The CP15 c11 encoding space with `<CRm>` = {0-8, 15} is IMPLEMENTATION DEFINED for all values of `<opc2>` and `<opc1>`. This space is reserved for DMA operations to and from the TCMs.

  For more information, see *VMSAv8-32 CP15 c11 register summary* on page G4-4145.

In addition, CP15 c15 is reserved for IMPLEMENTATION DEFINED registers, and can provide additional registers for the memory system. For more information, see *VMSAv8-32 CP15 c15 register summary* on page G4-4147.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G3.6 External aborts

The ARM architecture defines external aborts as errors that occur in the memory system, other than those that are detected by the MMU or Debug hardware. External aborts include parity or ECC errors detected by the caches or other parts of the memory system. For example, an uncorrectable parity or ECC failure on a Level 2 Memory structure might generate an external abort.

An external abort is one of:

- Synchronous.
- Precise asynchronous.
- Imprecise asynchronous.

For more information, see *Exception terminology* on page G1-3748.

The ARM architecture does not provide any method to distinguish between precise asynchronous and imprecise asynchronous aborts.

VMSAv8-32 permits external aborts on data accesses, translation table walks, and instruction fetches to be either synchronous or asynchronous. The reported fault code identifies whether the external abort is synchronous or asynchronous.

It is IMPLEMENTATION DEFINED which external aborts, if any, are supported.

In AArch32 state:

- Asynchronous aborts are taken as asynchronous Data Abort exceptions.
- Synchronous external aborts:
  - On data accesses are taken as synchronous Data Abort exceptions.
  - On instruction fetches, or prefetches, are taken as synchronous Prefetch Abort exceptions.

See also:

- *External abort on a translation table walk* on page G4-4088.
- *Handling exceptions that are taken to an Exception level using AArch32* on page G1-3774.

Normally, external aborts are rare. An imprecise asynchronous external abort is likely to be fatal to the process that is running. ARM recommends that implementations make external aborts precise wherever possible.

The following subsections give more information about possible external aborts:

- *External abort on instruction fetch*.
- *External abort on data read or write*.
- *Provision for classification of external aborts* on page G3-3977.
- *Parity or ECC error reporting* on page G3-3977.

The section *Exception reporting in a VMSAv8-32 implementation* on page G4-4091 describes the reporting of external aborts.

### G3.6.1 External abort on instruction fetch

An external abort on an instruction fetch can be either synchronous or asynchronous. A synchronous external abort on an instruction fetch is taken precisely.

An implementation can report the external abort asynchronously from the instruction that it applies to. In such an implementation these aborts behave essentially as interrupts. The aborts are masked when PSTATE.A is set to 1, otherwise they are reported using the Data Abort exception.

### G3.6.2 External abort on data read or write

Externally-generated errors during a data read or write can be either synchronous or asynchronous.

An implementation can report the external abort asynchronously from the instruction that generated the access. In such an implementation these aborts behave essentially as interrupts. The aborts are masked when PSTATE.A is set to 1, otherwise they are reported using the Data Abort exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G3.6.3 Provision for classification of external aborts

For a synchronous external abort taken to a privileged mode other than Hyp mode, an implementation can use the DFSR.ExT and IFSR.ExT bits to provide more information about synchronous external aborts:

- DFSR.ExT provides an IMPLEMENTATION DEFINED classification of synchronous external aborts on data accesses.
- IFSR.ExT provides an IMPLEMENTATION DEFINED classification of synchronous external aborts on instruction accesses.

For a synchronous external abort taken to Hyp mode, the HSR.EA, ISS[9] bit, provides an IMPLEMENTATION DEFINED classification of external aborts.

For all aborts other than synchronous external aborts these bits return a value of 0.

### G3.6.4 Parity or ECC error reporting

The ARM architecture supports the reporting of both synchronous and asynchronous parity or ECC errors from the cache systems. It is IMPLEMENTATION DEFINED what parity or ECC errors in the cache systems, if any, result in synchronous or asynchronous parity or ECC errors.

A fault code is defined for reporting parity or ECC errors, see *Exception reporting in a VMSAv8-32 implementation on page G4-4091*. However when parity or ECC error reporting is implemented it is IMPLEMENTATION DEFINED whether a parity or ECC error is reported using the assigned fault code, or using another appropriate encoding.

For all purposes other than the fault status encoding, parity or ECC errors are treated as external aborts.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G3.7 Memory barrier instructions

*Memory barriers* on page E2-2375 describes the memory barrier instructions. This section describes the system level controls of those instructions.

### G3.7.1 EL2 control of the Shareability of data barrier instructions executed at EL0 or EL1

In an implementation that includes EL2 and supports Shareability limitations on the data barrier instructions, the HCR.BSU field can upgrade the required Shareability of an instruction that is executed at EL0 or EL1 in Non-secure state. Table G3-7 shows the encoding of this field:

**Table G3-7 EL2 control of Shareability of barrier instructions executed at EL0 or EL1**

| HCR.BSU | Minimum Shareability of barrier instructions |
|---|---|
| 00 | No effect, Shareability is as specified by the instruction |
| 01 | Inner Shareable |
| 10 | Outer Shareable |
| 11 | Full system |

For an instruction executed at EL0 or EL1 in Non-secure state, Table G3-8 shows how the HCR.BSU is combined with the Shareability specified by the argument of the `DMB` or `DSB` instruction to give the scope of the instruction:

**Table G3-8 Effect of the HCR_EL2.BSU on barrier instructions executed at Non-secure EL1 or EL1**

| Shareability specified by the `DMB` or `DSB` argument | HCR.BSU | Resultant Shareability |
|---|---|---|
| Full system | Any | Full system |
| Outer Shareable | `00`, `01`, or `10` | Outer Shareable |
| | `11`, Full system | Full system |
| Inner Shareable | `00` or `01` | Inner Shareable |
| | `10`, Outer Shareable | Outer Shareable |
| | `11`, Full system | Full system |
| Non-shareable | `00`, No effect | Non-shareable |
| | `01`, Inner Shareable | Inner Shareable |
| | `10`, Outer Shareable | Outer Shareable |
| | `11`, Full system | Full system |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G3.8 Pseudocode description of general memory system instructions

This section contains the following pseudocode describing general memory operations:


- *Memory data type definitions*.
- *Basic memory access* on page G3-3980.
- *Aligned memory access* on page G3-3980.
- *Unaligned memory access* on page G3-3981.
- *Exclusive monitors operations* on page G3-3982.
- *Access permission checking* on page G3-3984.
- *Abort exceptions* on page G3-3985.
- *Memory barriers* on page G3-3987.


### G3.8.1 Memory data type definitions

This section describes the memory data type definitions.

The address descriptor type is defined as follows:

```
type AddressDescriptor is (
    FaultRecord      fault,      // fault.type indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    bits(64)         vaddress
)
```

The full address type is defined as follows:

```
type FullAddress is (
    bits(48) physicaladdress,
    bit      NS                  // '0' = Secure, '1' = Non-secure
)
```

The memory attributes types are defined is as follows:

```
type MemoryAttributes is (
    MemType      type,

    DeviceType   device,     // For Device memory types
    MemAttrHints inner,      // Inner hints and attributes
    MemAttrHints outer,      // Outer hints and attributes

    boolean      shareable,
    boolean      outershareable
)
```

The memory type is defined as follows.

```
enumeration MemType {MemType_Normal, MemType_Device};
```

The Device memory types are defined as follows:

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

For Normal memory, the inner and outer attributes are defined is as follows:

```
type MemAttrHints is (
    bits(2) attrs,  // The possible encodings for each attributes field are as below
    bits(2) hints,  // The possible encodings for the hints are below
    boolean transient
)
```

The cacheability attributes are defined as follows:

```
constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The allocation hints are defined as follows:

```
constant bits(2) MemHint_No = '00';     // No allocate
constant bits(2) MemHint_WA = '01';     // Write-allocate, Read-no-allocate
constant bits(2) MemHint_RA = '10';     // Read-allocate, Write-no-allocate
constant bits(2) MemHint_RWA = '11';    // Read-allocate and Write-allocate
```

The access permissions type is defined as follows:

```
type Permissions is (
    bits(3) ap,   // Access permission bits
    bit     xn,   // Execute-never bit
    bit     pxn   // Privileged execute-never bit
)
```

### G3.8.2 Basic memory access

The two `_Mem[]` accessors, Non-assignment (memory read) and Assignment (memory write), are the operations that perform single-copy atomic, aligned, little-endian memory accesses of `size` bytes to or from the underlying physical memory array of bytes.

```
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccType acctype];

_Mem[AddressDescriptor desc, integer size, AccType acctype] = bits(8*size) value;
```

The functions address the array using `desc.paddress` which supplies:

- A 48-bit physical address.
- A single NS bit to select between Secure and Non-secure parts of the array.

The `AccType` parameter describes the access type, such as normal, exclusive, ordered, and streaming. For a definition of `AccType`, see *Address space* on page E2-2356.

The actual implemented array of memory might be smaller than the $2^{48}$ bytes implied. In this case the scheme for aliasing is IMPLEMENTATION DEFINED, or some parts of the address space might give rise to external aborts or a System Error.

The attributes in `memaddrdesc.memattrs` are used by the memory system to determine caching and ordering behaviors as described in *Memory types and attributes* on page E2-2382, *Memory ordering* on page E2-2372, and *Atomicity in the ARM architecture* on page E2-2368.

`PAMax()` returns the IMPLEMENTATION DEFINED size of the physical address.

```
integer PAMax();
```

**Note**

In AArch32 a translation regime can never generate more than 40 bits of an address.

### G3.8.3 Aligned memory access

The `MemSingle[]` functions make atomic, little-endian accesses of `size` bytes.

```
// AArch32.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    value = _Mem[memaddrdesc, size, acctype];
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8)
value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.mematttrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    _Mem[memaddrdesc, size, acctype] = value;
    return;
```

### G3.8.4 Unaligned memory access

See *Unaligned data access* on page E2-2363 for details of the SCTLR.A and HSCTLR.A controls on the generation of alignment faults. The HSCTLR control applies to Normal memory accesses from Hyp mode, and the SCTLR control applies to Normal memory accesses from all other modes.

The `Mem_with_type[]` functions make an access of the required type. If that access is naturally aligned, each form of the function performs an atomic access by making a single call to `MemSingle[]`. If that access is not aligned but passes the `AArch32.CheckAlignment()` checks, each form of the function synthesizes the required access from multiple calls to `MemSingle[]`. It also reverses the byte order if the access is big-endian.

```
// Mem_with_type[] - non-assignment (read) form
// ============================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    integer i;
    boolean iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned];

    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem_with_type[] - assignment (write) form
// =========================================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
    integer i;
    boolean iswrite = TRUE;

    if BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    else
        AArch32.MemSingle[address, size, acctype, aligned] = value;
    return;
```

### G3.8.5 Exclusive monitors operations

The `SetExclusiveMonitors()` function sets the exclusive monitors for a Load-Exclusive instruction, for a block of bytes. The size of the blocks is determined by `size`, at the VA `address`. The `ExclusiveMonitorsPass()` function checks whether a Store-Exclusive instruction still has possession of the exclusive monitors and therefore completes successfully.

```
// AArch32.SetExclusiveMonitors()
// ==============================

// Sets the Exclusive Monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = (address != Align(address, size));

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

The ExclusiveMonitorsPass() function checks whether a Store-Exclusive instruction still has possession of the exclusive monitors, by checking whether the exclusive monitors are set to include the location of the memory block specified by size, at the virtual address defined by address. The atomic write that follows after the exclusive monitors have been set must be to the same physical address. It is permitted, but not required, for this function to return FALSE if the virtual address is not the same as that used in the previous call to SetExclusiveMonitors().

```
// AArch32.ExclusiveMonitorsPass()
// ===============================

// Return TRUE if the Exclusive Monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusive Monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

The ExclusiveMonitorsStatus() function returns 0 if the previous atomic write was to the same physical memory locations selected by ExclusiveMonitorsPass() and therefore succeeded. Otherwise the function returns 1, indicating that the address translation delivered a different physical address.

```
bit ExclusiveMonitorsStatus();
```

The MarkExclusiveGlobal() procedure takes as arguments a FullAddress.paddress, the PE identifier processorid and the size of the transfer. The procedure records that the PE processorid has requested exclusive access covering at least size bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED,


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

up to a limit of 2KB and no smaller than two words, and aligned in the address space to the size of the location. It is UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same PE to be cleared.

```
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

The `MarkExclusiveLocal()` procedure takes as arguments a FullAddress `paddress`, the PE identifier `processorid` and the size of the transfer. The procedure records in a local record that PE `processorid` has requested exclusive access to an address covering at least `size` bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory but is no smaller than two words, and is aligned in the address space to the size of the location. It is IMPLEMENTATION DEFINED whether this procedure also performs a `MarkExclusiveGlobal()` using the same parameters.

```
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

The `IsExclusiveGlobal()` function takes as arguments a FullAddress `paddress`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked in a global record an address range as exclusive access requested that covers at least `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, `IsExclusiveGlobal()` returns FALSE.

```
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

The `IsExclusiveLocal()` function takes as arguments a FullAddress `paddress`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked an address range as exclusive access requested that covers at least the `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of `size` bytes from address `paddress`. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of `IsExclusiveGlobal()` with the same parameters.

```
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

The `ClearExclusiveByAddress()` procedure takes as arguments a `FullAddress` `paddress`, the PE identifier `processorid` and the size of the transfer. The procedure clears the global records of all PEs, other than `processorid`, for which an address region including any of `size` bytes starting from `paddress` has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether the equivalent global record of the PE `processorid` is also cleared if any of `size` bytes starting from `paddress` has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

```
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

The `ClearExclusiveLocal()` procedure takes as arguments the PE identifier `processorid`. The procedure clears the local record of PE `processorid` for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE `processorid` that an address has had a request for an exclusive access.

```
ClearExclusiveLocal(integer processorid);
```

### G3.8.6 Access permission checking

The function `CheckPermission()` is used by the architecture to perform access permission checking based on attributes derived from the translation tables or location descriptors.

The interpretation of access permission is shown in *Memory access control* on page G4-4037.

The pseudocode function for checking access permissions is as follows:

```
// AArch32.CheckPermission()
// =========================
// Function used for permission checking from AArch32 stage 1 translations

FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level,
                                    bits(4) domain, bit NS, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if PSTATE.EL != EL2 then
        wxn = SCTLR.WXN == '1';
        if TTBCR.EAE == '1' || SCTLR.AFE == '1' || perms.ap<0> == '1' then
            priv_r = TRUE;
            priv_w = perms.ap<2> == '0';
            user_r = perms.ap<1> == '1';
            user_w = perms.ap<2:1> == '01';
        else
            priv_r = perms.ap<2:1> != '00';
            priv_w = perms.ap<2:1> == '01';
            user_r = perms.ap<1> == '1';
            user_w = FALSE;
        uwxn = SCTLR.UWXN == '1';
        user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
        priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
                   (priv_w && wxn) || (user_w && uwxn));
        ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2
        wxn = HSCTLR.WXN == '1';
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' then
        secure_instr_fetch = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;
        if secure_instr_fetch == '1' then xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(40) UNKNOWN;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch32.NoFault();
```

### G3.8.7 Abort exceptions

The `Abort()` function generates a Data Abort exception or a Prefetch Abort exception by calling the `TakeDataAbortException()` or `TakePrefetchAbortException()` function.

```
// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress), fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
```

The `FaultRecord` type describes a fault. Functions that check for faults return a record of this type appropriate to the type of fault. *Pseudocode description of VMSAv8-32 memory system operations* on page G4-4183 provides a number of wrappers to generate a FaultRecord.

The `NoFault()` function returns a null record that indicates no fault. The `IsFault()` function tests whether a FaultRecord contains a fault.

```
enumeration Fault {Fault_None,
                   Fault_AccessFlag,
                   Fault_Alignment,
                   Fault_Background,
                   Fault_Domain,
                   Fault_Permission,
                   Fault_Translation,
                   Fault_AddressSize,
                   Fault_SyncExternal,
                   Fault_SyncExternalOnWalk,
                   Fault_SyncParity,
                   Fault_SyncParityOnWalk,
                   Fault_AsyncParity,
                   Fault_AsyncExternal,
                   Fault_Debug,
                   Fault_TLBConflict,
                   Fault_Lockdown,
                   Fault_Exclusive,
                   Fault_ICacheMaint};

type FaultRecord is (Fault    type,         // Fault Status
                     AccType  acctype,      // Type of access that faulted
                     bits(48) ipaddress,    // Intermediate physical address
                     boolean  s2fs1walk,    // Is on a Stage 1 page table walk
                     boolean  write,        // TRUE for a write, FALSE for a read
                     integer  level,        // For translation, access flag and permission faults
                     bit      extflag,      // IMPLEMENTATION DEFINED syndrome for external aborts
                     boolean  secondstage,  // Is a Stage 2 abort
                     bits(4)  domain,       // Domain number, AArch32 only
                     bits(4)  debugmoe)     // Debug method of entry, from AArch32 only
// AArch32.NoFault()
// =================

FaultRecord AArch32.NoFault()

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_None, ipaddress,  domain, level, acctype, iswrite,
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                                        extflag, debugmoe, secondstage, s2fs1walk);
// IsFault()
// =========
// Return true if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.type != Fault_None;
```

### G3.8.8 Memory barriers

The definition for the memory barrier functions is:

```
enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};

enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

These functions define the required Shareability domains and required access types used as arguments for `DMB` and `DSB` instructions.

The following procedures perform the memory barriers:

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);

InstructionSynchronizationBarrier();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter G4
# The AArch32 Virtual Memory System Architecture

This chapter describes the ARMv8-A AArch32 Virtual Memory System Architecture (VMSA), that is backwards-compatible with VMSAv7. It includes the following sections:


- *Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990.
- *About VMSAv8-32* on page G4-3992.
- *The effects of disabling address translation stages on VMSAv8-32 behavior* on page G4-4000.
- *Translation tables* on page G4-4004.
- *The VMSAv8-32 Short-descriptor translation table format* on page G4-4009.
- *The VMSAv8-32 Long-descriptor translation table format* on page G4-4018.
- *Memory access control* on page G4-4037.
- *Memory region attributes* on page G4-4046.
- *Translation Lookaside Buffers (TLBs)* on page G4-4058.
- *TLB maintenance requirements* on page G4-4061.
- *Caches in VMSAv8-32* on page G4-4074.
- *VMSAv8-32 memory aborts* on page G4-4078.
- *Exception reporting in a VMSAv8-32 implementation* on page G4-4091.
- *Address translation instructions* on page G4-4110.
- *About the System registers for VMSAv8-32* on page G4-4116.
- *Organization of the CP14 registers in VMSAv8-32* on page G4-4138.
- *Organization of the CP15 registers in VMSAv8-32* on page G4-4141.
- *Functional grouping of VMSAv8-32 System registers* on page G4-4161.
- *Pseudocode description of VMSAv8-32 memory system operations* on page G4-4183.


**Note**

This chapter must be read with Chapter G3 *The AArch32 System Level Memory Model*.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G4.1 Execution privilege, Exception levels, and AArch32 Privilege levels

In ARMv8, the hierarchy of software execution privilege, within a particular Security state, is defined by the Exception levels, with higher Exception level numbers indicating higher privilege. Table G4-1 shows this hierarchy for each Security state.

**Table G4-1 Execution privilege and Exception levels, by Security state**

| Execution privilege | Secure state | Non-secure state | Typical use |
|---|---|---|---|
| Highest | EL3 | -[a] | Secure monitor |
| - | -[a] | EL2 | Hypervisor |
| - | EL1 | EL1 | Secure or Non-secure OS |
| Lowest, Unprivileged | EL0 | EL0 | Secure or Non-secure application |

a. EL2 is never implemented in Secure state, and EL3 is never implemented in Non-secure state.

When executing in AArch32 state, within a given Security state, the current PE state, including the execution privilege, is primarily indicated by the current *PE mode*. In Secure state, how the PE modes map onto the Exception levels depends on whether EL3 is using AArch32 or is using AArch64, and:

- Figure G1-1 on page G1-3754 shows this mapping when EL3 is using AArch32.
- Figure G1-2 on page G1-3761 shows this mapping when EL3 is using AArch64.

Table G4-2 shows this mapping. In interpreting this table:

- Monitor mode is implemented only in Secure state, and only if EL3 is using AArch32.
- Hyp mode is implemented only in Non-secure state, and only if EL2 is using AArch32.
- System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented:
  - **In Secure state** If either:
    - EL3 is using AArch32.
    - EL3 is using AArch64 and EL1 is using AArch32.
  - **In Non-secure state** If EL1 is using AArch32.
- User mode is implemented if EL0 is using AArch32.

**Table G4-2 Mapping of AArch32 PE modes to Exception levels**

| Exception level | PE modes in the given Security state, and EL3 Execution state | | |
|---|---|---|---|
| | Secure state, EL3 using AArch32 | Secure state, EL3 using AArch64 | Non-secure state |
| EL3 | Monitor, System, FIQ, IRQ, Supervisor, Abort, Undefined | - | - |
| EL2 | - | - | Hyp |
| EL1 | - | System, FIQ, IRQ, Supervisor, Abort, Undefined | System, FIQ, IRQ, Supervisor, Abort, Undefined |
| EL0 | User | User | User |

Because AArch32 behavior is described in terms of the PE modes, and transitions between PE modes, the Exception levels are implicit in most of the description of operation in AArch32 state.

However, the *translation regimes* provided by the VMSA cannot be described only in terms of the PE modes. In AArch64 state these regimes are defined by the Exception levels that use them. However, in AArch32 state this would result in descriptions that, for Secure state operation in modes other than User mode, would depend on the Exception level being used by AArch32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

To provide a consistent description of address translation as seen from AArch32 state, the translation regimes are described in terms of the Privilege levels originally defined in the ARMv7 descriptions of AArch32 state. Table G4-3 shows this.

**Table G4-3 Mapping of PE modes to AArch32 Privilege levels**

| Privilege level | Secure state | Non-secure state |
|---|---|---|
| PL2 | - | Hyp[a] |
| PL1 | Monitor[b], System, FIQ, IRQ, Supervisor, Abort, Undefined | System, FIQ, IRQ, Supervisor, Abort, Undefined |
| PL0 | User | User |

a. Implemented only in Non-secure state, and only if EL2 is using AArch32
b. Implemented only in Secure state, and only if EL3 is using AArch32.

Comparing Table G4-3 with Table G4-2 on page G4-3990 shows that:

**In Non-secure state**

Each privilege level maps to the corresponding Exception level. For example PL1 maps to EL1.

**In Secure state**

PL0 maps to EL0.

The mapping of PL1 depends on the Execution state being used by EL3, as follows:

**EL3 using AArch64** Secure PL1 maps to Secure EL1. Monitor mode is not implemented.

**EL3 using AArch32** Secure PL1 maps to Secure EL3. Monitor mode is implemented as one of the Secure PL1 modes.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.2 About VMSAv8-32

**Note**

- This chapter describes the ARMv8 VMSA for AArch32 state, VMSAv8-32. This is generally equivalent to VMSAv7 for an implementation that includes all of the Security Extensions, the Multiprocessing Extensions, the Large Physical Address Extension, and the Virtualization Extensions.
- This chapter describes the control of the VMSA by Exception levels that are using AArch32. *Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990 summarizes how the AArch32 PE modes map onto the Exception levels.

  Chapter D4 *The AArch64 Virtual Memory System Architecture* describes the control of the VMSA by exception levels that are using AArch64.
- For details of the VMSA differences in previous versions of the ARM architecture see the *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

The main function of the VMSA is to perform address translation, and access permissions and memory attribute determination and checking, for memory accesses made by the PE. Address translation, and permissions and attribute determination and checking, is performed by a *stage* of address translation.

In VMSAv8-32, the *Memory Management Unit* (MMU) provides a number of stages of address translation. This chapter describes only the stages that are visible from Exception levels that are using AArch32, which are as follows:

**For operation in Secure state**

A single stage of address translation, for use when executing at PL1 or PL0. This is the *Secure PL1&0 stage 1* address translation stage.

**For operation in Non-secure state**

- A single stage of address translation for use when executing at PL2. This is the *Non-secure PL2 stage 1* address translation stage.
- Two stages of address translation for use when executing at PL1 or PL0. These are:
  - The *Non-secure PL1&0 stage 1* address translation stage.
  - The *Non-secure PL1&0 stage 2* address translation stage.

The System registers provide independent control of each supported stage of address translation, including a control to disable that stage of translation.

These features mean the VMSAv8-32 can support a hierarchy of software supervision, for example an Operating System and a hypervisor.

Each stage of address translation uses address translations and associated memory properties held in memory mapped tables called *translation tables*.

For information about how the MMU features differ if an implementation does not include all of the Exception levels, see *About address translation for VMSAv8-32* on page G4-3995.

The translation tables define the following properties:

**Access to the Secure or Non-secure address map**

The translation table entries determine whether an access from Secure state accesses the Secure or the Non-secure address map. Any access from Non-secure state accesses the Non-secure address map.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Memory access permission control**

This controls whether a program is permitted to access a memory region. For instruction and data access, the possible settings are:

- No access.
- Read-only.
- Write-only. This is possible only in a translation regime with two stages of translation.
- Read/write.

For instruction accesses, additional controls determine whether instructions can be fetched and executed from the memory region.

If a PE attempts an access that is not permitted, a memory fault is signaled to the PE.

**Memory region attributes**

These describe the properties of a memory region. The top-level attribute, the Memory type, is one of Normal, or a type of Device memory, as follows:

- Both translation table formats support the following Device memory types:
  - Device-nGnRnE
  - Device-nGnRE
- The Long-descriptor translation table format supports, in addition, the following Device memory types:
  - Device-nGRE
  - Device-GRE

**Note**

ARMv8 added the Device-nGRE and Device-GRE memory types. Also, in versions of the ARM architecture before ARMv8:

- Device-nGnRnE memory is described as Strongly-ordered memory.
- Device-nGRE memory is described as Device memory.

Normal memory regions can have additional attributes.

For more information, see *Memory types and attributes* on page E2-2382.

**Address translation mappings**

An address translation maps an *input address* to an *output address*.

A stage 1 translation takes the address of an explicit data access or instruction fetch, a *virtual address* (VA), as the input address, and translates it to a different output address:

- If only one stage of translation is provided, this output address is the *physical address* (PA).
- If two stages of address translation are provided, the output address of the stage 1 translation is an *intermediate physical address* (IPA).

**Note**

In the ARMv8-32 architecture, a software agent, such as an Operating System, that uses or defines stage 1 memory translations, might be unaware of the distinction between IPA and PA.

A stage 2 translation translates the IPA to a PA.

The possible security states and privilege levels of memory accesses define a set of *translation regimes*. Figure G4-1 on page G4-3994 shows the VMSAv8-32 translation regimes, and their associated translation stages and the Exception levels from which they are controlled.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

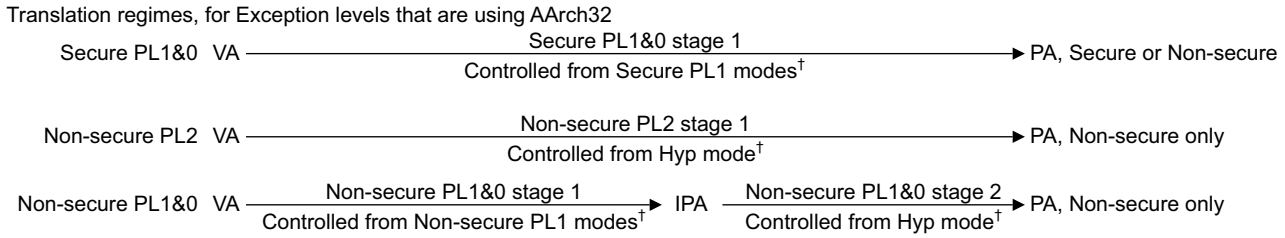



**Figure G4-1 VMSAv8-32 translation regimes, and associated control**

**Note**

Conceptually, a translation regime that has only a stage 1 address translation is equivalent to a regime with a fixed, flat stage 2 mapping from IPA to PA.

System registers control VMSAv8-32, including defining the location of the translation tables, and enabling and configuring the MMU, including enabling and disabling the different address translation stages. Also, they report any faults that occur on a memory access. For more information, see *Functional grouping of VMSAv8-32 System registers* on page G4-4161.

The following sections give an overview of VMSAv8-32, and of the implementation options for VMSAv8-32:

- *Address types used in a VMSAv8-32 description*.
- *Address spaces in VMSAv8-32* on page G4-3995.
- *About address translation for VMSAv8-32* on page G4-3995.

The remainder of the chapter fully describes the VMSA, including the different implementation options, as summarized in *Organization of this chapter* on page G4-3998.

### G4.2.1 Address types used in a VMSAv8-32 description

A description of VMSAv8-32 refers to the following address types.

**Note**

These descriptions relate to a VMSAv8-32 description and therefore sometimes differ from the generic definitions given in the Glossary.

**Virtual address (VA)**

An address used in an instruction, as a data or instruction address, is a Virtual Address (VA).

An address held in the PC, LR, or SP, is a VA.

The VA map runs from zero to the size of the VA space. For AArch32 state, the maximum VA space is 4GB, giving a maximum VA range of `0x00000000-0xFFFFFFFF`.

**Intermediate physical address (IPA)**

In a translation regime that provides two stages of address translation, the IPA is the address after the stage 1 translation, and is the input address for the stage 2 translation.

In a translation regime that provides only one stage of address translation, the IPA is identical to the PA.

A VMSAv8-32 implementation provides only one stage of address translation:

- If the implementation does not include EL2.
- When executing in Secure state.
- When executing in Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Physical address (PA)**

The address of a location in the Secure or Non-secure memory map. That is, an output address from the PE to the memory system.

### G4.2.2 Address spaces in VMSAv8-32

For execution in AArch32 state, the ARMv8 architecture supports:

- A VA space of up to 32 bits. The actual width is IMPLEMENTATION DEFINED.
- An IPA space of up to 40 bits. The translation tables and associated System registers define the width of the implemented address space.

**Note**

AArch32 defines two translation table formats. The *Long-descriptor* format gives access to the full 40-bit IPA or PA space at a granularity of 4KB. The *Short-descriptor* format:

- Gives access to a 32-bit PA space at 4KB granularity.
- Gives access to a 40-bit PA space, but only at 16MB granularity, by the use of Supersections.

If an implementation includes EL3, the address maps are defined independently for Secure and Non-secure operation, providing two independent 40-bit address spaces, where:

- A VA accessed from Non-secure state can only be translated to the Non-secure address map.
- A VA accessed from Secure state can be translated to either the Secure or the Non-secure address map.

### G4.2.3 About address translation for VMSAv8-32

Address translation is the process of mapping one address type to another, for example, mapping VAs to IPAs, or mapping VAs to PAs. A *translation table* defines the mapping from one address type to another, and a *Translation table base register* (TTBR) indicates the start of a translation table. Each implemented stage of address translation shown in Figure G4-1 on page G4-3994 requires its own translation tables.

For PL1&0 stage 1 translations, the mapping can be split between two tables, one controlling the lower part of the VA space, and the other controlling the upper part of the VA space. This can be used, for example, so that:

- One table defines the mapping for operating system and I/O addresses, that do not change on a context switch.
- A second table defines the mapping for application-specific addresses, and therefore might require updating on a context switch.

The VMSAv8-32 implementation options determine the supported address translation stages. The following descriptions apply when all implemented Exception levels are using AArch32:

**VMSAv8-32 without EL2 or EL3**

Supports only a single PL1&0 stage 1 address translation. Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by TTBR0 and TTBR1, and controlled by TTBCR.

**VMSAv8-32 with EL3 but without EL2**

Supports only the Secure PL1&0 stage 1 address translation and the Non-secure PL1&0 stage 1 address translation. In each security state, this stage of translation can be split between two sets of translation tables, with base addresses defined by the Secure and Non-secure copies of TTBR0 and TTBR1, and controlled by the Secure and Non-secure copies of TTBCR.

**VMSAv8-32 with EL2 but without EL3**

The implementation supports the following stages of address translation:

**Non-secure PL2 stage 1 address translation**

The HTTBR defines the base address of the translation table for this stage of address translation, controlled by HTCR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Non-secure PL1&0 stage 1 address translation**

Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Non-secure copies of TTBR0 and TTBR1 and controlled by the Non-secure instance of TTBCR.

**Non-secure PL1&0 stage 2 address translation**

The VTTBR defines the base address of the translation table for this stage of address translation, controlled by VTCR.

**VMSAv8-32 with EL2 and EL3**

The implementation supports all of the stages of address translation, as follows:

**Secure PL1&0 stage 1 address translation**

Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Secure copies of TTBR0 and TTBR1, and controlled by the Secure instance of TTBCR.

**Non-secure PL2 stage 1 address translation**

The HTTBR defines the base address of the translation table for this stage of address translation, controlled by HTCR.

**Non-secure PL1&0 stage 1 address translation**

Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Non-secure copies of TTBR0 and TTBR1 and controlled by the Non-secure instance of TTBCR.

**Non-secure PL1&0 stage 2 address translation**

The VTTBR defines the base address of the translation table for this stage of address translation, controlled by VTCR.

Figure G4-2 shows the translation regimes and stages in a VMSAv8-32 implementation that includes all of the Exception levels, and indicates the PE mode that, typically, defines each set of translation tables, if that stage of address translation is controlled by a Privilege level that is using AArch32:

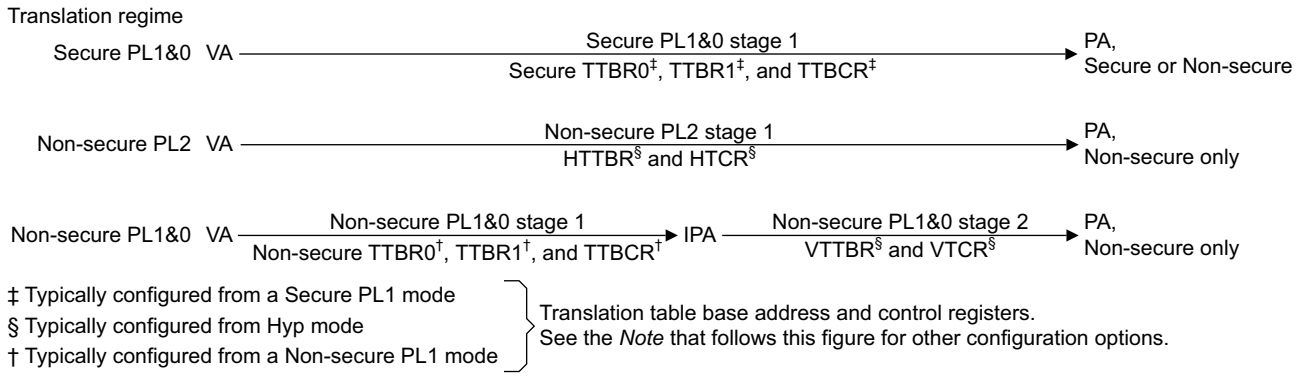



**Figure G4-2 VMSAv8-32 address translation summary**

**Note**

The term *Typically configured* is used in Figure G4-2 to indicate the expected software usage. However, stages of address translation used in AArch32 state can also be configured:

- From an Exception level higher than the Exception level of the configuring PE mode shown in Figure G4-2, regardless of whether that Exception level is using AArch32 or is using AArch64, except that a Non-secure Exception level can never configure a stage of address translation that is used in Secure state.
- From an Exception level that is using AArch64 and is higher than the level at which the translation stage is being used. For example, if Non-secure EL0 is the only Non-secure Exception level that is using AArch32, then the Non-secure PL1&0 stage of address translation is configured from Non-secure EL1, that is using AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In general:

- The translation from VA to PA can require multiple *stages* of address translation, as Figure G4-2 on page G4-3996 shows.
- A single stage of address translation takes an *input address* and translates it to an *output address*.

A full translation table lookup is called a *translation table walk*. It is performed automatically by hardware, and can have a significant cost in execution time. To support fine granularity of the VA to PA mapping, a single input address to output address translation can require multiple accesses to the translation tables, with each access giving finer granularity. Each access is described as a *level* of address lookup. The final level of the lookup defines:

- The required output address.
- The *attributes* and *access permissions* of the addressed memory.

*Translation Lookaside Buffers* (TLBs) reduce the average cost of a memory access by caching the results of translation table walks. TLBs behave as caches of the translation table information, and VMSAv8-32 provides TLB maintenance instructions for the management of TLB contents.

**Note**

The ARM architecture permits TLBs to hold any translation table entry that does not directly cause a Translation fault, an Address size fault, or an Access flag fault.

To reduce the software overhead of TLB maintenance, for the PL1&0 translation regimes VMSAv8-32 distinguishes between *Global pages* and *Process-specific pages*. The *Address Space Identifier* (ASID) identifies pages associated with a specific process and provides a mechanism for changing process-specific tables without having to maintain the TLB structures.

If an implementation includes EL2, the *virtual machine identifier* (VMID) identifies the current virtual machine, with its own independent ASID space. The TLB entries include this VMID information, meaning TLBs do not require explicit invalidation when changing from one virtual machine to another, if the virtual machines have different VMIDs. For stage 2 translations, all translations are associated with the current VMID. There is no mechanism to associate a particular stage 2 translation with multiple virtual machines.

### Atomicity of register changes on changing virtual machine

From the viewpoint of software executing at Non-secure PL1 or PL0, when there is a switch from one virtual machine to another, the registers that control or affect address translation must be changed atomically. This applies to the registers for the Non-secure PL1&0 translation regime. This means that all of the following registers must change atomically:

- The registers associated with the stage 1 translations:
  - MAIR0, MAIR1, AMAIR0, and AMAIR1.
  - TTBR0, TTBR1, TTBCR, and CONTEXTIDR.
  - SCTLR.
- The registers associated with the stage 2 translations:
  - VTTBR and VTCR.
  - HMAIR0 and HMAIR1.
  - HSCTLR.

**Note**

Only some bits of SCTLR affect the stage 1 translation, and only some bits of HSCTLR affect the stage 2 translation. However, in each case, changing these bits requires a write to the register, and that write must be atomic with the other register updates.

These registers apply to execution using the Non-secure PL1&0 translation regime. However, when updated as part of a switch of virtual machines they are updated by software executing at EL2. This means the registers are out of context when they are updated, and no synchronization precautions are required.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Use of out-of-context translation regimes

The architecture requires that:

- When executing at EL3 or EL2, the PE must not use the registers associated with the Non-secure PL1&0 translation regime for speculative memory accesses.
- When executing at EL3 the PE must not use the registers associated with the EL2 translation regime for speculative memory accesses.
- When executing at EL3, EL2, or Non-secure EL1, the PE must not use the registers associated with the Secure PL1&0 translation regime for speculative memory accesses.

When entering an exception level on completion of a `DSB` instruction, no new memory accesses using any translation table entries from a translation regime of an exception level lower than the exception level that has been entered, will be observed by any observers to the extent that those accesses are required to be observed, as determined by the Shareability and Cacheability of those translation table entries.

**Note**

- This does not require that speculative memory accesses cannot be performed using those entries if it is impossible to tell that those memory accesses have been observed by the observers.
- This requirement does not imply that, on taking an exception to a higher Exception level, any translation table walks started before the exception was taken will be completed by the time the higher Exception level is entered, and therefore memory accesses required for such a translation table walk might, in effect, be performed speculatively. However, the execution of a `DSB` on entry to the higher Exception level ensures that these accesses are complete.

## G4.2.4 Organization of this chapter

The remainder of this chapter is organized as follows.

The first part of the chapter describes address translation and the associated memory properties held in the translation table entries, in the following sections:

- *The effects of disabling address translation stages on VMSAv8-32 behavior* on page G4-4000.
- *Translation tables* on page G4-4004.
- *Secure and Non-secure address spaces* on page G4-4007.
- *The VMSAv8-32 Short-descriptor translation table format* on page G4-4009.
- *The VMSAv8-32 Long-descriptor translation table format* on page G4-4018.
- *Memory access control* on page G4-4037.
- *Memory region attributes* on page G4-4046.
- *Translation Lookaside Buffers (TLBs)* on page G4-4058.
- *TLB maintenance requirements* on page G4-4061.

*Caches in VMSAv8-32* on page G4-4074 describes VMSAv8-32-specific cache requirements.

The following sections describe aborts on VMSAv8-32 memory accesses, and how these and other faults are reported:

- *VMSAv8-32 memory aborts* on page G4-4078.
- *Exception reporting in a VMSAv8-32 implementation* on page G4-4091.

*Address translation instructions* on page G4-4110 describes these operations, and how they relate to address translation.

A number of sections then describe the System registers for VMSAv8-32. The following sections give general information about the System registers, and the organization of the registers in the two coprocessor encoding spaces, CP14 and CP15, that provide the interface to these registers:

- *About the System registers for VMSAv8-32* on page G4-4116.
- *Organization of the CP14 registers in VMSAv8-32* on page G4-4138.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *Organization of the CP15 registers in VMSAv8-32* on page G4-4141.
- *Functional grouping of VMSAv8-32 System registers* on page G4-4161.

The following sections then describe each of the functional groups of CP15 registers, including a full description of each register in the group:

- *Identification registers, functional group* on page G4-4162.
- *Virtual memory control registers, functional group* on page G4-4163.
- *Exception and fault handling registers, functional group* on page G4-4168.
- *Other System registers, functional group* on page G4-4163.
- *Lockdown, DMA, and TCM features, functional group* on page G4-4173.
- *Cache maintenance instructions, functional group* on page G4-4169.
- *TLB maintenance instructions, functional group* on page G4-4171.
- *Address translation instructions, functional group* on page G4-4172.
- *Legacy feature registers, functional group* on page G4-4179.
- *Performance Monitors Extension registers, functional group* on page G4-4173.
- *Security registers, functional group* on page G4-4167.
- *Virtualization registers, functional group* on page G4-4164.
- *IMPLEMENTATION DEFINED registers, functional group* on page G4-4179.

*Pseudocode description of VMSAv8-32 memory system operations* on page G4-4183 then describes many features of VMSAv8-32 operation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.3 The effects of disabling address translation stages on VMSAv8-32 behavior

*About VMSAv8-32* on page G4-3992 defines the translation regimes and the associated stages of address translation, each of which has its own System registers for control and configuration. VMSAv8-32 includes an enable bit for each stage of address translation, as follows:

- SCTLR.M, in the Secure instance of the register, controls Secure PL1&0 stage 1 address translation.
- SCTLR.M, in the Non-secure instance of the register, controls Non-secure PL1&0 stage 1 address translation.
- HCR.VM controls Non-secure PL1&0 stage 2 address translation.
- HSCTLR.M controls Non-secure PL2 stage 1 address translation.

**Note**

- The descriptions throughout this chapter describe address translation as seen by Exception levels that are using AArch32. However, for the Non-secure PL1&0 translation regime, the stage 2 translation:
  - Is controlled by the HCR if EL2 is using AArch32.
  - Is controlled by the HCR_EL2 if EL2 is using AArch64.

  For this reason, links to the HCR link to a table that disambiguates between the AArch32 HCR and the AArch64 HCR_EL2.
- If EL2 is using AArch64, then the equivalent of the Non-secure PL2 translation regime is described in Chapter D4 *The AArch64 Virtual Memory System Architecture*, not in this chapter.

The following sections describe the effect on VMSAv8-32 behavior of disabling each stage of translation:

- *VMSAv8-32 behavior when stage 1 address translation is disabled*.
- *VMSAv8-32 behavior when stage 2 address translation is disabled* on page G4-4002.
- *Behavior of instruction fetches when all associated address translations are disabled* on page G4-4002.

*Enabling stages of address translation* on page G4-4002 gives more information about each stage of address translation, in particular after a reset on an implementation that includes EL3.

### G4.3.1 VMSAv8-32 behavior when stage 1 address translation is disabled

When stage 1 address translation is disabled, memory accesses that would otherwise be translated by that stage of address translation are treated as follows:

**Non-secure PL1 and PL0 accesses when EL2 is implemented and HCR.DC is set to 1**

In an implementation that includes EL2, for an access from a Non-secure PL1 or PL0 mode when HCR.DC is set to 1, the stage 1 translation assigns the Normal Non-shareable, Inner Write-Back Write-Allocate, Outer Write-Back Write-Allocate memory attributes.

See also *Effect of the HCR.DC bit* on page G4-4001.

**All other accesses**

For all other accesses, when a stage 1 address translation is disabled, the assigned attributes depend on whether the access is a data access or an instruction access, as follows:

**Data access**

The stage 1 translation assigns the Device-nGnRnE memory type.

**Instruction access**

The stage 1 translation assigns Normal memory attribute, with the Cacheability and Shareability attributes determined by the value of:

- The Secure instance of SCTLR.I for the Secure PL1&0 translation regime.
- The Non-secure instance of SCTLR.I for the Non-secure PL1&0 translation regime.
- HSCTLR.I for the Non-secure PL2 translation regime.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In these cases, the meaning of the I bit is as follows:

**When I is set to 0**

The stage 1 translation assigns the attributes Outer Shareable, Non-cacheable.

**When I is set to 1**

The stage 1 translation assigns the attributes Inner Write-Through no Write-Allocate, Outer Write-Through no Write-Allocate Cacheable.

**Note**

On some implementations, if the SCTLR.TRE bit is set to 0 then this behavior can be changed by the remap settings in the memory remap registers. The details of TEX remap when SCTLR.TRE is set to 0 are IMPLEMENTATION DEFINED, see *SCTLR.TRE, SCTLR.M, and the effect of the TEX remap registers* on page G4-4051.

For this stage of translation, no memory access permission checks are performed, and therefore no MMU faults relating to this stage of translation can be generated.

**Note**

Alignment checking is performed, and therefore Alignment faults can occur.

For every access, when stage 1 translation is disabled, the output address of the stage 1 translation is equal to the input address. This is called a flat address mapping. If the implementation supports output addresses of more than 32 bits then the output address bits above bit[31] are zero. For example, for a VA to PA translation on an implementation that supports 40-bit PAs, PA[39:32] is `0x00`.

For a Non-secure PL1 or PL0 access, if the PL1&0 stage 2 address translation is enabled, the stage 1 memory attribute assignments and output address can be modified by the stage 2 translation.

See also *Behavior of instruction fetches when all associated address translations are disabled* on page G4-4002.

### Effect of the HCR.DC bit

The HCR.DC bit determines the default memory attributes assigned for the first stage of the Non-secure PL1&0 translation regime when that stage of translation is disabled.

When executing in a Non-secure PL1 or PL0 mode with HCR.DC set to 1:

- For all purposes other than reading the value of the SCTLR, the PE behaves as if the value of the SCTLR.M bit is 0. This means Non-secure PL1&0 stage 1 address translation is disabled.
- For all purposes other than reading the value of the HCR, the PE behaves as if the value of the HCR.VM bit is 1. This means Non-secure PL1&0 stage 2 address translation is enabled.

The effect of HCR.DC might be held in TLB entries associated with a particular VMID. Therefore, if software executing at EL2 changes the HCR.DC value without also changing the current VMID, it must also invalidate all TLB entries associated with the current VMID. Otherwise, the behavior of Non-secure software executing at EL1 or EL0 is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5427.

### Effect of disabling translation on maintenance and address translation instructions

Cache maintenance instructions act on the target cache whether address translation is enabled or not, and regardless of the values of the memory attributes. However, if a stage of translation is disabled, they use the flat address mapping for that stage, and all mappings are considered global.

TLB invalidate operations act on the target TLB whether address translation is enabled or not.

When the Non-secure PL1&0 stage 1 address translation is disabled, any ATS1C** or ATS12NSO** address translation instruction that accesses the Non-secure state translation reflects the effect of the HCR.DC bit. For more information about these operations see *Address translation instructions* on page G4-4110.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.3.2 VMSAv8-32 behavior when stage 2 address translation is disabled

When stage 2 address translation is disabled:

- The IPA output from the stage 1 translation maps flat to the PA
- The memory attributes and permissions from the stage 1 translation apply to the PA.

If the stage 1 address translation and the stage 2 address translation are both disabled, see *Behavior of instruction fetches when all associated address translations are disabled*.

### G4.3.3 Behavior of instruction fetches when all associated address translations are disabled

The information in this section applies to memory accesses:

- From Secure PL1 and PL0 modes, when the Secure PL1&0 stage 1 address translation is disabled
- From Hyp mode, when the Non-secure PL2 stage 1 address translation is disabled
- From Non-secure PL1 and PL0 modes, when all of the following apply:
  - The Non-secure PL1&0 stage 1 address translation is disabled.
  - The Non-secure PL1&0 stage 2 address translation is disabled.
  - HCR.DC is set to 0.

In these cases, a memory location might be accessed as a result of an instruction fetch if the following condition is met:

- The memory location is in the same 4KB block of memory, aligned to 4KB, as an instruction which a simple sequential execution of the program either requires to be fetched now or has previously required to be fetched since the last reset or the last synchronization of instruction cache maintenance targeting the instruction's location, or is in the 4KB block immediately following such a block.

These accesses can be caused by speculative instruction fetches, regardless of whether the prefetched instruction is committed for execution.

**Note**

To ensure architectural compliance, software must ensure that both of the following apply:

- Instructions that will be executed when address translation is disabled are located in 4KB blocks of the address space that contain only memory that is tolerant to speculative accesses.
- Each 4KB block of the address space that immediately follows a 4KB block that holds instructions that will be executed when address translation is disabled also contains only memory that is tolerant to speculative accesses.

### G4.3.4 Enabling stages of address translation

On powerup or reset, only the SCTLR.M bit for the Exception level and Security state entered on reset is reset to 0, disabling address translation for the initial state of the PE. All other SCTLR.M and HSCTLR.M bits that are implemented are UNKNOWN after the reset.

This means, on powerup or reset:

- On an implementation that includes EL3, where EL3 is using AArch32:
  - The PL1&0 stage 1 address translation enable bit, SCTLR.M, is Banked, meaning there are separate enables for operation in Secure and Non-secure state.
  - If EL3 is using AArch32, only the Secure instance of the SCTLR.M bit resets to 0, disabling the Secure state PL1&0 stage 1 address translation. The reset value of the Non-secure instance of SCTLR.M is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- On an implementation that includes EL2, where EL2 is using AArch32, the HSCTLR.M bit, that controls the Non-secure PL2 stage 1 address translation:
  - If the implementation does not include EL3, resets to 0.
  - Otherwise, is UNKNOWN.
- On an implementation that does not include either EL2 or EL3, there is a single stage of translation. This is controlled by SCTLR.M, that resets to 0.

**Note**

If, for the software that enables or disables a stage of address translation, the input address of a stage 1 translation differs from the output address of that stage 1 translation, and the software is running in translation regime that is affected by that stage of translation, then the requirement to synchronize changes to the System registers means it is uncertain where in the instruction stream the change of the translation takes place. For this reason, ARM strongly recommends that the input address and the output address are identical in this situation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.4 Translation tables

VMSAv8-32 defines two alternative translation table formats:

**Short-descriptor format**

It uses 32-bit descriptor entries in the translation tables, and provides:

- Up to two levels of address lookup.
- 32-bit input addresses.
- Output addresses of up to 40 bits.
- Support for PAs of more than 32 bits by use of supersections, with 16MB granularity.
- Support for No access, Client, and Manager domains.

**Long-descriptor format**

It uses 64-bit descriptor entries in the translation tables, and provides:

- Up to three levels of address lookup.
- Input addresses of up to 40 bits, when used for stage 2 translations.
- Output addresses of up to 40 bits.
- 4KB assignment granularity across the entire PA range.
- No support for domains, all memory regions are treated as in a Client domain.
- Fixed 4KB table size, unless truncated by the size of the input address space.

**Note**

— Translation with a 40-bit input address range requires two concatenated 4KB top-level tables, aligned to 8KB.

— The VMASv8-64 Long-descriptor translation table format is generally similar to this format, but supports input and output addresses of up to 48 bits, and has an assignment granularity and table size defined by its *translation granule*. This can be 4KB, 16KB, or 64KB. See *The VMSAv8-64 translation table format* on page D4-1771.

In all implementations, of the possible address translations shown in Figure G4-2 on page G4-3996, for stages of address translation that are using AArch32:

- In a particular Security state, the translation tables for the PL1&0 stage 1 translations can use either translation table format, and the TTBCR.EAE bit indicates the current translation table format.
- The translation tables for the Non-secure PL2 stage 1 translations, and for the Non-secure PL1&0 stage 2 translations, must use the Long-descriptor translation table format.

Many aspects of performing a translation table walk depend on the current translation table format. Therefore, the following sections describe the two formats, including how the MMU performs a translation table walk for each format:

- *The VMSAv8-32 Short-descriptor translation table format* on page G4-4009.
- *The VMSAv8-32 Long-descriptor translation table format* on page G4-4018.

The following subsections describe aspects of the translation tables and translation table walks, for memory accesses from AArch32 state, that are independent of the translation table format:

- *Translation table walks for memory accesses using VMSAv8-32 translation regimes* on page G4-4005.
- *Information returned by a translation table lookup* on page G4-4005.
- *Determining the translation table base address in the VMSAv8-32 translation regimes* on page G4-4006.
- *Control of translation table walks on a TLB miss* on page G4-4007.
- *Access to the Secure or Non-secure physical address map* on page G4-4007.

See also *TLB maintenance requirements* on page G4-4061.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.4.1 Translation table walks for memory accesses using VMSAv8-32 translation regimes

A translation table walk occurs as the result of a TLB miss, and starts with a read of the appropriate starting-level translation table. The result of that read determines whether additional translation table reads are required, for this stage of translation, as described in either:

- *Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format* on page G4-4015.
- *Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format* on page G4-4032.

**Note**

When using the Short-descriptor translation table format, the starting level for a translation table walk is always a level 1 lookup. However, with the Long-descriptor translation table format, the starting-level can be either a level 1or a level 2 lookup.

For the PL1&0 stage 1 translations, SCTLR.EE determines the endianness of the translation table lookups. SCTLR is Banked, and therefore the endianness is determined independently for each Security state.

HSCTLR.EE defines the endianness for the Non-secure PL2 stage 1 and Non-secure PL1&0 stage 2 translations.

**Note**

**Dynamically changing translation table endianness**

Because any change to SCTLR.EE or HSCTLR.EE requires synchronization before it is visible to subsequent operations, ARM strongly recommends that:

- SCTLR.EE is changed only when either:
  - Executing in a mode that does not use the translation tables affected by SCTLR.EE.
  - Executing with SCTLR.M set to 0.
- HSCTLR.EE is changed only when either:
  - Executing in a mode that does not use the translation tables affected by HSCTLR.EE.
  - Executing with HSCTLR.M set to 0.

The physical address of the base of the starting-level translation table is determined from the appropriate TTBR, see *Determining the translation table base address in the VMSAv8-32 translation regimes* on page G4-4006.

For more information, see *Ordering and completion of TLB maintenance instructions* on page G4-4064.

Translation table walks must access data or unified caches, or data and unified caches, of other agents participating in the coherency protocol, according to the Shareability attributes described in the TTBR. These Shareability attributes must be consistent with the Shareability attributes for the translation tables themselves.

### G4.4.2 Information returned by a translation table lookup

When an associated stage of address translation is enabled, a memory access requires one or more translation table lookups. If the required translation table descriptor is not held in a TLB, a translation table walk is performed to obtain the descriptor. A lookup, whether from the TLB or as the result of a translation table walk, returns both:

- An output address that corresponds to the input address for the lookup.
- A set of properties that correspond to that output address.

The returned properties are classified as providing *address map control*, *access controls*, or *region attributes*. This classification determines how the descriptions of the properties are grouped. The classification is based on the following model:

**Address map control**

Memory accesses from Secure state can access either the Secure or the Non-secure address map, as summarized in *Access to the Secure or Non-secure physical address map* on page G4-4007.

Memory accesses from Non-secure state can only access the Non-secure address map.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Access controls**

Determine whether the PE, in its current state, can access the output address that corresponds to the given input address. If not, an MMU fault is generated and there is no memory access.

*Memory access control* on page G4-4037 describes the properties in this group.

**Attributes** Are valid only for an output address that the PE, in its current state, can access. The attributes define aspects of the required behavior of accesses to the target memory region.

*Memory region attributes* on page G4-4046 describes the properties in this group.

### G4.4.3 Determining the translation table base address in the VMSAv8-32 translation regimes

On a TLB miss, the VMSA must perform a translation table walk, and therefore must find the base address of the translation table to use for its lookup. A TTBR holds this address. As Figure G4-2 on page G4-3996 shows:

- For a Non-secure PL2 stage 1 translation, the HTTBR holds the required base address. The HTCR is the control register for these translations.
- For a Non-secure PL1&0 stage 2 translation, the VTTBR holds the required base address. The VTCR is the control register for these translations.
- For a PL1&0 stage 1 translation, either TTBR0 or TTBR1 holds the required base address. The TTBCR is the control register for these translations.

  The Non-secure copies of TTBR0, TTBR1, and TTBCR, relate to the Non-secure PL1&0 stage 1 translation. The Secure copies of TTBR0, TTBR1, and TTBCR, relate to the Secure PL1&0 stage 1 translation.

For the PL1&0 translation table walks:

- TTBR0 can be configured to describe the translation of VAs in the entire address map, or to describe only the translation of VAs in the lower part of the address map.
- If TTBR0 is configured to describe the translation of VAs in the lower part of the address map, TTBR1 is configured to describe the translation of VAs in the upper part of the address map.

The contents of the appropriate instance of the TTBCR determine whether the address map is separated into two parts, and where the separation occurs. The details of the separation depend on the current translation table format, see:

- *Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format* on page G4-4014.
- *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G4-4026.

Example G4-1 shows a typical use of the two sets of translation tables:

**Example G4-1 Example use of TTBR0 and TTBR1**

An example of using the two TTBRs for PL1&0 stage 1 address translations is:

**TTBR0** Used for process-specific addresses.

Each process maintains a separate level 1 translation table. On a context switch:

- TTBR0 is updated to point to the level 1 translation table for the new context.
- TTBCR is updated if this change changes the size of the translation table.
- The CONTEXTIDR is updated.

TTBCR can be programmed so that all translations use TTBR0 in a manner compatible with architecture versions before ARMv6.

**TTBR1** Used for operating system and I/O addresses, that do not change on a context switch.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.4.4 Control of translation table walks on a TLB miss

Two bits in the TCR for the translation stage required by a memory access control whether a translation table walk is performed on a TLB miss. These two bits are the:

- PD0 and PD1 bits, on a PE using the Short-descriptor translation table format.
- EPD0 and EPD1 bits, on a PE using the Long-descriptor translation table format.

**Note**

For the VMSAv8-32 translation regimes, the different bit names are because the bits are in different positions in TTBCR, depending on the translation table format.

The effect of these bits is:

**{E}PDx == 0** If a TLB miss occurs based on TTBRx, a translation table walk is performed. The current security state determines whether the memory access is Secure or Non-secure.

**{E}PDx == 1** If a TLB miss occurs based on TTBRx, a level 1 Translation fault is returned, and no translation table walk is performed.

### G4.4.5 Access to the Secure or Non-secure physical address map

As stated in *Address spaces in VMSAv8-32* on page G4-3995, a PE can access independent Secure and Non-secure address maps. When the PL1 Exception level is using AArch32, these are defined by the translation tables identified by the Secure TTBR0 and TTBR1. In both translation table formats in the Secure translation tables, the NS bit in a descriptor indicates whether the descriptor refers to the Secure or the Non-secure address map:

**NS == 0** Access the Secure physical address space.

**NS == 1** Access the Non-secure physical address space.

**Note**

In the Non-secure translation tables, the corresponding bit is SBZ. Non-secure accesses always access the Non-secure physical address space, regardless of the value of this bit.

The Long-descriptor translation table format extends this control, adding an NSTable bit to the Secure translation tables, as described in *Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format* on page G4-4025. In the Non-secure translation tables, the corresponding bit is SBZ, and Non-secure accesses ignore the value of this bit.

The following sections describe the address map controls in the two implementations:

- *Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format* on page G4-4014.
- *Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format* on page G4-4025.

The following subsection gives more information.

#### Secure and Non-secure address spaces

EL3 provides two physical address spaces, a Secure physical address space and a Non-secure physical address space.

As described in *Access to the Secure or Non-secure physical address map*, for the PL1&0 stage 1 translations when controlled from an Exception level using AArch32, the registers that control the stage of translation, TTBR0, TTBR1, and TTBCR, are Banked between Secure and Non-secure versions, and the Security state of the PE when it performs a memory access selects the corresponding version of the registers. This means there are independent Secure and Non-secure versions of these translation tables, and translation table walks are made to the physical address space corresponding to the security state of the translation tables used.

For a translation table walk caused by a memory access from Non-secure state, all memory accesses are to the Non-secure address space.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a translation table walk caused by a memory access from Secure state:

- When address translation is using the Long-descriptor translation table format:
  - The initial lookup performed must access the Secure address space.
  - If a table descriptor read from the Secure address space has the NSTable bit set to 0, then the next level of lookup is from the Secure address space.
  - If a table descriptor read from the Secure address space has the NSTable bit set to 1, then the next level of lookup, and any subsequent level of lookup, is from the Non-secure address space.

  For more information, see *Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format* on page G4-4025.
- Otherwise, all memory accesses are to the Secure address space.

**Note**

- When executing in Non-secure state, additional translations are supported. For memory accesses from AArch32 state these are:
  - Non-secure PL2 stage 1 translation.
  - Non-secure PL1&0 stage 2 translation.

  These translations can access only the Non-secure address space.
- A system implementation can alias parts of the Secure physical address space to the Non-secure physical address space in an implementation-specific way. As with any other aliasing of physical memory, the use of aliases in this way can require the use of cache maintenance instructions to ensure that changes to memory made using one alias of the physical memory are visible to accesses to the other alias of the physical memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.5 The VMSAv8-32 Short-descriptor translation table format

The Short-descriptor translation table format supports a memory map based on memory sections or pages:

**Supersections** Consist of 16MB blocks of memory. Support for Supersections is optional, except that an implementation that supports more than 32 bits of Physical Address must also support Supersections to provide access to the entire Physical Address space.

**Sections** Consist of 1MB blocks of memory.

**Large pages** Consist of 64KB blocks of memory.

**Small pages** Consist of 4KB blocks of memory.

Supersections, Sections and Large pages map large regions of memory using only a single TLB entry.

**Note**

Whether a VMSAv8-32 implementation of the Short-descriptor format translation tables supports supersections is IMPLEMENTATION DEFINED.

When using the Short-descriptor translation table format, two levels of translation tables are held in memory:

**Level 1 table**

Holds *level 1 descriptors* that contain the base address and

- Translation properties for a Section and Supersection.
- Translation properties and pointers to a level 2 table for a Large page or a Small page.

**Level 2 tables**

Hold *level 2 descriptors* that contain the base address and translation properties for a Small page or a Large page. With the Short-descriptor format, level 2 tables can be referred to as *Page tables*.

A level 2 table requires 1KB of memory.

In the translation tables, in general, a descriptor is one of:

- An invalid or fault entry.
- A page table entry, that points to a next-level translation table.
- A page or section entry, that defines the memory properties for the access.
- A reserved format.

Bits[1:0] of the descriptor give the primary indication of the descriptor type.

Figure G4-3 on page G4-4010 gives a general view of address translation when using the Short-descriptor translation table format.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

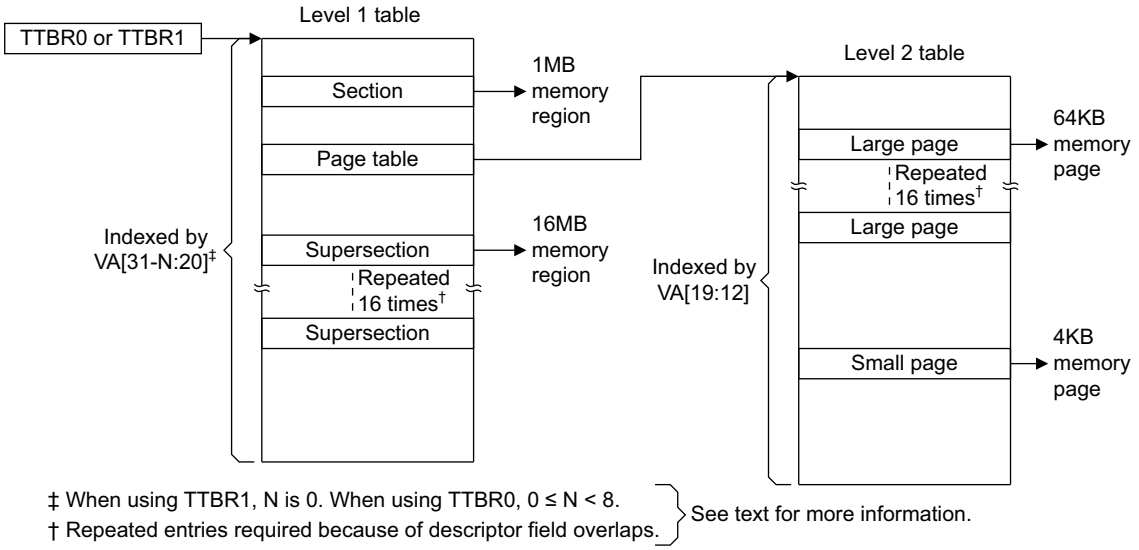



**Figure G4-3 General view of address translation using VMSAv8-32 Short-descriptor format translation tables**

*Additional requirements for Short-descriptor format translation tables* on page G4-4013 describes why, when using the Short-descriptor format, Supersection and Large page entries must be repeated 16 times, as shown in Figure G4-3.

*VMSAv8-32 Short-descriptor translation table format descriptors*, *Memory attributes in the VMSAv8-32 Short-descriptor translation table format descriptors* on page G4-4013, and *Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format* on page G4-4014 describe the format of the descriptors in the Short-descriptor format translation tables.

The following sections then describe the use of this translation table format:

- *Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format* on page G4-4014.
- *Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format* on page G4-4015.

### G4.5.1 VMSAv8-32 Short-descriptor translation table format descriptors

The following sections describe the formats of the entries in the Short-descriptor translation tables:

- *Short-descriptor translation table level 1 descriptor formats* on page G4-4011.
- *Short-descriptor translation table level 2 descriptor formats* on page G4-4012.

For more information about level 2 translation tables see *Additional requirements for Short-descriptor format translation tables* on page G4-4013.

**Note**

Previous versions of the *ARM Architecture Reference Manual*, and some other documentation, describes the AP[2] bit in the translation table entries as the APX bit.

*Information returned by a translation table lookup* on page G4-4005 describes the classification of the non-address fields in the descriptors as address map control, access control, or attribute fields.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Short-descriptor translation table level 1 descriptor formats

Each entry in the level 1 table describes the mapping of the associated 1MB VA range.

Figure G4-4 shows the possible level 1 descriptor formats.

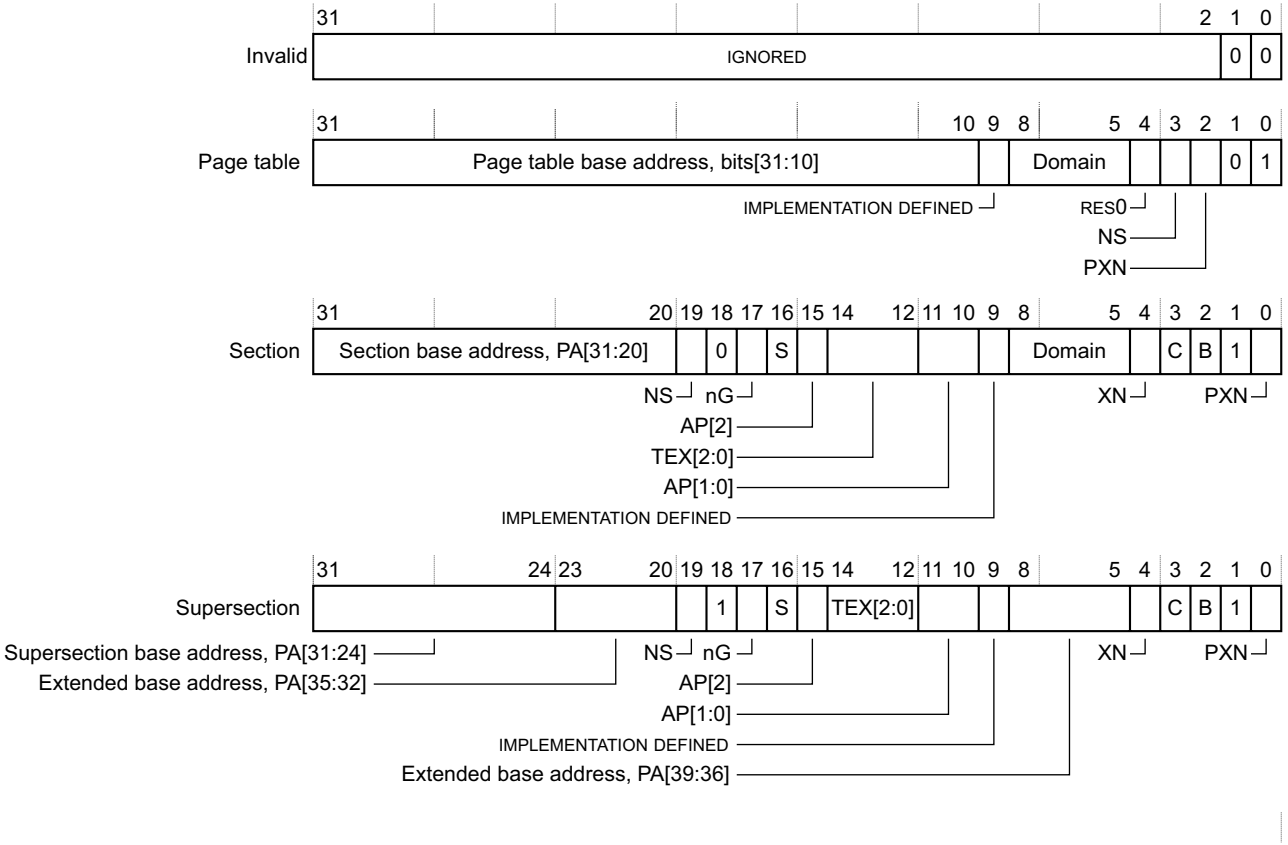



**Figure G4-4 VMSAv8-32 Short-descriptor level 1 descriptor formats**

Descriptor bits[1:0] identify the descriptor type. The encoding of these bits is:

**`0b00`, Invalid entry**

The associated VA is unmapped, and any attempt to access it generates a Translation fault.

Bits[31:2] of the descriptor are IGNORED, see *IGNORED* on page Glossary-5766. This means software can use these bits for its own purposes.

**`0b01`, Page table**

The descriptor gives the address of a level 2 translation table, that specifies the mapping of the associated 1MByte VA range.

**`0b10`, Section or Supersection**

The descriptor gives the base address of the Section or Supersection. Bit[18] determines whether the entry describes a Section or a Supersection.

This encoding also defines the PXN bit as 0.

**`0b11`, Section or Supersection, if the implementation supports the PXN attribute**

This encoding is identical to `0b10`, except that it defines the PXN bit as 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

A VMSAv8-32 implementation can use the Short-descriptor translation table format for the PL1&0 stage 1 translations, by setting TTBCR.EAE to 0.

The address information in the level 1 descriptors is:

**Page table** Bits[31:10] of the descriptor are bits[31:10] of the address of a Page table.

**Section** Bits[31:20] of the descriptor are bits[31:20] of the address of the Section.

**Supersection** Bits[31:24] of the descriptor are bits[31:24] of the address of the Supersection.
Optionally, bits[8:5, 23:20] of the descriptor are bits[39:32] of the extended Supersection address.

For the Non-secure PL1&0 translation tables, the address in the descriptor is the IPA of the Page table, Section, or Supersection. Otherwise, the address is the PA of the Page table, Section, or Supersection.

For descriptions of the other fields in the descriptors, see *Memory attributes in the VMSAv8-32 Short-descriptor translation table format descriptors* on page G4-4013.

## Short-descriptor translation table level 2 descriptor formats

Figure G4-5 shows the possible formats of a level 2 descriptor.

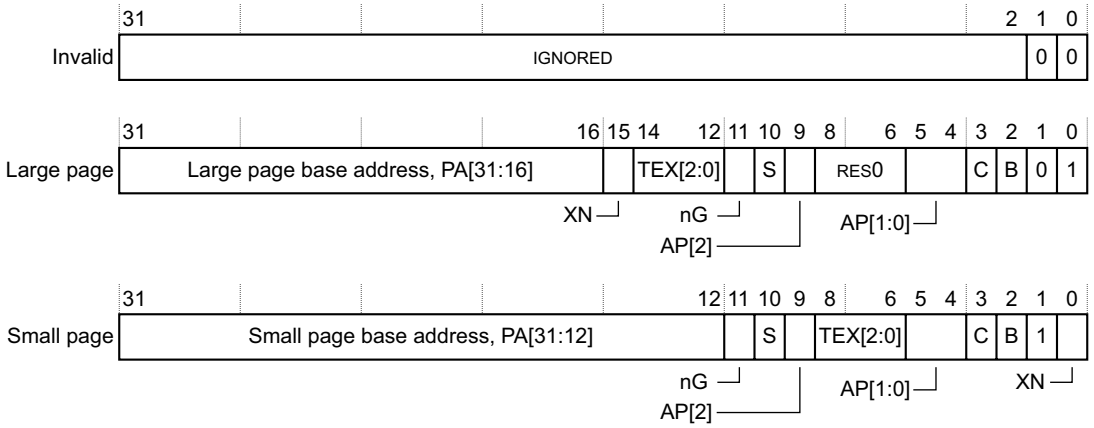



**Figure G4-5 Short-descriptor level 2 descriptor formats**

Descriptor bits[1:0] identify the descriptor type. The encoding of these bits is:

**`0b00`, Invalid entry**

The associated VA is unmapped, and attempting to access it generates a Translation fault.

Bits[31:2] of the descriptor are IGNORED, see *IGNORED* on page Glossary-5766. This means software can use these bits for its own purposes.

**`0b01`, Large page**

The descriptor gives the base address and properties of the Large page.

**`0b1x`, Small page**

The descriptor gives the base address and properties of the Small page.

In this descriptor format, bit[0] of the descriptor is the XN bit.

The address information in the level 2 descriptors is:

**Large page** Bits[31:16] of the descriptor are bits[31:16] of the address of the Large page.

**Small page** Bits[31:12] of the descriptor are bits[31:12] of the address of the Small page.

For the Non-secure PL1&0 translation tables, the address in the descriptor is the IPA of the Page table, Section, or Supersection. Otherwise, the address is the PA of the Page table, Section, or Supersection.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For descriptions of the other fields in the descriptors, see *Memory attributes in the VMSAv8-32 Short-descriptor translation table format descriptors*.

### Additional requirements for Short-descriptor format translation tables

When using Supersection or Large page descriptors in the Short-descriptor translation table format, the input address field that defines the Supersection or Large page descriptor address overlaps the table address field. In each case, the size of the overlap is 4 bits. The following diagrams show these overlaps:

- Figure K7-14 on page K7-5586 for the level 1 translation table Supersection entry.
- Figure K7-16 on page K7-5588 for the level 2 translation table Large page table entry.

Considering the case of using Large page table descriptors in a level 2 translation table, this overlap means that for any specific Large page, the bottom four bits of the level 2 translation table entry might take any value from `0b0000` to `0b1111`. Therefore, each of these sixteen index values must point to a separate copy of the same descriptor.

This means that each Large page or Supersection descriptor must:

- Occur first on a sixteen-word boundary.
- Be repeated in 16 consecutive memory locations.

## G4.5.2 Memory attributes in the VMSAv8-32 Short-descriptor translation table format descriptors

This section describes the descriptor fields other than the descriptor type field and the address field:

**TEX[2:0], C, B**

Memory region attribute bits, see *Memory region attributes* on page G4-4046.

These bits are not present in a Page table entry.

**XN bit** The Execute-never bit. Determines whether the PE can execute software from the addressed region, see *Execute-never restrictions on instruction fetching* on page G4-4040.

This bit is not present in a Page table entry.

**PXN bit** The Privileged execute-never bit. Determines whether the PE can execute software from the region when executing at PL1, see *Execute-never restrictions on instruction fetching* on page G4-4040.

**Note**

Memory accesses by software executing at EL2 always use the Long-descriptor translation table format.

When this bit is set to 1 in the Page table descriptor, it indicates that all memory pages described in the corresponding page table are Privileged execute-never.

**NS bit** Non-secure bit. Specifies whether the translated PA is in the Secure or Non-secure address map, see *Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format* on page G4-4014.

This bit is not present in level 2 descriptors. The value of the NS bit in the level 1 Page table descriptor applies to all entries in the corresponding level 2 translation table.

**Domain** Domain field, see *Domains, Short-descriptor format only* on page G4-4042.

This field is not present in a Supersection entry. Memory described by Supersections is in domain 0.

This bit is not present in level 2 descriptors. The value of the Domain field in the level 1 Page table descriptor applies to all entries in the corresponding level 2 translation table.

**An IMPLEMENTATION DEFINED bit**

This bit is not present in level 2 descriptors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**AP[2], AP[1:0]**

Access Permissions bits, see *Memory access control* on page G4-4037.

AP[0] can be configured as the *Access flag*, see *The Access flag* on page G4-4043.

These bits are not present in a Page table entry.

**S bit** Shareable bit. Used in determining the Shareability of the addressed region, see *Memory region attributes* on page G4-4046.

**Note**

The naming of this bit as the *Shareable* bit is carried forward from early versions of the ARM architecture. This name is no longer an adequate description of the interpretation of the bit.

This bit is not present in a Page table entry.

**nG bit** The not global bit. Determines how the translation is marked in the TLB, see *Global and process-specific translation table entries* on page G4-4058.

This bit is not present in a Page table entry.

**Bit[18], when bits[1:0] indicate a *Section or Supersection* descriptor**

**0** Descriptor is for a Section.

**1** Descriptor is for a Supersection.

### G4.5.3 Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format

*Access to the Secure or Non-secure physical address map* on page G4-4007 describes how the NS bit in the translation table entries:

- For accesses from Secure state, determines whether the access is to Secure or Non-secure memory.
- Is ignored by accesses from Non-secure state.

In the Short-descriptor translation table format, the NS bit is defined only in the level 1 translation tables. This means that, in a level 1 Page table descriptor, the NS bit defines the physical address space, Secure or Non-secure, for all of the Large pages and Small pages of memory described by that table.

The NS bit of a level 1 Page table descriptor has no effect on the physical address space in which that translation table is held. As stated in *Secure and Non-secure address spaces* on page G4-4007, the physical address of that translation table is in:

- The Secure address space if the translation table walk is in Secure state.
- The Non-secure address space if the translation table walk is in Non-secure state.

This means the granularity of the Secure and Non-secure memory spaces is 1MB. However, in these memory spaces, table entries can define physical memory regions with a granularity of 4KB.

### G4.5.4 Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format

As described in *Determining the translation table base address in the VMSAv8-32 translation regimes* on page G4-4006, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. When using the Short-descriptor translation table format, the value of TTBCR.N indicates the number of most significant bits of the input VA that determine whether TTBR0 or TTBR1 holds the required translation table base address, as follows:

- If N == 0 then use TTBR0. Setting TTBCR.N to zero disables use of a second set of translation tables.
- if N > 0 then:
  - If bits[31:32-N] of the input VA are all zero then use TTBR0.
  - Otherwise use TTBR1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G4-4 shows how the value of N determines the lowest address translated using TTBR1, and the size of the level 1 translation table addressed by TTBR0.

**Table G4-4 Effect of TTBCR.N on address translation, Short-descriptor format**

| TTBCR.N | First address translated with TTBR1 | TTBR0 table | |
|---|---|---|---|
| | | Size | Index range |
| 0b000 | TTBR1 not used | 16KB | VA[31:20] |
| 0b001 | 0x80000000 | 8KB | VA[30:20] |
| 0b010 | 0x40000000 | 4KB | VA[29:20] |
| 0b011 | 0x20000000 | 2KB | VA[28:20] |
| 0b100 | 0x10000000 | 1KB | VA[27:20] |
| 0b101 | 0x08000000 | 512 bytes | VA[26:20] |
| 0b110 | 0x04000000 | 256 bytes | VA[25:20] |
| 0b111 | 0x02000000 | 128 bytes | VA[24:20] |

Whenever TTBCR.N is nonzero, the size of the translation table addressed by TTBR1 is 16KB.

Figure G4-6 shows how the value of TTBCR.N controls the boundary between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

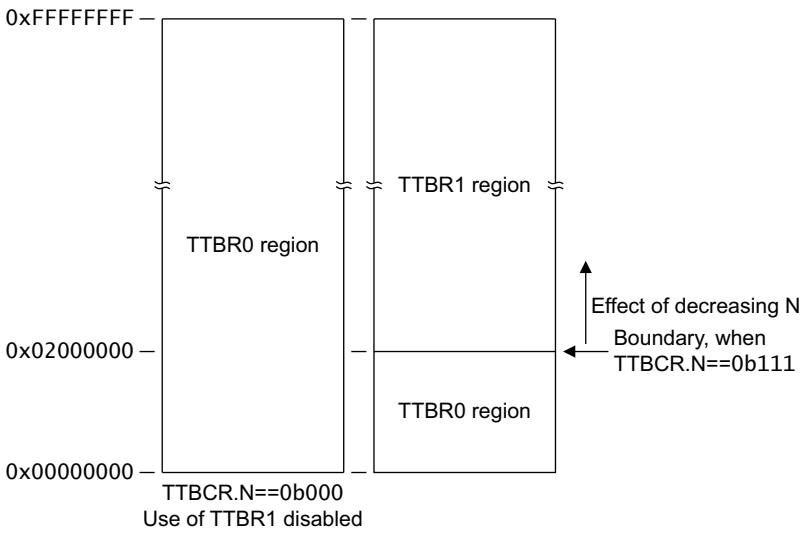



**Figure G4-6 How TTBCR.N controls the boundary between the TTBRs, Short-descriptor format**

In the selected TTBR, bits RGN, S and IRGN[1:0] define the memory region attributes for the translation table walk.

*Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format* describes the translation.

### G4.5.5 Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format

When using the Short-descriptor translation table format, and a memory access requires a translation table walk:

- A section-mapped access only requires a read of the level 1 translation table.
- A page-mapped access also requires a read of the level 2 translation table.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Reading a level 1 translation table* describes how either TTBR1 or TTBR0 is used, with the accessed VA, to determine the address of the level1 descriptor.

*Reading a level 1 translation table* shows the output address as A[39:0]:

- For a Non-secure PL1&0 stage 1 translation, this is the IPA of the required descriptor. A Non-secure PL1&0 stage 2 translation of this address is performed to obtain the PA of the descriptor.
- Otherwise, this address is the PA of the required descriptor.

*The full translation flow for Sections, Supersections, Small pages and Large pages* on page G4-4017 then shows the complete translation flow for each valid memory access.

## Reading a level 1 translation table

When performing a fetch based on TTBR0:

- The address bits taken from TTBR0 vary between bits[31:14] and bits[31:7].
- The address bits taken from the VA, that is the input address for the translation, vary between bits[31:20] and bits[24:20].

The width of the TTBR0 and VA fields depend on the value of TTBCR.N, as Figure G4-7 shows.

When performing a fetch based on TTBR1, Bits TTBR1[31:14] are concatenated with bits[31:20] of the VA. This makes the fetch equivalent to that shown in Figure G4-7, with N==0.

**Note**

See *The address and Properties fields shown in the translation flows* on page K7-5590 for more information about the *Properties* label used in this and other figures.

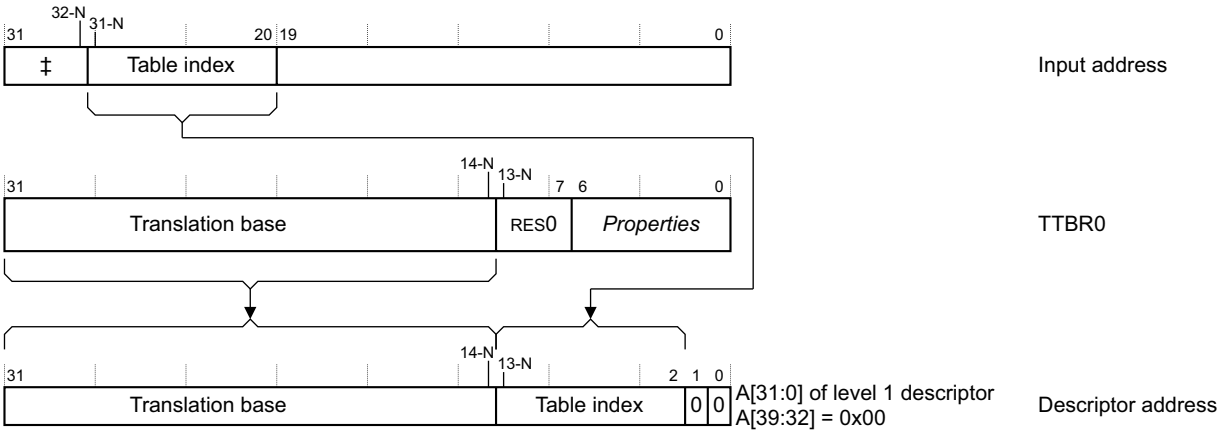



**Figure G4-7 Accessing level 1 translation table based on TTBR0, Short-descriptor format**

Regardless of which register is used as the base for the fetch, the resulting output address selects a four-byte translation table entry that is one of:

- A level 1 descriptor for a Section or Supersection.
- A *Page table* descriptor that points to a level 2 translation table. In this case:
  - A second fetch is performed to retrieve a level 2 descriptor.
  - The descriptor also contains some attributes for the access, see Figure G4-4 on page G4-4011.
- A faulting entry.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### The full translation flow for Sections, Supersections, Small pages and Large pages

In a translation table walk, only the initial lookup uses the translation table base address from the appropriate TTBR. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

*Address translation examples using the VMSAv8-32 Short descriptor translation table format* on page K7-5585 shows the full translation flow for each of the memory section and page options. As described in *VMSAv8-32 Short-descriptor translation table format descriptors* on page G4-4010, these options are:

**Supersection** A 16MB memory region, see *Translation flow for a Supersection* on page K7-5586.

**Section** A 1MB memory region, see *Translation flow for a Section* on page K7-5587.

**Large page** A 64KB memory region, described by the combination of:

- A level 1 translation table entry that indicates a level 2 Page table address.
- A level 2 descriptor that indicates a Large page.

See *Translation flow for a Large page* on page K7-5588.

**Small page** A 4KB memory region, described by the combination of:

- A level 1 translation table entry that indicates a level 2 Page table address.
- A level 2 descriptor that indicates a Small page.

See *Translation flow for a Small page* on page K7-5589.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.6 The VMSAv8-32 Long-descriptor translation table format

The VMSAv8-32 Long-descriptor translation table format supports the assignment of memory attributes to memory *Pages,* at a granularity of 4KB, across the complete input address range. It also supports the assignment of memory attributes to *blocks* of memory, where a block can be 2MB or 1GB.

**Note**

- Although the VMSAv8-32 Long-descriptor format is limited to three levels of address lookup, its design and naming conventions support extension to additional levels, to support a larger input address range.
- Similarly, while the VMSAv8-32 implementation limits the output address range to 40 bits, its design supports extension to a larger output address range.

Figure G4-2 on page G4-3996 shows the different address translation stages. The Long-descriptor translation table format:

- Is used for:
  - The Non-secure PL2 stage 1 translation.
  - The Non-secure PL1&0 stage 2 translation.
- Can be used for the Secure and Non-secure PL1&0 translations.

When used for a stage 1 translation, the translation tables support an input address of up to 32 bits, corresponding to the VA address range of the PE.

When used for a stage 2 translation, the translation tables support an input address range of up to 40 bits, to support the translation from IPA to PA. If the input address for the stage 2 translation is a 32-bit address then this address is zero-extended to 40 bits.

**Note**

When the Short-descriptor translation table format is used for the Non-secure stage 1 translations, this generates 32-bit IPAs. These are zero-extended to 40 bits to provide the input address for the stage 2 translation.

*Overview of VMSAv8-32 address translation using Long-descriptor translation tables* on page G4-4019 summarizes address translation from AArch32 state when using the Long-descriptor format translation tables.

The following sections then describe the format of the descriptors in the Long-descriptor format translation tables:

- *VMSAv8-32 Long-descriptor translation table format descriptors* on page G4-4019.
- *Memory attributes in the VMSAv8-32 Long-descriptor translation table format descriptors* on page G4-4022.
- *Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format* on page G4-4025.

The following sections then describe this translation table format:

- *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G4-4026.
- *VMSAv8-32 Long-descriptor translation table format address lookup levels* on page G4-4029.
- *Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format* on page G4-4032.
- *The algorithm for finding the translation table entries, VMSAv8-32 Long-descriptor format* on page G4-4035.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.6.1 Overview of VMSAv8-32 address translation using Long-descriptor translation tables

Figure G4-8 gives a general view of VMSAv8-32 stage 1 address translation when using the Long-descriptor translation table format.

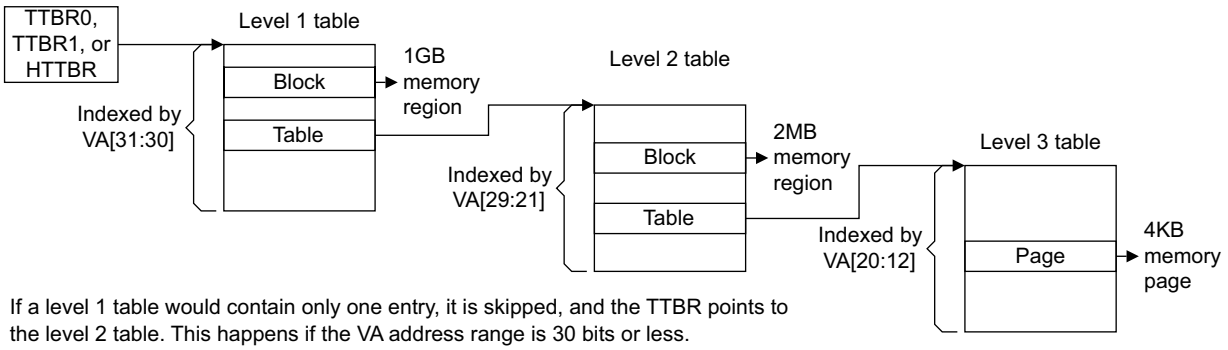



**Figure G4-8 General view of VMSAv8-32 stage 1 address translation using Long-descriptor format**

Figure G4-9 gives a general view of VMSAv8-32 stage 2 address translation. Stage 2 translation always uses the Long-descriptor translation table format.

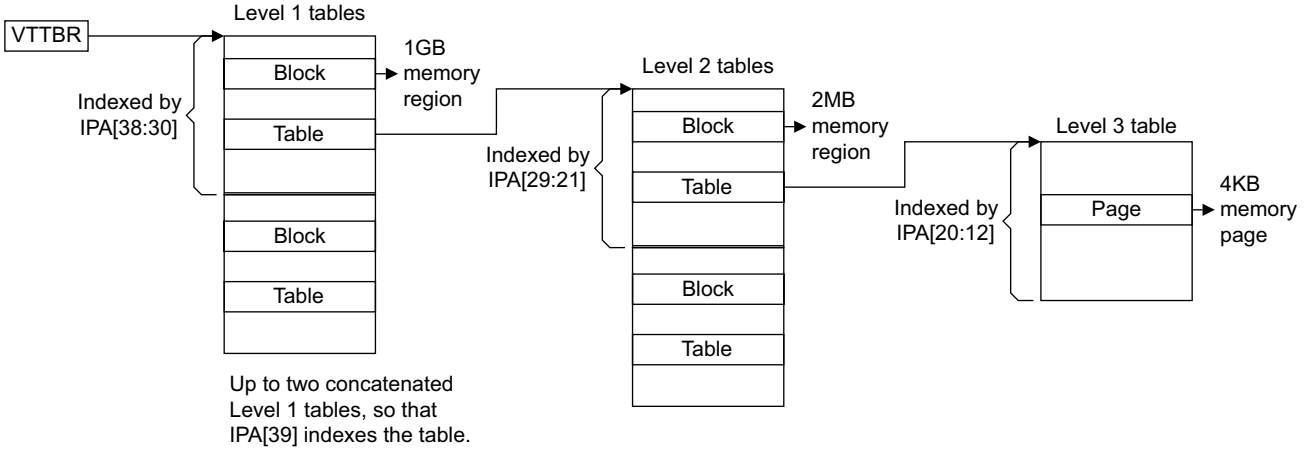



**Figure G4-9 General view of VMSAv8-32 stage 2 address translation, Long-descriptor translation table format**

*Use of concatenated translation tables for stage 2 translations* on page G4-4029 describes how using concatenated level 2 tables means lookup can start at level 2, as referred to in Figure G4-9.

### G4.6.2 VMSAv8-32 Long-descriptor translation table format descriptors

As described in *VMSAv8-32 Long-descriptor translation table format address lookup levels* on page G4-4029, the Long-descriptor translation table format provides up to three levels of address lookup. A translation table walk starts either at level 1 or level 2 of the address lookup.

In general, a descriptor is one of:

- An invalid or fault entry.
- A table entry, that points to the next-level translation table.
- A block entry, that defines the memory properties for the access.
- A reserved format.

Bit[1] of the descriptor indicates the descriptor type, and bit[0] indicates whether the descriptor is valid.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following sections describe the Long-descriptor translation table descriptor formats:

- *VMSAv8-32 Long-descriptor level 1 and level 2 descriptor formats*.
- *VMSAv8-32 Long-descriptor translation table level 3 descriptor formats* on page G4-4021.

*Information returned by a translation table lookup* on page G4-4005 describes the classification of the non-address fields in the descriptors between *address map control*, *access controls*, and *region attributes*.

## VMSAv8-32 Long-descriptor level 1 and level 2 descriptor formats

In the Long-descriptor translation tables, the formats of the level 1 and level 2 descriptors differ only in the size of the block of memory addressed by the block descriptor. A block entry:

- In a level 1 table describes the mapping of the associated 1GB input address range.
- In a level 2 table describes the mapping of the associated 2MB input address range.

Figure G4-10 shows the Long-descriptor level 1 and level 2 descriptor formats:

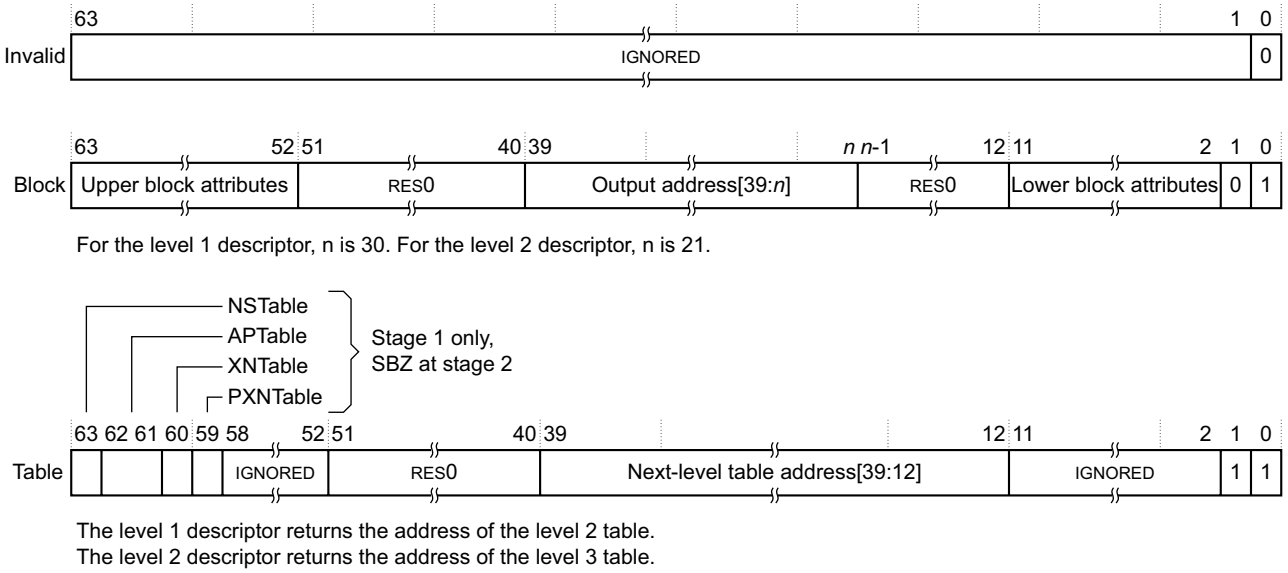



**Figure G4-10 VMSAv8-32 Long-descriptor level 1and level 2 descriptor formats**

### *Descriptor encodings, Long-descriptor level 1 and level 2 formats*

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

**0, Block** The descriptor gives the base address of a block of memory, and the attributes for that memory region.

**1, Table** The descriptor gives the address of the next level of translation table, and for a stage 1 translation, some attributes for that translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The other fields in the valid descriptors are:

**Block descriptor**

Gives the base address and attributes of a block of memory:

- For a level 1 Block descriptor, bits[39:30] are bits[39:30] of the output address that specifies a 1GB block of memory.
- For a level 2 Block descriptor, bits[39:21] are bits[39:21] of the output address that specifies a 2MB block of memory.

Bits[63:52, 11:2] provide attributes for the target memory block, see *Memory attributes in the VMSAv8-32 Long-descriptor translation table format descriptors* on page G4-4022. The position and contents of these bits is identical in the level 2 block descriptor and in the level 3 page descriptor.

**Table descriptor**

Bits[39:m] are bits[39:m] of the address of the required next-level table. Bits[m-1:0] of the table address are zero:

- For a level 1 Table descriptor, this is the address of a level 2 table.
- For a level 2 Table descriptor, this is the address of a level 3 table.

For a stage 1 translation only, bits[63:59] provide attributes for the next-level lookup, see *Memory attributes in the VMSAv8-32 Long-descriptor translation table format descriptors* on page G4-4022.

If the translation table defines the Non-secure PL1&0 stage 1 translations, then the output address in the descriptor is the IPA of the target block or table. Otherwise, it is the PA of the target block or table.

## VMSAv8-32 Long-descriptor translation table level 3 descriptor formats

Each entry in a level 3 table describes the mapping of the associated 4KB input address range.

Figure G4-11 shows the Long-descriptor level 3 descriptor formats.

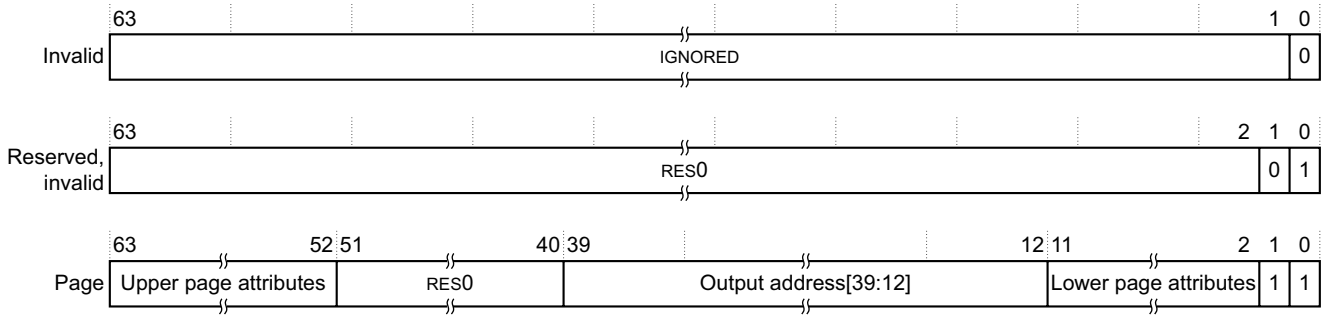



**Figure G4-11 VMSAv8-32 Long-descriptor level 3 descriptor formats**

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

**0, Reserved, invalid**

Behaves identically to encodings with bit[0] set to 0.

This encoding must not be used in level 3 translation tables.

**1, Page** Gives the address and attributes of a 4KB page of memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

At this level, the only valid format is the Page descriptor. The other fields in the Page descriptor are:

**Page descriptor**

Bits[39:12] are bits[39:12] of the output address for a page of memory.

Bits[63:52, 11:2] provide attributes for the target memory page, see *Memory attributes in the VMSAv8-32 Long-descriptor translation table format descriptors*. The position and contents of these bits are identical in the level 1 block descriptor and in the level 2 block descriptor.

If the translation table defines the Non-secure PL1&0 stage 1 translations, then the output address in the descriptor is the IPA of the target page. Otherwise, it is the PA of the target page.

### G4.6.3 Memory attributes in the VMSAv8-32 Long-descriptor translation table format descriptors

The memory attributes in the VMSAv8-32 Long-descriptor translation tables are based on those in the Short-descriptor translation table format, with some extensions. *Memory region attributes* on page G4-4046 describes these attributes. In the Long-descriptor translation table format:

- Table entries for stage 1 translations define attributes for the next level of lookup, see *Next-level attributes in VMSAv8-32 Long-descriptor stage 1 Table descriptors*
- Block and Page entries define memory attributes for the target block or page of memory. Stage 1 and stage 2 translations have some differences in these attributes, see:
  - *Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors* on page G4-4023.
  - *Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors* on page G4-4024.

#### Next-level attributes in VMSAv8-32 Long-descriptor stage 1 Table descriptors

In a Table descriptor for a stage 1 translation, bits[63:59] of the descriptor define the following attributes for the next-level translation table access:

**NSTable, bit[63]** For memory accesses from Secure state, specifies the Security state for subsequent levels of lookup, see *Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format* on page G4-4025.

For memory accesses from Non-secure state, this bit is ignored.

**APTable, bits[62:61]** Access permissions limit for subsequent levels of lookup, see *Hierarchical control of access permissions, Long-descriptor format* on page G4-4038.

APTable[0] is reserved, SBZ, in the Non-secure PL2 stage 1 translation tables.

**XNTable, bit[60]** XN limit for subsequent levels of lookup, see *Hierarchical control of instruction fetching, Long-descriptor format* on page G4-4041.

**PXNTable, bit[59]** PXN limit for subsequent levels of lookup, see *Hierarchical control of instruction fetching, Long-descriptor format* on page G4-4041.

This bit is RES0 in the Non-secure PL2 stage 1 translation tables.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors

Block and Page descriptors split the memory attributes into an upper block and a lower block. Figure G4-12 shows the memory attribute fields in these blocks, for a stage 1 translation:

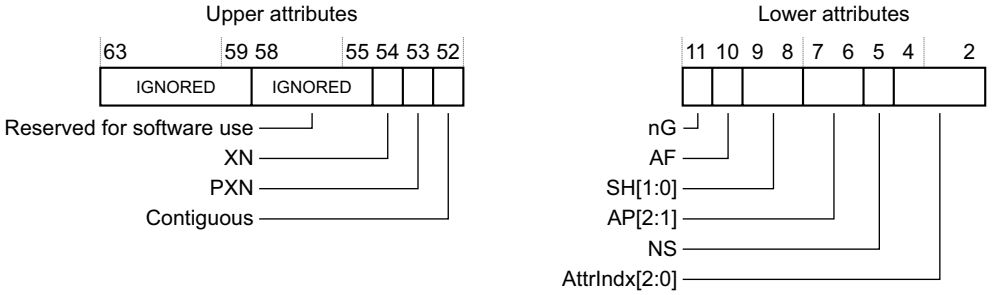



**Figure G4-12 VMSAv8-32 memory attribute fields in Long-descriptor stage 1 Block and Page descriptors**

For a stage 1 descriptor, the attributes are:

**XN, bit[54]** The Execute-never bit. Determines whether the region is executable, see *Execute-never restrictions on instruction fetching* on page G4-4040.

**PXN, bit[53]** The Privileged execute-never bit. Determines whether the region is executable at EL1, see *Execute-never restrictions on instruction fetching* on page G4-4040.

This bit is RES0 in the Non-secure PL2 stage 1 translation tables.

**Contiguous, bit[52]**

Indicates that 16 adjacent translation table entries point to contiguous memory regions, see *Contiguous bit* on page G4-4053.

**nG, bit[11]** The not global bit. Determines how the translation is marked in the TLB, see *Global and process-specific translation table entries* on page G4-4058.

This bit is RES0 in the Non-secure PL2 stage 1 translation tables.

**AF, bit[10]** The Access flag, see *The Access flag* on page G4-4043.

**SH, bits[9:8]** Shareability field, see *Memory region attributes* on page G4-4046.

**AP[2:1], bits[7:6]**

Access Permissions bits, see *Memory access control* on page G4-4037.

**Note**

For consistency with the Short-descriptor translation table formats, the Long-descriptor format defines AP[2:1] as the Access Permissions bits, and does not define an AP[0] bit.

AP[1] is RES1 in the Non-secure PL2 stage 1 translation tables.

**NS, bit[5]** Non-secure bit. For memory accesses from Secure state, specifies whether the output address is in Secure or Non-secure memory, see *Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format* on page G4-4025.

For memory accesses from Non-secure state, this bit is RES0 and is ignored by the PE.

**AttrIndx[2:0], bits[4:2]**

Stage 1 memory attributes index field, for the indicated Memory Attribute Indirection Register, see *VMSAv8-32 Long-descriptor format memory region attributes* on page G4-4052.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see *IGNORED* on page Glossary-5766. For more information about these fields see *Other fields in the Long-descriptor translation table format descriptors* on page G4-4053.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors

Block and Page descriptors split the memory attributes into an upper block and a lower block. Figure G4-13 shows the memory attribute fields in these blocks, for a stage 2 translation:

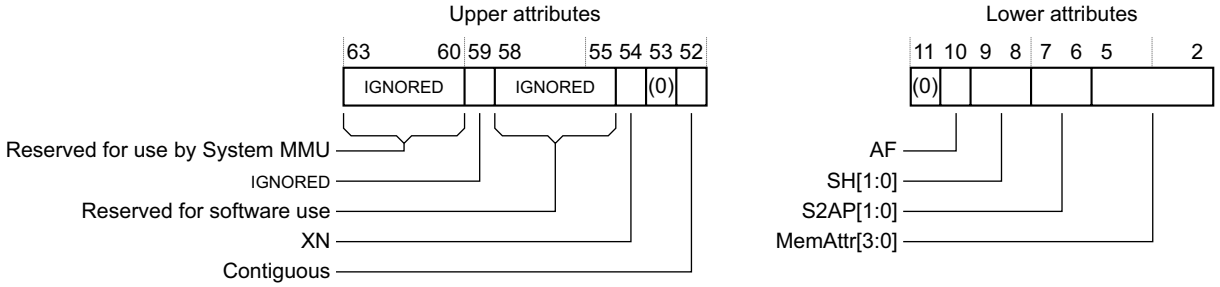



**Figure G4-13 VMSAv8-32 memory attribute fields in Long-descriptor stage 2 Block and Page descriptors**

For a stage 2 descriptor, the attributes are:

**XN, bit[54]** The Execute-never bit. Determines whether the region is executable, see *Execute-never restrictions on instruction fetching* on page G4-4040.

**Contiguous, bit[52]**

Indicates that 16 adjacent translation table entries point to contiguous memory regions, see *Contiguous bit* on page G4-4053.

**AF, bit[10]** The Access flag, see *The Access flag* on page G4-4043.

**SH, bits[9:8]** Shareability field, see *EL2 control of Non-secure memory region attributes* on page G4-4053.

**S2AP, bits[7:6]**

Stage 2 Access Permissions bits, see *Hyp mode control of Non-secure access permissions* on page G4-4044.

**Note**

In the original VMSAv7-32 Long-descriptor attribute definition, this field was called HAP[2:1], for consistency with the AP[2:1] field in the stage 1 descriptors and despite there being no HAP[0] bit. ARMv8 renames the field for greater clarity.

**MemAttr, bits[5:2]**

Stage 2 memory attributes, see *EL2 control of Non-secure memory region attributes* on page G4-4053.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see *IGNORED* on page Glossary-5766. For more information about these fields see *Other fields in the Long-descriptor translation table format descriptors* on page G4-4053.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.6.4 Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format

*Access to the Secure or Non-secure physical address map* on page G4-4007 describes how the NS bit in the translation table entries:

- For accesses from Secure state, determines whether the access is to Secure or Non-secure memory.
- Is ignored by accesses from Non-secure state.

In the Long-descriptor format:

- The NS bit relates only to the memory block or page at the output address defined by the descriptor.
- The descriptors also include an NSTable bit, see *Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format*.

The NS and NSTable bits are valid only for memory accesses from Secure state. Memory accesses from Non-secure state ignore the values of these bits.

#### Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format

For Long-descriptor format table descriptors for stage 1 translations, the descriptor includes an NSTable bit, that indicates whether the table identified in the descriptor is in Secure or Non-secure memory. For accesses from Secure state, the meaning of the NSTable bit is:

**NSTable == 0** The defined table address is in the Secure physical address space. In the descriptors in that translation table, NS bits and NSTable bits have their defined meanings.

**NSTable == 1** The defined table address is in the Non-secure physical address space. Because this table is fetched from the Non-secure address space, the NS and NSTable bits in the descriptors in this table must be ignored. This means that, for this table:

- The value of the NS bit in any block or page descriptor is ignored. The block or page address refers to Non-secure memory.
- The value of the NSTable bit in any table descriptor is ignored, and the table address refers to Non-secure memory. When this table is accessed, the NS bit in any block or page descriptor is ignored, and all descriptors in the table refer to Non-secure memory.

In addition, an entry fetched in Secure state is treated as non-global if it is read from Non-secure memory. That is, these entries must be treated as if nG==1, regardless of the value of the nG bit. For more information about the nG bit, see *Global and process-specific translation table entries* on page G4-4058.

The effect of NSTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to NSTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

**Note**

- When using the Long-descriptor format, table descriptors are defined only for the level 1 and level 2 of lookup.
- Stage 2 translations are performed only for operations in Non-secure state, that can access only the Non-secure address space. Therefore, the stage 2 descriptors do not include NS or NSTable bits.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.6.5 Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format

As described in *Determining the translation table base address in the VMSAv8-32 translation regimes* on page G4-4006, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. The Long-descriptor translation table format provides more flexibility in defining the boundary between using TTBR0 and using TTBR1. When a PL1&0 stage 1 address translation is enabled, TTBR0 is always used. If TTBR1 is also used then:

- TTBR1 is used for the top part of the input address range.
- TTBR0 is used for the bottom part of the input address range.

The TTBCR.T0SZ and TTBCR.T1SZ size fields control the use of TTBR0 and TTBR1, as Table G4-5 shows.

**Table G4-5 Use of TTBR0 and TTBR1, Long-descriptor format**

| TTBCR | | Input address range using: | |
|---|---|---|---|
| **T0SZ** | **T1SZ** | **TTBR0** | **TTBR1** |
| 0b000 | 0b000 | All addresses | Not used |
| *M*[a] | 0b000 | Zero to $(2^{(32-M)}-1)$ | $2^{32-M}$ to maximum input address |
| 0b000 | *N*[a] | Zero to $(2^{32}-2^{(32-N)}-1)$ | $2^{32}-2^{(32-N)}$ to maximum input address |
| *M*[a] | *N*[a] | Zero to $(2^{(32-M)}-1)$ | $2^{32}-2^{(32-N)}$ to maximum input address |

a. *M*, *N* must be greater than 0.The maximum possible value for each of T0SZ and T1SZ is 7.

For stage 1 translations, the input address is always a VA, and the maximum possible VA is $(2^{32}-1)$.

When address translation is using the Long-descriptor translation table format:

- Figure G4-14 shows how, when TTBCR.T1SZ is zero, the value of TTBCR.T0SZ controls the boundary between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

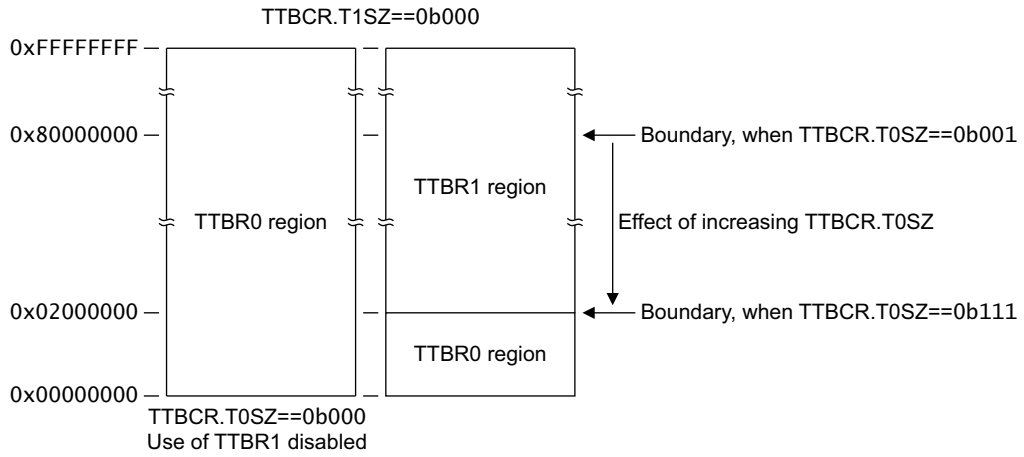



**Figure G4-14 Control of TTBR boundary, when TTBCR.T1SZ is zero**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Figure G4-15 shows how, when TTBCR.T1SZ is nonzero, the values of TTBCR.T0SZ and TTBCR.T1SZ control the boundaries between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

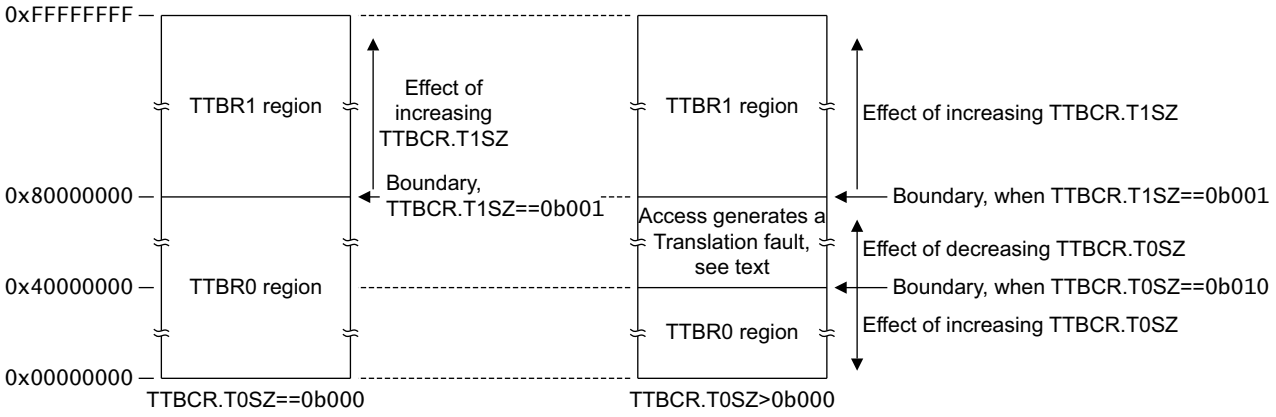



**Figure G4-15 Control of TTBR boundaries, when TTBCR.T1SZ is nonzero**

When T0SZ and T1SZ are both nonzero:

— If both fields are set to `0b001`, the boundary between the two regions is `0x80000000`. This is identical to having T0SZ set to `0b000` and T1SZ set to `0b001`.

— Otherwise, the TTBR0 and TTBR1 regions are non-contiguous. In this case, any attempt to access an address that is in that gap between the TTBR0 and TTBR1 regions generates a Translation fault.

**Note**

The handling of the Contiguous bit can mean that the boundary between the translation regions defined by the TCR_EL1.TnSZ values and the region for which an access generates a Translation fault is wider than shown in Figure G4-15. That is, if the descriptor for an access to the region shown as generating a fault has the Contiguous bit set to 1, the access might not generate a fault. *Possible translation table registers programming errors* describes this possibility.

When using the Long-descriptor translation table format:

- The TTBCR contains fields that define memory region attributes for the translation table walk, for each TTBR. These are the SH0, ORGN0, IRGN0, SH1, ORGN1, and IRGN1 bits.
- TTBR0 and TTBR1 each contain an ASID field, and the TTBCR.A1 field selects which ASID to use.

For this translation table format, *VMSAv8-32 Long-descriptor translation table format address lookup levels* on page G4-4029 summarizes the lookup levels, and *Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format* on page G4-4032 describes the possible translations.

### Possible translation table registers programming errors

In all the descriptions in this subsection, the *size of the input address* supported for a PL1&0 stage 1 translation refers to the size specified by a TTBCR.TxSZ field.

**Note**

For a PL1&0 stage 1 translation, this section has described how the input address range can be split so that the lower addresses are translated by TTBR0 and the higher addresses are translated by TTBR1. In this case, each of input address sizes specified by TTBCR.{T0SZ, T1SZ} is smaller than the total address size supported by the stage of translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following are possible errors in the programming of TTBR0, TTBR1, and TTBCR. For the translation of a particular address at a particular stage of translation, either:

- The block size being used to translate the address is larger than the size of the input address supported at a stage of translation used in performing the required translation. This can occur only for the PL1&0 stage 1 translations, and only when either TTBCR.T0SZ or TTBCR.T1SZ is zero, meaning there is no gap between the address range translated by TTBR0 and the range translated by TTBR1. In this case, this programming error occurs if a block translated from the region that has TxSZ set to zero straddles the boundary between the two address ranges. Example G4-2 shows an example of this mis-programming.
- The address range translated by a set of blocks marked as contiguous, by use of the contiguous bit, is larger than the size of the input address supported at a stage of translation used in performing the required translation.

**Example G4-2 Translation table programming error**

---

If TTBCR.T0SZ is programmed to 0 and TTBCR.T1SZ is programmed to 7, this means:

- TTBR0 translates addresses in the range `0x00000000-0xFDFFFFFF`.
- TTBR1 translates addresses in the range `0xFE000000-0xFFFFFFFF`.

The translation table indicated by TTBR0 might be programmed with a block entry for a 1GB region starting at `0xC0000000`. This covers the address range `0xC0000000-0xFFFFFFFF`, that overlaps the TTBR1 address range. This means this block size is larger than the input address size supported for translations using TTBR0, and therefore this is a programming error.

To understand why this must be a programming error, consider a memory access to address `0xFFFF0000`. According to the TTBCR.{T0SZ, T1SZ} values, this must be translated using TTBR1. However, the access matches a TLB entry for the translation, using TTBR0, of the block at `0xC0000000`. Hardware is not required to detect that the access to `0xFFFF0000` is being translated incorrectly.

---

In these cases, an implementation might use one of the following approaches:

- Treat such a block, that might be a block within a contiguous set of blocks, as causing a Translation fault, even though the block is valid, and the address accessed within that block is within the size of the input address supported at a stage of translation.
- Treat such a block, that might be a block within a contiguous set of blocks, as not causing a Translation fault, even though the address accessed within that block is outside the size of the input address supported at a stage of translation, provided that both of the following apply:
  - The block is valid.
  - At least one address within the block, or contiguous set of blocks, is within the size of the input address supported at a stage of translation.

Additional constraints apply to programming the VTCR, see *Determining the required initial lookup level for stage 2 translations* on page G4-4034.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### G4.6.6 VMSAv8-32 Long-descriptor translation table format address lookup levels

As stated at the start of this section, because the Long-descriptor translation table format is used for the Non-secure PL1&0 stage 2 translations, the format must support input addresses of up to 40 bits.

Table G4-6 summarizes the properties of the different levels of address lookup when using this format.

**Table G4-6 Properties of the three levels of address lookup with VMSAv8-32 Long-descriptor translation tables**

| Level | Input address | | Output address[a] | | Number of entries |
|---|---|---|---|---|---|
| | Size | Address range[b] | Size | Address range | |
| First | Up to 512GB | Up to Address[38:0] | 1GB | Address[39:30] | Up to 512 |
| Second | Up to 1GB | Up to Address[29:0] | 2MB | Address[39:21] | Up to 512 |
| Third | 2MB | Address[20:0] | 4KB | Address[39:12] | 512 |

a. Output address when an entry addresses a block of memory or a memory page. If an entry addresses the next level of address lookup it specifies Address[39:12] for the next-level translation table.

b. Input address range for the translation table. See *Use of concatenated level 1 translation tables* on page G4-4030 for details of support for additional bits of address at a given level, including possible support of a 40-bit input address range for stage 2 translations at level 1. For stage 1 translations at level 1 the input address range is limited to the VA size of [31:0].

For level 1 and level 2 tables, reducing the input address range reduces the number of addresses in the table and therefore reduces the table size. The appropriate Translation Table Control Register specifies the input address range.

Stage 1 translations require an input address range of up to 32 bits, corresponding to VA[31:0]. For these translations:

- For a memory access from a mode other than Hyp mode, the Secure or Non-secure TTBR0 or TTBR1 holds the translation table base address, and the Secure or Non-secure TTBCR is the control register.
- For a memory access from Hyp mode, HTTBR holds the translation table base address, and HTCR is the control register.

**Note**

For translations controlled by TTBR0 and TTBR1, if neither TTBR has an input address range larger than 1GB, then translation starts at level 2. Together, TTBR0 and TTBR1 can still cover the 32-bit VA input address range.

Stage 2 translations require an input address range of up to 40 bits, corresponding to IPA[39:0], and the supported input address size is configurable in the range 25-40 bits. Table G4-6 indicates a requirement for the translation mechanism to support a 39-bit input address range, Address[38:0]. *Use of concatenated translation tables for stage 2 translations* describes how a 40-bit IPA address range is supported. For stage 2 translations:

- VTTBR holds the translation table base address, and VTCR is the control register.
- If a supplied input address is larger than the configured input address size, a Translation fault is generated.

#### Use of concatenated translation tables for stage 2 translations

If a stage 2 translation requires 16 entries or fewer in its top-level translation table, that stage of translation can instead:

- Require the corresponding number of concatenated translation tables at the next translation level, aligned to the size of the block of concatenated translation tables.
- Start the translation at that next translation level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Stage 2 translations always use the Long-descriptor translation table format.

Use of this translation scheme is:

- Required when the stage 2 translation supports a 40-bit input address range, see *Use of concatenated level 1 translation tables*.
- Supported for a stage 2 translation with an input address range of 31-34 bits, see *Use of concatenated level 2 translation tables*.

**Note**

This translation scheme:

- Avoids the overhead of an additional level of translation
- Requires the software that is defining the translation to:
  - Define the concatenated translation tables with the required overall alignment.
  - Program VTTBR to hold the address of the first of the concatenated translation tables.
  - Program VTCR to indicate the required input address range and initial lookup level.

### *Use of concatenated level 1 translation tables*

The Long-descriptor format translation tables provide 9 bits of address resolution at each level of lookup. However, a 40-bit input address range with a translation granularity of 4KB requires a total of 28 bits of address resolution. Therefore, a stage 2 translation that supports a 40-bit input address range requires two concatenated level 1 translation tables, together aligned to 8KB, where:

- The table at the address with PA[12:0]==`0b0_0000_0000_0000` defines the translations for input addresses with bit[39]==0.
- The table at the address with PA[12:0]==`0b1_0000_0000_0000` defines the translations for input addresses with bit[39]==1.
- The 8KB alignment requirement means that both tables have the same value for PA[39:13].

### *Use of concatenated level 2 translation tables*

A stage 2 translation with an input address range of 31-34 bits can start the translation either:

- With a level 1 lookup, accessing a level 1 translation table with 2-16 entries.
- With a level 2 lookup, accessing a set of concatenated level 2 translation tables.

Table G4-7 on page G4-4031 shows these options, for each of the input address ranges that can use this scheme.

**Note**

Because these are stage 2 translations, the input address range is an IPA range.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-7 Possible uses of concatenated translation tables for level 2 lookup**

| Input address range | | Lookup starts at level 1 | Lookup starts at level 2 | |
|---|---|---|---|---|
| IPA range | Size | Required level 1 entries | Number of concatenated tables | Required alignment[a] |
| IPA[30:0] | $2^{31}$ bytes | 2 | 2 | 8KB |
| IPA[31:0] | $2^{32}$ bytes | 4 | 4 | 16KB |
| IPA[32:0] | $2^{33}$ bytes | 8 | 8 | 32KB |
| IPA[33:0] | $2^{34}$ bytes | 16 | 16 | 64KB |

a. Required alignment of the set of concatenated level 2 tables.

See also *Determining the required initial lookup level for stage 2 translations* on page G4-4034.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### G4.6.7 Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format

Figure G4-2 on page G4-3996 shows the possible address translations. If a stage of translation is controlled from an Exception level that is using AArch32, the input and output address constraints and the registers that control the translation are as follows:

**Stage 1 translations**

For all stage 1 translations:

- The input address range is up to 32 bits, as determined by either:
  - TTBCR.T0SZ or TTBCR.T1SZ, for a PL1&0 stage 1 translation.
  - HTCR.T0SZ, for a PL2 stage 1 translation.
- The output address range is 40 bits.

The stage 1 translations are:

**Non-secure PL1&0 stage 1 translation**

The stage 1 translation for memory accesses from Non-secure modes other than Hyp mode. This translates a VA to an IPA. For this translation, when Non-secure EL1 is using AArch32:

- Non-secure TTBR0 or TTBR1 holds the translation table base address.
- Non-secure TTBCR determines which TTBR is used.

**Non-secure PL2 stage 1 translation**

The stage 1 translation for memory accesses from Hyp mode, translates a VA to a PA. For this translation, when EL2 is using AArch32, HTTBR holds the translation table base address.

**Secure PL1&0 stage 1 translation**

The stage 1 translation for memory accesses from Secure modes, translates a VA to a PA. For this translation, when the Secure PL1 modes are using AArch32:

- Secure TTBR0 or TTBR1 holds the translation table base address.
- Secure TTBCR determines which TTBR is used.

**Stage 2 translation**

**Non-secure PL1&0 stage 2 translation**

The stage 2 translation for memory accesses from Non-secure modes other than Hyp mode, and translates an IPA to a PA. For this translation, when EL2 is using AArch32:

- The input address range is 40 bits, as determined by VTCR.T0SZ.
- The output address range depends on the implemented memory system, and is up to 40 bits.
- VTTBR holds the translation table base address.
- VTCR specifies the required input address range, and whether the initial lookup is at level 1 or at level 2.

The descriptions of the VMSAv8-32 translation stages state that the maximum output address size is 40 bits. However, the register and Long-descriptor format descriptor fields that hold these addresses are 48 bits wide. If bits[47:40] of an output address are not all zero then the address generates an Address size fault.

The Long-descriptor translation table format provides up to three levels of address lookup, as described in *VMSAv8-32 Long-descriptor translation table format address lookup levels* on page G4-4029, and the initial lookup, in which the MMU reads the translation table base address, is at either level 1or level 2. The following determines the level of the initial lookup:

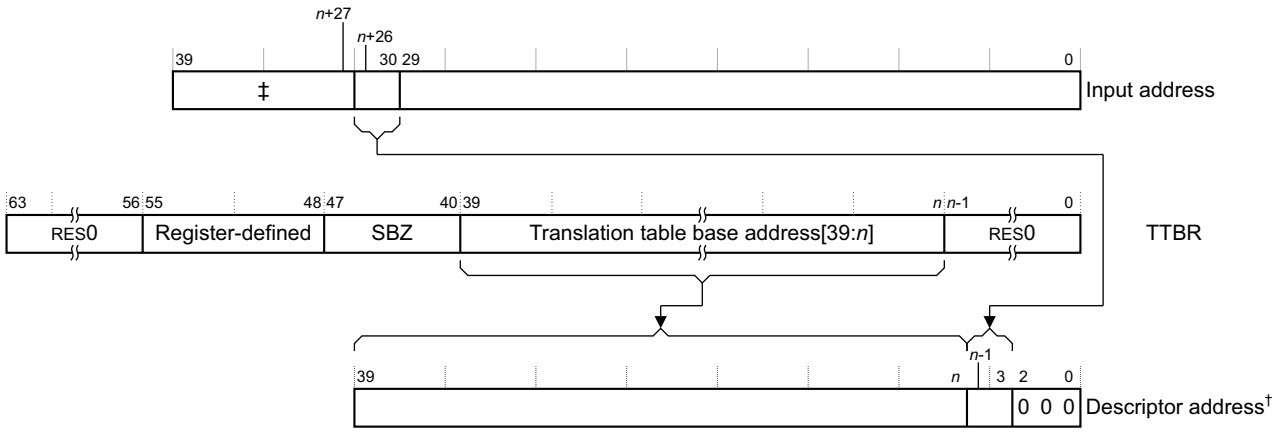
- For a stage 1 translation, the required input address range. For more information see *Determining the required initial lookup level for stage 1 translations* on page G4-4034.
- For a stage 2 translation, the level specified by the VTCR.SL0 field. For more information see *Determining the required initial lookup level for stage 2 translations* on page G4-4034.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

For a stage 2 translation, the size of the required input address range constrains the VTCR.SL0 value.

Figure G4-16 shows how the descriptor address for the initial lookup for a translation using the Long-descriptor translation table format is determined from the input address and the TTBR value. This figure shows the lookup for a translation that starts with a level 1 lookup, that translates bits[39:30] of the input address, zero extended if necessary.




See text for more information about the translation table base register used, and the value of *n*.

‡ This field is absent if *n* is 13.

† For a Non-secure PL1&0 stage 1 translation, the IPA of the descriptor. Otherwise, the PA of the descriptor.

**Figure G4-16 VMSAv8-32 Long-descriptor initial lookup, starting at level 1**

For a translation that starts with a level 1 lookup, as shown in Figure G4-16:

**For a stage 1 translation**

*n* is in the range 4-5 and:

- For a memory access from Hyp mode:
  - HTTBR is the TTBR.
  - *n*=5-(HTCR.T0SZ).
- For other accesses:
  - The Secure or Non-secure instance of TTBR0 or TTBR1 is the TTBR.
  - *n*=(5-TTBCR.T*x*SZ), where *x* is 0 when using TTBR0, and 1 when using TTBR1.

**For a stage 2 translation**

*n* is in the range 4-13 and:

- VTTBR is the TTBR.
- *n*=5-(VTCR.T0SZ).

For a translation that starts with a level 2 lookup, the descriptor address is obtained in the same way, except that bits[(*n*+17):21] of the input address provide bits[(*n*-1):3] of the descriptor address, where:

**For a stage 1 translation**

*n* is in the range 7-12. As *Determining the required initial lookup level for stage 1 translations* on page G4-4034 shows, for a stage 1 translation to start with a level 2 lookup, the corresponding T0SZ or T1SZ field must be 2 or more. This means:

- For a memory access from Hyp mode, *n*=14-HTCR.T0SZ.
- For other memory accesses, *n*=14-(TTBCR.T*x*SZ), where *x* is 0 when using TTBR0, and 1 when using TTBR1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**For a stage 2 translation**

*n* is in the range 7-16. For a stage 2 translation to start with a level 2 lookup, VTCR.SL0 is `0b00`, and *n*=14-(VTCR.T0SZ).

The following sections describe how the level of the initial lookup is determined:

- *Determining the required initial lookup level for stage 1 translations*.
- *Determining the required initial lookup level for stage 2 translations*.

*Address translation examples using the VMSAv8-32 Long descriptor translation table format* on page K7-5590 shows examples of full translation flows, to an entry for a 4KB memory page, for lookups starting at level 1 and lookups starting at level 2.

## Determining the required initial lookup level for stage 1 translations

For a stage 1 translation, the required input address range, indicated by a T0SZ or T1SZ field in a translation table control register, determines the initial lookup level. The size of this input address region is $2^{(32-TxSZ)}$ bytes, and if this size is:

- Less than or equal to $2^{30}$ bytes, the required start is at level 2, and translation requires two levels of table to map to 4KB pages. This corresponds to a T*x*SZ value of 2 or more.
- More than $2^{30}$ bytes, the required start is at level 1, and translation requires three levels of table to map to 4KB pages. This corresponds to a T*x*SZ value that is less than 2.

For the PL1&0 stage 1 translations, the TTBCR:

- Splits the 32-bit VA input address range between TTBR0 and TTBR1, see *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G4-4026.
- Holds the input address range sizes for TTBR0 and TTBR1, in the TTBCR.T0SZ and TTBCR.T1SZ fields.

For the PL2 stage 1 translations, HTCR.T0SZ indicates the size of the required input address range. For example, if this field is `0b000`, it indicates a 32-bit VA input address range, and translation lookup must start at level 1.

## Determining the required initial lookup level for stage 2 translations

For a PL1&0 stage 2 translation, the output address range from the PL1&0 stage 1 translations determines the required input address range for the stage 2 translation.

VTCR.SL0 indicates the starting level for the lookup. The permitted SL0 values are:

`0b00` Stage 2 translation lookup must start at level 2.

`0b01` Stage 2 translation lookup must start at level 1.

In addition, VTCR.T0SZ must indicate the required input address range. The size of the input address region is $2^{(32-T0SZ)}$ bytes.

**Note**

VTCR.T0SZ holds a four-bit signed integer value, meaning it supports values from -8 to 7. This is different from the other translation control registers, where T*n*SZ holds a three-bit unsigned integer, supporting values from 0 to 7.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The programming of VTCR must follow the constraints shown in Table G4-8, otherwise behavior is CONSTRAINED UNPREDICTABLE with the resulting behavior being that any attempt to perform a translation table walk using these values generate a stage 2 level 1 Translation Fault. The table also shows how the VTCR.SL0 and VTCR.T0SZ values determine the VTTBR.BADDR field width.

**Table G4-8 Input address range constraints on programming VTCR**

| VTCR.SL0 | VTCR.T0SZ | Input address range, R | Initial lookup level | BADDR[39:x] width[a] |
|---|---|---|---|---|
| 0b00 | 2 to 7 | $R \le 2^{30}$ bytes | Level 2 | [39:12] to [39:7] |
| 0b00 | -2 to 1 | $2^{30} < R \le 2^{34}$ bytes | Level 2 | [39:16] to [39:13] |
| 0b01 | -2 to 1 | | Level 1 | [39:7] to [39:4] |
| 0b01 | -8 to -3 | $2^{34} < R$ | Level 1 | [39:13] to [39:8] |

a. The first range corresponds to the first T0SZ value, the second range to the second T0SZ value.

In addition, VTCR.S must be programmed to the value of T0SZ[3], otherwise behavior is CONSTRAINED UNPREDICTABLE with the resulting behavior being that VTCR.T0SZ is treated as an UNKNOWN value.

Where necessary, the initial lookup level provides multiple concatenated translation tables, as described in *Use of concatenated level 2 translation tables* on page G4-4030. This section also gives more information about the alternatives, shown in Table G4-8, when R is in the range $2^{31}$-$2^{34}$.

### G4.6.8 The algorithm for finding the translation table entries, VMSAv8-32 Long-descriptor format

This section gives the algorithms for finding the translation table entry that corresponds to a given IA, for each required level of lookup. The algorithms encode the descriptions of address translation given earlier in this section. The VMSAv8-32 Long-descriptor format uses a 4KB translation granule.

The description uses the following terms:

**BaseAddress** The base address for the level of lookup, as defined by:
- For the initial lookup level, the appropriate TTBR, see the description of *TnSZ*.
- Otherwise, the translation table address returned by the previous level of lookup.

**IA** The supplied IA for this stage of translation.

**T*n*SZ** The translation table size for this stage of translation:

**For PL1&0 stage 1** Either:
- TTBCR.T0SZ if the translation is using TTBR0.
- TTBCR.T1SZ if the translation is using TTBR1.

**For PL1&0 stage 2** VTCR.T0SZ. The translation uses VTTBR.

**For PL2 stage 1** HTCR.T0SZ. The translation uses HTTBR.

**SL0** VTCR.SL0. Applies to the Non-secure PL1&0 stage 2 translation only.

Table G4-9 on page G4-4036 shows the translation table descriptor address, for each level of lookup. The table shows only architecturally-valid programming of the TCR. See also *Possible translation table registers programming errors* on page G4-4027.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-9 Translation table entry addresses, VMSAv8-32 using Long-descriptor format**

| Lookup level | Entry address and conditions | | General conditions |
|---|---|---|---|
| | Stage 1 translation | Stage 2 translation | |
| One | BaseAddr[39:*x*]:IA[*y*:30]:0b000<br>if 0 ≤ T*n*SZ ≤ 1 then *x* = (5 - T*n*SZ) | BaseAddr[39:*x*]:IA[*y*:30]:0b000<br>if -8 ≤ T0SZ ≤ 1 then *x* = (5 - T0SZ) | *y* = (*x* + 26) |
| Two | BaseAddr[39:*x*]:IA[*y*:21]:0b000<br>if 2 ≤ T*n*SZ ≤ 7then *x* = (14 - T*n*SZ)<br><br>else[a] *x* =12 | BaseAddr[39:*x*]:IA[*y*:21]:0b000<br>if SL0 == 0 then<br>if -2 ≤ T0SZ ≤ 7 then *x* = (14 - T0SZ)<br>elsif[a] SL0 == 1 then *x* = 12 | *y* = (*x* + 17) |
| Three | BaseAddr[39:12]:IA[20:12]:0b000 | BaseAddr[39:12]:IA[20:12]:0b000 | - |

a. This is the case where this level of lookup is not the initial level of lookup.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.7 Memory access control

In addition to an output address, a translation table entry that refers to page or region of memory includes fields that define properties of the target memory region. *Information returned by a translation table lookup* on page G4-4005 describes the classification of those fields as address map control, access control, and memory attribute fields. The access control fields, described in this section, determine whether the PE, in its current state, is permitted to perform the required access to the output address given in the translation table descriptor. If a translation stage does not permit the access then an MMU fault is generated for that translation stage, and no memory access is performed.

The following sections describe the memory access controls:

- *Access permissions*.
- *Execute-never restrictions on instruction fetching* on page G4-4040.
- *Domains, Short-descriptor format only* on page G4-4042.
- *The Access flag* on page G4-4043.
- *Hyp mode control of Non-secure access permissions* on page G4-4044.

### G4.7.1 Access permissions

**Note**

This section gives a general description of memory access permissions. Software executing at PL1 in Non-secure state can see only the access permissions defined by the Non-secure PL1&0 stage 1 translations. However, software executing at PL2 can modify these permissions, as described in *Hyp mode control of Non-secure access permissions* on page G4-4044. This modification is invisible to Non-secure software executing at EL1 or EL0.

Access permission bits in a translation table descriptor control access to the corresponding memory region. The details of this control depend on the translation table format, as follows:

**Short-descriptor format**

This format supports two options for defining the access permissions:

- Three bits, AP[2:0], define the access permissions.
- Two bits, AP[2:1], define the access permissions, and AP[0] can be used as an Access flag.

SCTLR.AFE selects the access permissions option. Setting this bit to 1, to enable the Access flag, also selects use of AP[2:1] to define access permissions.

ARM deprecates any use of the AP[2:0] scheme for defining access permissions.

**Long-descriptor format**

AP[2:1] to control the access permissions, and the descriptors provide an AF bit for use as an Access flag. This means VMSAv8-32 behaves as if the value of SCTLR.AFE is 1, regardless of the value that software has written to this bit.

**Note**

When use of the Long-descriptor format is enabled, SCTLR.AFE is UNK/SBOP.

*The Access flag* on page G4-4043 describes the Access flag, for both translation table formats.

The XN and PXN bits provide additional access controls for instruction fetches, see *Execute-never restrictions on instruction fetching* on page G4-4040.

An attempt to perform a memory access that the translation table access permission bits do not permit generates a Permission fault, for the corresponding stage of translation. However, when using the Short-descriptor translation table format, it generates the fault only if the access is to memory in the Client domain, see *Domains, Short-descriptor format only* on page G4-4042.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

For the Non-secure PL1&0 translation regime, memory accesses are subject to two stages of translation. Each stage of translation has its own, independent, fault checking. Fault handling is different for the two stages, see *Exception reporting in a VMSAv8-32 implementation* on page G4-4091.

The following sections describe the two access permissions models:

- *AP[2:1] access permissions model*.
- *AP[2:0] access permissions control, Short-descriptor format only* on page G4-4039. This section includes some information on access permission control in earlier versions of the ARM VMSA.

## AP[2:1] access permissions model

**Note**

ARM recommends that this model is always used, even where the AP[2:0] model is permitted. Some documentation describes the AP[2:1] model as the *simplified access permissions* model.

This access permissions model is used if the translation is either:

- Using the Long-descriptor translation table format.
- Using Short-descriptor translation table format, and the SCTLR.AFE bit is set to 1.

In this model:

- One bit, AP[2], selects between read-only and read/write access.
- A second bit, AP[1], selects between Application level (PL0) and System level (PL1) control.
  For the Non-secure PL2 stage 1 translations, AP[1] is SBO.

This provides four access combinations:

- Read-only at all privilege levels.
- Read/write at all privilege levels.
- Read-only at PL1, no access by software executing at PL0.
- Read/write at PL1, no access by software executing at PL0.

Table G4-10 shows this access control model.

**Table G4-10 VMSAv8-32 AP[2:1] access permissions model**

| AP[2], disable write access | AP[1], enable unprivileged access | Access |
|---|---|---|
| 0 | 0[a] | Read/write, only at PL1 |
| 0 | 1 | Read/write, at any privilege level |
| 1 | 0[a] | Read-only, only at PL1 |
| 1 | 1 | Read-only, at any privilege level |

a. Not valid for Non-secure PL2 stage 1 translation tables. AP[1] is SBO in these tables.

### *Hierarchical control of access permissions, Long-descriptor format*

The Long-descriptor translation table format introduces a mechanism that entries at one level of translation table lookup can use to set limits on the permitted entries at subsequent levels of lookup. This applies to the access permissions, and also to the restrictions on instruction fetching described in *Hierarchical control of instruction fetching, Long-descriptor format* on page G4-4041.

The restrictions apply only to subsequent levels of lookup at the same stage of translation. The APTable[1:0] field restricts the access permissions, as Table G4-11 on page G4-4039 shows.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

As stated in the table footnote, for the Non-secure PL2 stage 1 translation tables, APTable[0] is reserved, SBZ.

**Table G4-11 Effect of APTable[1:0] on subsequent levels of lookup**

| APTable[1:0] | Effect |
|---|---|
| 00 | No effect on permissions in subsequent levels of lookup. |
| 01 [a] | Access at PL0 not permitted, regardless of permissions in subsequent levels of lookup. |
| 10 | Write access not permitted, at any exception level, regardless of permissions in subsequent levels of lookup. |
| 11 [a] | Regardless of permissions in subsequent levels of lookup:<br>• Write access not permitted, at any exception level.<br>• Read access not permitted at PL0. |

a. Not valid for the Non-secure PL2 stage 1 translation tables. In those tables, APTable[0] is SBZ.

**Note**

The APTable[1:0] settings are combined with the translation table access permissions in the translation tables descriptors accessed in subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The Long-descriptor format provides APTable[1:0] control only for the stage 1 translations. The corresponding bits are SBZ in the stage 2 translation table descriptors.

The effect of APTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to APTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

### AP[2:0] access permissions control, Short-descriptor format only

This access permissions model applies when using the Short-descriptor translation tables format, and the SCTLR.AFE bit is set to 0. ARM deprecates any use of this access permissions model.

When SCTLR.AFE is set to 0, ensuring that the AP[0] bit is always set to 1 effectively changes the access model to the simpler model described in *AP[2:1] access permissions model* on page G4-4038.

Table G4-12 shows the full AP[2:0] access permissions model:

**Table G4-12 VMSAv8-32 MMU access permissions**

| AP[2] | AP[1:0] | PL1 access | Unprivileged access | Description |
|---|---|---|---|---|
| 0 | 00 | No access | No access | All accesses generate Permission faults |
| | 01 | Read/write | No access | Access only at PL1 |
| | 10 | Read/write | Read-only | Writes at PL0 generate Permission faults |
| | 11 | Read/write | Read/write | Full access |
| 1 | 00 | - | - | Reserved |
| | 01 | Read-only | No access | Read-only, only at PL1 |
| | 10 | Read-only | Read-only | Read-only at any exception level, deprecated [a] |
| | 11 | Read-only | Read-only | Read-only at any exception level [b] |

a. From VMSAv7, ARM strongly recommends use of the `0b11` encoding for Read-only at any exception level.

b. This mapping was introduced in VMSAv7, and is reserved in earlier versions of the VMSA.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- VMSAv8-32 supports the full set of access permissions shown in Table G4-12 on page G4-4039 only when SCTLR.AFE is set to 0. When SCTLR.AFE is set to 1, the only supported access permissions are those described in *AP[2:1] access permissions model* on page G4-4038.
- Some old documentation describes the AP[2] bit in the translation table entries as the APX bit.

### G4.7.2 Execute-never restrictions on instruction fetching

Execute-never (XN) controls provide an additional level of control on memory accesses permitted by the access permissions settings. These controls are:

**XN, Execute-never**

When the XN bit is 1, a Permission fault is generated if the PE attempts to execute an instruction from the corresponding memory region. However, when using the Short-descriptor translation table format, the fault is generated only if the access is to memory in the Client domain, see *Domains, Short-descriptor format only* on page G4-4042. A PE can execute instructions from a memory region only if the access permissions for its current state permit read access, and the value of the XN bit is 0.

**PXN, Privileged execute-never**

When the PXN bit is 1, a Permission fault is generated if the PE is executing at PL1 and attempts to execute an instruction from the corresponding memory region. As with the XN bit, when using the Short-descriptor translation table format, the fault is generated only if the access is to memory in the Client domain.

In both the Short-descriptor format and the Long-descriptor format translation tables, all descriptors for memory blocks and pages always include an XN bit.

Support for the PXN bit is as follows:

- The Long-descriptor translation table formats always include the PXN bit.
- All implementations must:
  - Support the use of the PXN bit.
  - Use the Short-descriptor translation table formats that include the PXN bit.

In the Non-secure PL2 stage 1 translation tables, the PXN bit is reserved, SBZ.

In addition, additional controls can enforce execute-never restrictions, regardless of the settings in the translation tables, see:

- *Restriction on Secure instruction fetch* on page G4-4042.
- *Preventing execution from writable locations* on page G4-4042.

The execute-never controls apply also to speculative instruction fetching. This means a speculative instruction fetch from a memory region that is execute-never at the current level of privilege is prohibited.

The XN control means that, when the stage of address translation is enabled, the PE can fetch, or speculatively fetch, an instruction from a memory location only if all of the following apply:

- If using the Short-descriptor translation table format, the translation table descriptor for the location does not indicate that it is in a No access domain.
- If using the Long-descriptor translation table format, or using the Short descriptor format and the descriptor indicates that the location is in a Client domain, in the descriptor for the location the following apply:
  - XN is set to 0.
  - The access permissions permit a read access from the current PE mode.
- No other Prefetch Abort condition exists.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- The PXN control applies to the PE privilege when it attempts to execute the instruction. In an implementation that fetches instructions speculatively, this might not be the privilege when the instruction was prefetched. Therefore, the architecture does not require the PXN control to prevent instruction fetching.
- Although the XN control applies to speculative fetching, on a speculative instruction fetch from an XN location, no Permission fault is generated unless the PE attempts to execute the instruction that would have been fetched from that location. This means that, if a speculative fetch from an XN location is attempted, but there is no attempt to execute the corresponding instruction, a Permission fault is not generated.
- The software that defines a translation table must mark any region of memory that is read-sensitive as XN, to avoid the possibility of a speculative fetch accessing the memory region. This means it must mark any memory region that corresponds to a read-sensitive peripheral as XN. Hardware does not prevent speculative accesses to a region of any Device memory type unless that region is also marked as execute-never for all Exception levels from which it can be accessed.
- When using the Short-descriptor translation table format, the XN attribute is not checked for domains marked as Manager. Therefore, the system must not include read-sensitive memory in domains marked as Manager, because the XN bit does not prevent speculative fetches from a Manager domain.

When no stage of address translation for the translation regime is enabled, memory regions cannot have XN or PXN attributes assigned. *Behavior of instruction fetches when all associated address translations are disabled* on page G4-4002 describes how disabling all MMUs affects instruction fetching.

### Hierarchical control of instruction fetching, Long-descriptor format

The Long-descriptor translation table format introduces a mechanism that means entries at one level of translation tables lookup can set limits on the permitted entries at subsequent levels of lookup. This applies to the restrictions on instruction fetching, and also to the access permissions described in *Hierarchical control of access permissions, Long-descriptor format* on page G4-4038.

The restrictions apply only to subsequent levels of lookup at the same stage of translation, and:

- XNTable restricts the XN control:
  - When XNTable is set to 1, the XN bit is treated as 1 in all subsequent levels of lookup, regardless of the actual value of the bit.
  - When XNTable is set to 0 it has no effect.
- PXNTable restricts the PXN control:
  - When PXNTable is set to 1, the PXN bit is treated as 1 in all subsequent levels of lookup, regardless of the actual value of the bit.
  - When PXNTable is set to 0 it has no effect.

**Note**

The XNTable and PXNTable settings are combined with the XN and PXN bits in the translation table descriptors accessed at subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The XNTable and PXNTable controls are provided only in the Long-descriptor translation table format, and only for stage 1 translations. The corresponding bits are SBZ in the stage 2 translation table descriptors.

The effect of XNTable or PXNTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to XNTable or PXNTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Restriction on Secure instruction fetch

EL3 provides a Secure instruction fetch bit, SCR.SIF. When this bit is set to 1, any attempt in Secure state to execute an instruction fetched from Non-secure physical memory causes a Permission fault. As with all Permission fault checking, when using the Short-descriptor format translation tables the check applies only to Client domains, see *Access permissions* on page G4-4037.

ARM expects SCR.SIF to be static during normal operation. In particular, whether the TLB holds the effect of the SIF bit is IMPLEMENTATION DEFINED. The generic sequence to ensure visibility of a change to the SIF bit is:

```
Change the SCR.SIF bit
ISB                     ; This ensures synchronization of the change
Invalidate entire TLB
DSB                     ; This completes the TLB Invalidation
ISB                     ; This ensures instruction synchronization
```

### Preventing execution from writable locations

The architecture includes control bits that, when the corresponding stage 1 address translation is enabled, force writable memory to be treated as XN, regardless of the setting of the XN bit. When the translation stages are controlled by an Exception level that is using AArch32:

- For PL1&0 stage 1 translations, when SCTLR.WXN is set to 1, all regions that are writable at stage 1 of the address translation are treated as XN.
- For Non-secure PL2 stage 1 translations, when HSCTLR.WXN is set to 1, all regions that are writable at stage 1 of the address translation are treated as XN.
- For PL1&0 stage 1 translations, when SCTLR.UWXN is set to 1, an instruction fetch is treated as accessing a PXN region if it accesses a region that software executing at PL0 can write to.

**Note**

Setting a WXN or UWXN bit to 1 changes the interpretation of the translation table entry, overriding a zero value of an XN or PXN field. It does not cause any change to the translation table entry.

For any given virtual machine, ARM expects WXN and UWXN to remain static in normal operation. In particular, it is IMPLEMENTATION DEFINED whether TLB entries associated with a particular VMID reflect the effect of the values of these bits. A generic sequence to ensure synchronization of a change to these bits, when that change is made without a corresponding change of VMID, is:

```
Change the WXN or UWXN bit
ISB                     ; This ensures synchronization of the change
Invalidate entire TLB of associated entries
DSB                     ; This completes the TLB Invalidation
ISB                     ; This ensures instruction synchronization
```

As with all Permission fault checking, if the stage 1 translation is using the Short-descriptor translation table format, the permission checks are performed only for Client domains. For more information see *Access permissions* on page G4-4037.

For more information about address translation see *About address translation for VMSAv8-32* on page G4-3995.

## G4.7.3 Domains, Short-descriptor format only

A domain is a collection of memory regions. The Short-descriptor translation table format supports 16 domains, and requires the software that defines a translation table to assign each VMSAv8-32 memory region to a domain. When using the Short-descriptor format:

- Level 1 translation table entries for Page tables and Sections include a domain field.
- Translation table entries for Supersections do not include a domain field. The Short-descriptor format defines Supersections as being in domain 0.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

- Level 2 translation table entries inherit a domain setting from the parent level 1 Page table entry.
- Each TLB entry includes a domain field.

The domain field specifies which of the 16 domains the entry is in, and a two-bit field in the DACR defines the permitted access for each domain. The possible settings for each domain are:

**No access** Any access using the translation table descriptor generates a Domain fault.

**Clients** On an access using the translation table descriptor, the access permission attributes are checked. Therefore, the access might generate a Permission fault.

**Managers** On an access using the translation table descriptor, the access permission attributes are not checked. Therefore, the access cannot generate a Permission fault.

See *The MMU fault-checking sequence* on page G4-4081 for more information about how, when using the Short-descriptor translation table format, the Domain attribute affects the checking of the other attributes in the translation table descriptor.

**Note**

A single program might:

- Be a Client of some domains.
- Be a Manager of some other domains.
- Have no access to the remaining domains.

The Long-descriptor translation table format does not support domains. When a stage of translation is using this format, all memory is treated as being in a Client domain, and the settings in the DACR are ignored.

## G4.7.4 The Access flag

The Access flag indicates when a page or section of memory is accessed for the first time since the Access flag in the corresponding translation table descriptor was set to 0:

- If address translation is using the Short-descriptor translation table format, it must set SCTLR.AFE to 1 to enable use of the Access flag. Setting this bit to 1 redefines the AP[0] bit in the translation table descriptors as an Access flag, and limits the access permissions information in the translation table descriptors to AP[2:1], as described in *AP[2:1] access permissions model* on page G4-4038.
- The Long-descriptor format always supports an Access flag bit in the translation table descriptors, and address translation using this format behaves as if SCTLR.AFE is set to 1, regardless of the value of that bit.

In ARMv8 the Access flag is managed by software as described in the following subsection.

**Note**

Previous version of the ARM architecture optionally supported hardware management of the Access flag. ARMv8 obsoletes this option.

### Software management of the Access flag

An implementation that requires software to manage the Access flag generates an Access flag fault whenever a translation table entry with the Access flag set to 0 is read into the TLB

**Note**

When using the Short-descriptor translation table format, Access flag faults are generated only if SCTLR.AFE is set to 1, to enable use of a translation table descriptor bit as an Access flag.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The Access flag mechanism expects that, when an Access flag fault occurs, software resets the Access flag to 1 in the translation table entry that caused the fault. This prevents the fault occurring the next time that memory location is accessed. Entries with the Access flag set to 0 are never held in the TLB, meaning software does not have to flush the entry from the TLB after setting the flag.

### G4.7.5 Hyp mode control of Non-secure access permissions

When EL2 is using AArch32, Non-secure software executing in Hyp mode controls two sets of translation tables, both of which use the Long-descriptor translation table format:

- The translation tables that control the Non-secure PL2 stage 1 translations. These map VAs to PAs, for memory accesses made when executing in Non-secure state in Hyp mode, and are indicated and controlled by the HTTBR and HTCR.

  These translations have similar access controls to other Non-secure stage 1 translations using the Long-descriptor translation table format, as described in:
  - *AP[2:1] access permissions model* on page G4-4038.
  - *Execute-never restrictions on instruction fetching* on page G4-4040.

  The differences from the Non-secure stage 1 translations are that:
  - The APTable[0], PXNTable, and PXN bits are reserved, SBZ.
  - AP[1] is reserved, SBO.

- The translation tables that control the Non-secure PL1&0 stage 2 translations. These map the IPAs from the stage 1 translation onto PAs, for memory accesses made when executing in Non-secure state at PL1 or PL0, and are indicated and controlled by the VTTBR and VTCR.

  The descriptors in the virtualization translation tables define stage 2 access permissions, that are combined with the permissions defined in the stage 1 translation. This section describes this combining of access permissions.

**Note**

The level 2 access permissions mean a hypervisor can define additional access restrictions to those defined by a Guest OS in the stage 1 translation tables. For a particular access, the actual access permission is the more restrictive of the permissions defined by:

- The Guest OS, in the stage 1 translation tables.
- The hypervisor, in the stage 2 translation tables.

The stage 2 access controls defined from Hyp mode:

- Affect only the Non-secure stage 1 access permissions settings.
- Take no account of whether the accesses are from a Non-secure PL1 mode or a Non-secure PL0 mode.
- Permit software executing in Hyp mode to assign a write-only attribute to a memory region.

The S2AP field in the stage 2 descriptors define the stage 2 access permissions, as Table G4-13 shows:

**Table G4-13 Stage 2 control of access permissions**

| S2AP | Access permission |
|---|---|
| 00 | No access permitted |
| 01 | Read-only. Writes to the region are not permitted, regardless of the stage 1 permissions. |
| 10 | Write-only. Reads from the region are not permitted, regardless of the stage 1 permissions. |
| 11 | Read/write. The stage 1 permissions determine the access permissions for the region. |

For more information about the S2AP field see *Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors* on page G4-4024.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the stage 2 permissions cause a Permission fault, this is a stage 2 MMU fault. Stage 2 MMU faults are taken to Hyp mode, and reported in the HSR using an EC code of `0x20` or `0x24`. For more information, see *Use of the HSR on page G4-4105*.

**Note**

In the HSR, the combination of the EC code and the DFSC or IFSC value in the ISS indicate that the fault is a stage 2 MMU fault.

The stage 2 permissions include an XN attribute. If this is set to 1, execution from the region is not permitted, regardless of the value of the XN attribute in the stage 1 translation. If a Permission fault is generated because the stage 2 XN bit is set to 1, this is reported as a stage 2 MMU fault.

*AArch32 prioritization of synchronous aborts from a single stage of address translation on page G4-4088* describes the abort prioritization if both stages of a translation generate a fault.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G4.8 Memory region attributes

In addition to an output address, a translation table entry that refers to a page or region of memory includes fields that define properties of that target memory region. *Information returned by a translation table lookup* on page G4-4005 describes the classification of those fields as address map control, access control, and memory attribute fields. The memory region attribute fields control the memory type, Cacheability, and Shareability of the region.

The following sections describe the assignment of memory region attributes for stage 1 translations:

- *Overview of memory region attributes for stage 1 translations*.
- *Short-descriptor format memory region attributes, without TEX remap* on page G4-4047.
- *Short-descriptor format memory region attributes, with TEX remap* on page G4-4049.
- *VMSAv8-32 Long-descriptor format memory region attributes* on page G4-4052.

For an implementation that is operating in Secure state, or in Hyp mode, these assignments define the memory attributes of the accessed region.

For an implementation that is operating in a Non-secure PL1 or PL0 mode, the Non-secure PL1&0 stage 2 translation can modify the memory attributes assigned by the stage 1 translation. *EL2 control of Non-secure memory region attributes* on page G4-4053 describes these stage 2 assignments.

### G4.8.1 Overview of memory region attributes for stage 1 translations

The description of the memory region attributes in a translation descriptor divides into:

**Memory type and attributes**

These are described either:

- Directly, by bits in the translation table descriptor.
- Indirectly, by registers referenced by bits in the table descriptor. This is described as *remapping* the memory type and attribute description.

The Short-descriptor translation table format can use either of these approaches, selected by the SCTLR.TRE bit:

**TRE == 0** Remap disabled. The TEX[2:0], C, and B bits in the translation table descriptor define the memory region attributes. *Short-descriptor format memory region attributes, without TEX remap* on page G4-4047 describes this encoding.

**Note**

With the Short-descriptor format, remapping is called *TEX remap*, and the SCTLR.TRE bit is the *TEX remap enabled* bit.

The description of the TRE == 0 encoding includes information about the encoding in previous versions of the architecture.

**TRE == 1** Remap enabled. The TEX[0], C, and B bits in the translation table descriptor are index bits to the remap registers, that define the memory region attributes:

- The Primary Region Remap Register, PRRR.
- The Normal Memory Remap Register, NMRR.

*Short-descriptor format memory region attributes, with TEX remap* on page G4-4049 describes this encoding scheme.

This scheme reassigns translation table descriptor bits TEX[2:1] for use as bits managed by the operating system.

The Long-descriptor translation table format always uses remapping. This means that when the value of TTBCR.EAE is 1, enabling use of the Long-descriptor translation table format, SCTLR.TRE is RES1.

*VMSAv8-32 Long-descriptor format memory region attributes* on page G4-4052 describes this encoding.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Shareability** In the Short-descriptor translation table format, the S bit in the translation table descriptor is used in determining the Shareabilty of the region. How the S bit is interpreted depends on whether TEX remap is enabled, see:

- *Shareability and the S bit, without TEX remap* on page G4-4048.
- *Determining the Shareability, with TEX remap* on page G4-4050.

In the Long-descriptor translation table format, the SH[1:0] field in the translation table descriptor encodes the Shareability of the region, see *Shareability, Long-descriptor format* on page G4-4052.

**Note**

Shareability is one of Non-shareable, Inner Shareable, and Outer Shareable. However, when using the Short-descriptor translation table format without TEX remap, VMSAv8-32 does not support any distinction between Inner Shareable and Outer Shareable memory, and a memory region is either Non-shareable or Outer Shareable.

### G4.8.2 Short-descriptor format memory region attributes, without TEX remap

When using the Short-descriptor translation table formats, TEX remap is disabled when the value of SCTLR.TRE is 0.

**Note**

- The Short-descriptor format scheme without TEX remap is the scheme used in VMSAv6.
- The B (Bufferable), C (Cacheable), and TEX (Type extension) bit names are inherited from earlier versions of the architecture. These names no longer adequately describe the function of the B, C, and TEX bits.

Table G4-14 shows the C, B, and TEX[2:0] encodings when TEX remap is disabled. In the *Page Shareability* column, an entry of *S bit* indicates that the S bit in the translation table descriptor determines the Shareability, see *Shareability and the S bit, without TEX remap* on page G4-4048.

**Table G4-14 TEX, C, and B encodings when TRE == 0**

| TEX[2:0] | C | B | Description | Memory type | Page Shareability |
|---|---|---|---|---|---|
| 000 | 0 | 0 | Device-nGnRnE | Device-nGnRnE | Outer Shareable |
| | | 1 | Device-nGnRE[a] | Device-nGnRE | Outer Shareable[a] |
| | 1 | 0 | Outer and Inner Write-Through, no Write-Allocate | Normal | S bit |
| | | 1 | Outer and Inner Write-Back, no Write-Allocate | Normal | S bit |
| 001 | 0 | 0 | Outer and Inner Non-cacheable | Normal | Outer Shareable[b] |
| | | 1 | Reserved | - | - |
| | 1 | 0 | IMPLEMENTATION DEFINED | IMPLEMENTATION DEFINED | IMPLEMENTATION DEFINED |
| | | 1 | Outer and Inner Write-Back, Write-Allocate | Normal | S bit |
| 010 | 0 | 0 | Device-nGnRE[a] | Device-nGnRE | Outer Shareable[a] |
| | | 1 | Reserved | - | - |
| | 1 | x | Reserved | - | - |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-14 TEX, C, and B encodings when TRE == 0 (continued)**

| TEX[2:0] | C | B | Description | | Memory type | Page Shareability |
|---|---|---|---|---|---|---|
| 011 | x | x | Reserved | | - | - |
| 1BB | A | A | Cacheable memory: | AA = Inner attribute[c]<br>BB = Outer attribute | Normal | S bit |

a. In ARMv8, all Device memory types are Outer Shareable. For the Device-nGnRE memory type this is a change from previous versions of the architecture. This is why Device-nGnRE memory has two entries in this table. For compatibility with ARMv7 software should use the {TEX, C, B} values {000, 0, 1}.

b. In ARMv8, Normal Inner Non-cacheable, Outer Non-cacheable memory is always Outer Shareable. This is a change from previous versions of the architecture, where the S bit determined the Shareability. For compatibility with ARMv7 software should set S to 1.

c. For more information, see *Cacheable memory attributes, without TEX remap*.

See *Memory types and attributes* on page E2-2382 for an explanation of Normal memory, and the types of Device memory, and of the Shareability attribute.

### Cacheable memory attributes, without TEX remap

When TEX[2] == 1, the memory described by the translation table entry is cacheable, and the rest of the encoding defines the Inner and Outer Cacheability attributes:

**TEX[1:0]** Define the Outer Cacheability attribute.

**C, B** Define the Inner Cacheability attribute.

The translation table entries use the same encoding for the Outer and Inner Cacheability attributes, as Table G4-15 shows.

**Table G4-15 Inner and Outer Cacheability attribute encoding**

| Encoding | Cacheability attribute |
|---|---|
| 00 | Non-cacheable |
| 01 | Write-Back, Write-Allocate |
| 10 | Write-Through, no Write-Allocate |
| 11 | Write-Back, no Write-Allocate |

### Shareability and the S bit, without TEX remap

The Short-descriptor format translation table entries include an S bit. This bit:

- Is ignored if the entry refers to any type of Device memory, or to Normal memory that is Inner Non-cacheable, Outer Non-cacheable.
- For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, determines whether the memory region is Outer Shareable or Non-shareable:

  **S == 0** Normal memory region is Non-shareable.

  **S == 1** Normal memory region is Outer Shareable.

Without TEX remapping there is no distinction between Inner Shareable and Outer Shareable memory, meaning the S bit determines whether the region is Non-shareable or Outer Shareable.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.8.3 Short-descriptor format memory region attributes, with TEX remap

When using the Short-descriptor translation table formats, TEX remap is enabled when the value of SCTLR.TRE is 1. In this configuration:

- The software that defines the translation tables must program the PRRR and NMRR to define seven possible memory region attributes.
- The TEX[0], C, and B bits of the translation table descriptors define the memory region attributes, by indexing PRRR and NMRR.
- Hardware makes no use of TEX[2:1], see *The OS managed translation table bits* on page G4-4051.

When TEX remap is enabled:

- For seven of the eight possible combinations of the TEX[0], C and B bits, fields in the PRRR and NMRR define the region attributes, as described in this section.
- The meaning of the eighth combination for the TEX[0], C and B bits is IMPLEMENTATION DEFINED.
- If the TEX[0], C and B bits determine that the region is a Device memory type, or is Normal Inner Non-cacheable, Outer Non-cacheable, then the region is Outer Shareable. Otherwise, the Shareability is determined by the combination of:
  - The S bit from the translation table descriptor.
  - The value of the PRRR.NS0 or PRRR.NS1 bit.
  - The value of the appropriate PRRR.NOS*n* bit, as shown in Table G4-16.

  For more information, see *Determining the Shareability, with TEX remap* on page G4-4050.

For each of the possible encodings of the TEX[0], C, and B bits in a translation table entry, Table G4-16 shows which fields of the PRRR and NMRR registers describe the memory region attributes.

**Table G4-16 TEX, C, and B encodings when TRE == 1**

| Encoding | | | Memory type[a] | Cache attributes[a, b]: | | Outer Shareable attribute[a, c] |
|---|---|---|---|---|---|---|
| TEX[0] | C | B | | Inner cacheability | Outer cacheability | |
| 0 | 0 | 0 | PRRR.TR0 | NMRR.IR0 | NMRR.OR0 | NOT(PRRR.NOS0) |
| | | 1 | PRRR.TR1 | NMRR.IR1 | NMRR.OR1 | NOT(PRRR.NOS1) |
| | 1 | 0 | PRRR.TR2 | NMRR.IR2 | NMRR.OR2 | NOT(PRRR.NOS2) |
| | | 1 | PRRR.TR3 | NMRR.IR3 | NMRR.OR3 | NOT(PRRR.NOS3) |
| 1 | 0 | 0 | PRRR.TR4 | NMRR.IR4 | NMRR.OR4 | NOT(PRRR.NOS4) |
| | | 1 | PRRR.TR5 | NMRR.IR5 | NMRR.OR5 | NOT(PRRR.NOS5) |
| | 1 | 0 | IMPLEMENTATION DEFINED | | | |
| | | 1 | PRRR.TR7 | NMRR.IR7 | NMRR.OR7 | NOT(PRRR.NOS7) |

a. For details of the *Memory type* and *Outer Shareable* encodings see *PRRR, Primary Region Remap Register* on page G6-4511. For details of the *Cache attributes* encodings see Table G4-15 on page G4-4048.

b. Applies only if the memory type for the region is mapped as Normal memory.

c. Applies only if both of the following apply:
   - The memory type for the region is mapped as Normal memory that is not Inner Non-cacheable and Outer Non-cacheable.
   - The region is not Non-shareable.

   See *Determining the Shareability, with TEX remap* on page G4-4050.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

As Table G4-16 on page G4-4049 shows, when TEX remap is enabled, for a given set of {TEX[0], C, B} bits from a translation table descriptor:

1. The primary mapping, to memory type, is given by the PRRR.TR*n* field as shown in the Memory type column.
2. For any region that the PRRR.TR*n* maps as Normal memory, NMRR.IR*n* determines the Inner cacheability attribute, and NMRR.OR*n* determines the Outer cacheability attribute.
3. For a region that PRRR.TR*n* maps as Normal memory, if NMRR.{IR*n*, OR*n*} do not map the region as Inner Non-cacheable, Outer Non-cacheable, PRRR.{NS0, NS1} and PRRR.NOS*n* are used to determine the Shareability of the region, see *Determining the Shareability, with TEX remap*.

The TEX remap registers and the SCTLR.TRE bit are Banked between the Secure and Non-secure Security states. For more information, see *The effect of EL3 on TEX remap* on page G4-4052.

The TEX remap registers must be static during normal operation. In particular, when the remap registers are changed:

- It is IMPLEMENTATION DEFINED when the changes take effect.
- It is CONSTRAINED UNPREDICTABLE whether the TLB caches the effect of the TEX remap on translation tables, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5427.

The software sequence to ensure the synchronization of changes to the TEX remap registers is:

1. Execute a `DSB` instruction. This ensures any memory accesses using the old mapping have completed.
2. Write the TEX remap registers or SCTLR.TRE bit.
3. Execute an `ISB` instruction. This ensures synchronization of the register updates.
4. Invalidate the entire TLB.
5. Execute a `DSB` instruction. This ensures completion of the entire TLB operation.
6. Clean and invalidate all caches. This removes any cached information associated with the old mapping.
7. Execute a `DSB` instruction. This ensures completion of the cache maintenance.
8. Execute an `ISB` instruction. This ensures instruction synchronization.

This extends the standard rules for the synchronization of changes to System registers described in *Synchronization of changes to System registers* on page G4-4132, and provides implementation freedom as to whether or not the effect of the TEX remap is cached.

### Determining the Shareability, with TEX remap

The memory type of a region, as indicated in the Memory type column of Table G4-16 on page G4-4049, provides the first level of control of the Shareability of the region:

- If the memory is any type of Device memory, then the region is Outer Shareable, and any Shareability attributes in the translation table descriptor and PRRR for that region are ignored.

  This applies also to a Normal memory region that the NMRR attributes identify as Inner Non-cacheable and Outer Non-cacheable,

- If using the Short descriptor translation table format then the Shareability of the region is determined using the value of the S bit in the translation table descriptor to index one of the PRRR.{NS1. NS0} bits, as described in this section.

Table G4-17 shows how the translation table S bit indexes into the PRRR:

**Table G4-17 Determining the Shareability attribute, with TEX remap**

| Memory type | Remapping when S == 0 | Remapping when S == 1 |
|---|---|---|
| Device or Normal Inner Non-cacheable, Outer Non-cacheable | Outer Shareable | Outer Shareable |
| Normal, not Inner Non-cacheable, Outer Non-cacheable | PRRR.NS0 | PRRR.NS1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For a Normal memory region that is not Inner Non-cacheable, Outer Non-cacheable, the appropriate bit of the PRRR indicates whether the region is Non-shareable, as follows:

**PRRR.NS*n*==0** Non-shareable.

PRRR.{NOS7:NOS0} are ignored.

**PRRR.NS*n*==1** The appropriate PRRR.NOS*m* field, as shown in Table G4-16 on page G4-4049, indicates whether the region is Inner Shareable or Outer Shareable:

- **PRRR.NOS*m*==0** Region is Outer Shareable.
- **PRRR.NOS*m*==1** Region is Inner Shareable.

**Note**

This means that TEX remapping can map a translation table entry with S == 0 as shareable memory.

### SCTLR.TRE, SCTLR.M, and the effect of the TEX remap registers

When TEX remap is disabled, because the value of the SCTLR.TRE bit is 0:

- The effect of the PRRR and NMRR registers can be IMPLEMENTATION DEFINED.
- The interpretation of the fields of the PRRR and NMRR registers can differ from the description given earlier in this section. One implication of this is that the implementation can provide an IMPLEMENTATION DEFINED mechanism to interpret the PRRR.{NOS7:NOS0} fields.

VMSAv8-32 requires that the effect of these registers is limited to remapping the attributes of memory locations. These registers must not change whether any cache hardware or stages of address translation are enabled. The mechanism by which the TEX remap registers have an effect when the value of the SCTLR.TRE bit is 0 is IMPLEMENTATION DEFINED. The AArch32 architecture requires that from reset, if the IMPLEMENTATION DEFINED mechanism has not been invoked:

- If the PL1&0 stage 1 address translation is enabled and is using the Short-descriptor format translation tables, the architecturally-defined behavior of the TEX[2:0], C, and B bits must apply, without reference to the TEX remap functionality. In other words, memory attribute assignment must comply with the scheme described in *Short-descriptor format memory region attributes, without TEX remap* on page G4-4047.
- If the PL1&0 stage 1 address translation is disabled, then the architecturally-defined behavior of VMSAv8-32 with address translation disabled must apply, without reference to the TEX remap functionality. See *The effects of disabling address translation stages on VMSAv8-32 behavior* on page G4-4000.

Possible mechanisms for enabling the IMPLEMENTATION DEFINED effect of the TEX remap registers when the value of SCTLR.TRE is 0 include:

- A control bit in the ACTLR, or in an IMPLEMENTATION DEFINED System register.
- Changing the behavior when the PRRR and NMRR registers are changed from their IMPLEMENTATION DEFINED reset values.

In addition, if the stage of address translation is disabled and the value of the SCTLR.TRE bit is 1, the architecturally-defined behavior of the VMSAv8-32 with the stage of address translation disabled must apply without reference to the TEX remap functionality.

In an implementation that includes EL3, the IMPLEMENTATION DEFINED effect of these registers must only take effect in the Security state of the registers. See also *The effect of EL3 on TEX remap* on page G4-4052.

### The OS managed translation table bits

When TEX remap is enabled, the TEX[2:1] bits in the translation table descriptors are available as two bits that can be managed by the operating system. In VMSAv8-32, as long as the SCTLR.TRE bit is set to 1, the values of the TEX[2:1] bits are IGNORED by the PE. Software can write any value to these bits in the translation tables.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### The effect of EL3 on TEX remap

In an implementation that includes EL3, when EL3 is using AArch32, the TEX remap registers are Banked between the Secure and Non-secure Security states. When EL3 is using AArch32, write accesses to the Secure register for the current security state apply to all PL1&0 stage 1 translation table lookups in that state. The SCTLR.TRE bit is Banked in the Secure and Non-secure copies of the register, and the appropriate version of this bit determines whether TEX remap is applied to translation table lookups in the current security state.

Write accesses to the Secure copies of the TEX remap registers are disabled when the **CP15SDISABLE** input is asserted HIGH, meaning the `MCR` operations to access these registers are UNDEFINED. For more information, see *The CP15SDISABLE input signal* on page G4-4129.

## G4.8.4 VMSAv8-32 Long-descriptor format memory region attributes

When a PE is using the VMSAv8-32 Long-descriptor translation table format, the AttrIndx[2:0] field in a block or page translation table descriptor for a stage 1 translation indicates the 8-bit field, in the appropriate MAIR register, that specifies the attributes for the corresponding memory region, as follows:

- AttrIndx[2] indicates the MAIR register to be used:

  **AttrIndx[2] == 0** Use MAIR0.

  **AttrIndx[2] == 1** Use MAIR1.

- AttrIndx[2:0] indicates the required Attr field, Attr*n*, where *n* = AttrIndx[2:0].

Each AttrIndx field defines, for the corresponding memory region:

- The memory type, Normal or a type of Device memory.
- For Normal memory:
  - The inner and outer cacheability, Non-cacheable, Write-Through, or Write-Back.
  - For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate policy hints, each of which is *Allocate* or *Do not allocate*.

For more information about the AttrIndx[2:0] descriptor field, see *Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors* on page G4-4023.

### Shareability, Long-descriptor format

When a PE is using the Long-descriptor translation table format, the SH[1:0] field in a block or page translation table descriptor specifies the Shareability attributes of the corresponding memory region, if the MAIR entry for that region identifies it as Normal memory that is not both Inner Non-cacheable and Outer Non-cacheable. Table G4-18 shows the encoding of this field.

**Table G4-18 SH[1:0] field encoding for Normal memory, Long-descriptor format**

| SH[1:0] | Normal memory |
|---|---|
| 00 | Non-shareable |
| 01 | Reserved, CONSTRAINED UNPREDICTABLE, see *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427 for the permitted behavior. |
| 10 | Outer Shareable |
| 11 | Inner Shareable |

See *Combining the Shareability attribute* on page G4-4057 for constraints on the Shareability attributes of a Normal memory region that is Inner Non-cacheable, Outer Non-cacheable.

For any type of Device memory, and for Normal Inner Non-cacheable, Outer Non-cacheable memory, the value of the SH[1:0] field of the translation table descriptor is ignored.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Other fields in the Long-descriptor translation table format descriptors

The following subsections describe the other fields in the translation table block and page descriptors when a PE is using the Long-descriptor translation table format:

- *Contiguous bit*
- *IGNORED fields*.
- *Field reserved for software use*

#### *Contiguous bit*

The Long-descriptor translation table format descriptors contain a Contiguous bit. Setting this bit to 1 indicates that 16 adjacent translation table entries point to a *contiguous output address range*. These 16 entries must be aligned in the translation table so that the top five bits of their input addresses, that index their position in the translation table, are the same. For example, to use this bit for a block of 16 entries in the level 3 translation table, bits[20:16] of the input addresses for the 16 entries must be the same.

The contiguous output address range must be aligned to size of 16 translation table entries at the same translation table level.

Use of this bit means that the TLB can cache a single entry to cover the 16 translation table entries.

This bit acts as a hint. The architecture does not require a PE to cache TLB entries in this way. To avoid TLB coherency issues, any TLB maintenance by address must not assume any optimization of the TLB tables that might result from use of this bit.

**Note**

The use of the contiguous bit is similar to the approach used, in the Short-descriptor translation table format, for optimized caching of Large Pages and Supersections in the TLB. However, an important difference in the contiguous bit capability is that TLB maintenance must be performed based on the size of the underlying translation table entries, to avoid TLB coherency issues. That is, any use of the contiguous bit has no effect on the minimum size of entry that must be invalidated from the TLB.

#### *IGNORED fields*

For stage 1 and stage 2 Block and Page descriptors, the architecture defines bits[63:59] and bits[58:55] as IGNORED fields, meaning the architecture guarantees that a PE makes no use of these fields. In addition:

- Bits[58:55] are reserved for software use, see *Field reserved for software use*.
- In the stage 2 Block and Page descriptors, bits[63:60] are reserved for use by a System MMU.

**Note**

The definition of IGNORED means there is no need to invalidate the TLB if these bits are changed.

#### *Field reserved for software use*

The architecture reserves a 4-bit field in the Block and Page table descriptors, bits[58:55], for software use. In considering migration from using the Short-descriptor format to the Long-descriptor format, this field is an extension of the Short-descriptor field described in *The OS managed translation table bits* on page G4-4051.

**Note**

The definition of IGNORED means there is no need to invalidate the TLB if these bits are changed.

## G4.8.5 EL2 control of Non-secure memory region attributes

Software executing at EL2 controls two sets of translation tables, both of which use the Long-descriptor translation table format. These are:

- The translation tables that control Non-secure PL2 stage 1 translations. These map VAs to PAs, and when EL2 is using AArch32 they are indicated and controlled by the HTTBR and HTCR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

These translations have exactly the same memory region attribute controls as any other stage 1 translations, as described in *VMSAv8-32 Long-descriptor format memory region attributes* on page G4-4052.

- The translation tables that control Non-secure PL1&0 stage 2 translations. These map the IPAs from the stage 1 translation onto PAs, and are indicated and when EL2 is using AArch32 they are controlled by the VTTBR and VTCR.

  The descriptors in the virtualization translation tables define level 2 memory region attributes, that are combined with the attributes defined in the stage 1 translation. This section describes this combining of attributes.

*VMSAv8-32 Long-descriptor translation table format descriptors* on page G4-4019 describes the format of the entries in these tables.

**Note**

In a virtualization implementation, a hypervisor might usefully:

- Reduce the permitted Cacheability of a region.
- Increase the required Shareability of a region.

The combining of attributes from stage 1 and stage 2 translations supports both of these options.

In the stage 2 translation table descriptors for memory regions and pages, the MemAttr[3:0] and SH[1:0] fields describe the stage 2 memory region attributes:

- The definition of the stage 2 SH[1:0] field is identical to the same field for a stage 1 translation, see *Shareability, Long-descriptor format* on page G4-4052.
- MemAttr[3:2] give a top-level definition of the memory type, and of the cacheability of a Normal memory region, as Table G4-19 shows:

**Table G4-19 Long-descriptor MemAttr[3:2] encoding, stage 2 translation**

| MemAttr[3:2] | Memory type | Cacheability |
|---|---|---|
| 00 | Device, of type determined by MemAttr[1:0] | Not applicable |
| 01 | Normal, Inner cacheability determined by MemAttr[1:0] | Outer Non-cacheable |
| 10 | | Outer Write-Through Cacheable |
| 11 | | Outer Write-Back Cacheable |

The encoding of MemAttr[1:0] depends on the Memory type indicated by MemAttr[3:2]:

— When MemAttr[3:2]==`0b00`, indicating a type of Device memory, Table G4-20 shows the encoding of MemAttr[1:0]:

**Table G4-20 MemAttr[1:0] encoding for the types of Device memory**

| MemAttr[1:0] | Meaning when MemAttr[3:2] == `0b00` |
|---|---|
| 00 | Region is Device-nGnRnE memory |
| 01 | Region is Device-nGnRE memory |
| 10 | Region is Device-nGRE memory |
| 11 | Region is Device-GRE memory |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— When MemAttr[3:2]!=0b00, indicating Normal memory, Table G4-21 shows the encoding of MemAttr[1:0]:

**Table G4-21 MemAttr[1:0] encoding for Normal memory**

| MemAttr[1:0] | Meaning when MemAttr[3:2] != 0b00 |
|---|---|
| 00 | Reserved, CONSTRAINED UNPREDICTABLE, See *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427 for the permitted behavior. |
| 01 | Inner Non-cacheable |
| 10 | Inner Write-Through Cacheable |
| 11 | Inner Write-Back Cacheable |

**Note**

The stage 2 translation does not assign any allocation hints.

The following sections describe how the memory type attributes assigned at stage 2 of the translation are combined with those assigned at stage 1:

- *Combining the memory type attribute*.
- *Combining the Cacheability attribute* on page G4-4056.
- *Combining the Shareability attribute* on page G4-4057.

**Note**

The following stage 2 translation table attribute settings leave the stage 1 settings unchanged:

- MemAttr[3:2] == 0b11, Normal memory, Outer Write-Back Cacheable.
- MemAttr[1:0] == 0b11, Inner Write-Back Cacheable.

### Combining the memory type attribute

Table G4-22 shows how the stage 1 and stage 2 memory type assignments are combined:

**Table G4-22 Combining the stage 1 and stage 2 memory type assignments**

| Assignment in stage 1 | Assignment in stage 2 | Resultant type |
|---|---|---|
| Device-nGnRnE | Any | Device-nGnRnE |
| Device-nGnRE | Device-nGnRnE | Device-nGnRnE |
| | Not Device-nGnRnE | Device-nGnRE |
| Device-nGRE | Device-nGnRnE | Device-nGnRnE |
| | Device-nGnRE | Device-nGnRE |
| | Not (Device-nGnRnE or Device-nGnRE) | Device-nGRE |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-22 Combining the stage 1 and stage 2 memory type assignments (continued)**

| Assignment in stage 1 | Assignment in stage 2 | Resultant type |
|---|---|---|
| Device-GRE | Device-nGnRnE | Device-nGnRnE |
| | Device-nGnRE | Device-nGnRE |
| | Device-nGRE | Device-nGRE |
| | Device-GRE or Normal | Device-GRE |
| Normal | Any type of Device | Device type assigned at stage 2 |
| | Normal | Normal |

See *Combining the Shareability attribute* on page G4-4057 for information about the Shareability of:

- A region for which the resultant type is any Device type.
- A region with a resultant type of Normal for which the resultant cacheability, described in *Combining the Cacheability attribute*, is Inner Non-cacheable, Outer Non-cacheable.

The combining of the memory type attribute means a translation table walk for a stage 1 translation can be made to a type of Device memory. If this occurs, then:

- If the value of HCR.PTW is 0, then the translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be done speculatively.
- If the value of HCR.PTW is 1, then the memory access generates a stage 2 Permission fault.

### Combining the Cacheability attribute

For a Normal memory region, Table G4-23 shows how the stage 1 and stage 2 Cacheability assignments are combined. This combination applies, independently, for the Inner Cacheability and Outer Cacheability attributes:

**Table G4-23 Combining the stage 1 and stage 2 cacheability assignments**

| Assignment in stage 1 | Assignment in stage 2 | Resultant cacheability |
|---|---|---|
| Non-cacheable | Any | Non-cacheable |
| Any | Non-cacheable | Non-cacheable |
| Write-Through Cacheable | Write-Through or Write-Back Cacheable | Write-Through Cacheable |
| Write-Through or Write-Back Cacheable | Write-Through Cacheable | Write-Through Cacheable |
| Write-Back Cacheable | Write-Back Cacheable | Write-Back Cacheable |

**Note**

Only Normal memory has a Cacheability attribute.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Combining the Shareability attribute

In the following cases, a memory region is treated as Outer Shareable, regardless of any shareability assignments at either stage of translation:

- The resultant memory type attribute, described in *Combining the memory type attribute* on page G4-4055, is any type of Device memory.
- The resultant memory type attribute is Normal memory, and the resultant Cacheability, described in *Combining the Cacheability attribute* on page G4-4056, is Inner Non-cacheable Outer Non-cacheable.

For a memory region with a resultant memory type attribute of Normal that is not Inner Non-cacheable Outer Non-cacheable, Table G4-24 shows how the stage 1 and stage 2 shareability assignments are combined:

**Table G4-24 Combining the stage 1 and stage 2 Shareability assignments**

| Assignment in stage 1 | Assignment in stage 2 | Resultant Shareability |
|---|---|---|
| Outer Shareable | Any | Outer Shareable |
| Inner Shareable | Outer Shareable | Outer Shareable |
| Inner Shareable | Inner Shareable | Inner Shareable |
| Inner Shareable | Non-shareable | Inner Shareable |
| Non-shareable | Outer Shareable | Outer Shareable |
| Non-shareable | Inner Shareable | Inner Shareable |
| Non-shareable | Non-shareable | Non-shareable |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.9 Translation Lookaside Buffers (TLBs)

*Translation Lookaside Buffers* (TLBs) are an implementation technique that caches translations or translation table entries. TLBs avoid the requirement to perform a translation table walk in memory for every memory access. The ARM architecture does not specify the exact form of the TLB structures for any design. In a similar way to the requirements for caches, the architecture only defines certain principles for TLBs:

- The architecture has a concept of an entry locked down in the TLB. The method by which lockdown is achieved is IMPLEMENTATION DEFINED, and an implementation might not support lockdown.
- The architecture does not guarantee that an unlocked TLB entry remains in the TLB.
- The architecture guarantees that a locked TLB entry remains in the TLB. However, a locked TLB entry might be updated by subsequent updates to the translation tables. Therefore, when a change is made to the translation tables, the architecture does not guarantee that a locked TLB entry remains incoherent with an entry in the translation table.
- The architecture guarantees that a translation table entry that generates a Translation fault, an Address size fault, or an Access flag fault is not held in the TLB. However a translation table entry that generates a Domain fault or a Permission fault might be held in the TLB.
- Any translation table entry that does not generate a Translation fault, an Address size fault, or an Access flag fault and is not from a translation regime for an Exception level that is lower than the current Exception level might be allocated to an enabled TLB at any time. The only translation table entries guaranteed not to be held in the TLB are those that generate a Translation fault, an Address size fault, or an Access flag fault.

  **Note**

  An enabled TLB can hold translation table entries that do not generate a Translation fault but point to subsequent tables in the translation table walk. This can be referred to as *intermediate caching* of TLB entries.

- Software can rely on the fact that between disabling and re-enabling a stage of address translation, entries in the TLB relating to that stage of translation have not been corrupted to give incorrect translations.

The following sections give more information about TLB implementation:

- *Global and process-specific translation table entries*.
- *TLB matching* on page G4-4059.
- *TLB behavior at reset* on page G4-4059.
- *TLB lockdown* on page G4-4059.
- *TLB conflict aborts* on page G4-4060.

See also *TLB maintenance requirements* on page G4-4061.

### G4.9.1 Global and process-specific translation table entries

For VMSAv8-32, system software can divide a virtual memory map used by memory accesses at PL1 and PL0 into global and non-global regions, indicated by the nG bit in the translation table descriptors:

**nG == 0** The translation is global, meaning the region is available for all processes.

**nG == 1** The translation is non-global, or process-specific, meaning it relates to the current ASID, as defined by the CONTEXTIDR.

Each non-global region has an associated *Address Space Identifier* (ASID). These identifiers mean different translation table mappings can co-exist in a caching structure such as a TLB. This means that software can create a new mapping of a non-global memory region without removing previous mappings.

For a symmetric multiprocessor cluster where a single operating system is running on the set of PEs, the architecture requires all ASID values to be assigned uniquely within any single Inner Shareable domain. In other words, each ASID value must have the same meaning to all PEs in the system.

The translation regime used for accesses made at PL2 does not support ASIDs, and all pages are treated as global.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When a PE is using the Long-descriptor translation table format, and is in Secure state, a translation must be treated as non-global, regardless of the value of the nG bit, if NSTable is set to 1 at any level of the translation table walk.

For more information see *Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format* on page G4-4025.

### G4.9.2 TLB matching

A TLB is a hardware caching structure for translation table information. Like other hardware caching structures, it is mostly invisible to software. However, there are some situations where it can become visible. These are associated with coherency problems caused by an update to the translation table that has not been reflected in the TLB. Use of the TLB maintenance instructions described in *TLB maintenance requirements* on page G4-4061 can prevent any TLB incoherency becoming a problem.

A particular case where the presence of the TLB can become visible is if the translation table entries that are in use under a particular ASID and VMID are changed without suitable invalidation of the TLB. This is an issue regardless of whether or not the translation table entries are global. In some cases, the TLB can hold two mappings for the same address, and this might lead to CONSTRAINED UNPREDICTABLE behavior, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5427.

### G4.9.3 TLB behavior at reset

The ARM architecture does not require a reset to invalidate the TLBs, and recognizes that an implementation might require caches, including TLBs, to maintain context over a system reset. Possible reasons for doing so include power management and debug requirements.

Therefore, for ARMv8:

- All TLBs reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
- All TLBs are disabled from reset. All stages of address translation that are used from the PE state entered on coming out of reset are disabled from reset, and the contents of the TLBs have no effect on address translation. For more information see *Enabling stages of address translation* on page G4-4002.
- An implementation can require the use of a specific TLB invalidation routine, to invalidate the TLB arrays before they are enabled after a reset. The exact form of this routine is IMPLEMENTATION DEFINED, but if an invalidation routine is required it must be documented clearly as part of the documentation of the device.

  ARM recommends that if an invalidation routine is required for this purpose, and the PE resets into AArch32 state, the routine is based on the AArch32 TLB maintenance instructions described in *The scope of TLB maintenance instructions* on page G4-4069.

Similar rules apply:

- To cache behavior, see *Behavior of caches at reset* on page G3-3955.
- To branch predictor behavior, see *Behavior of the branch predictors at reset* on page G3-3965.

### G4.9.4 TLB lockdown

The ARM architecture recognizes that any TLB lockdown scheme is heavily dependent on the microarchitecture, making it inappropriate to define a common mechanism across all implementations. This means that:

- The architecture does not require TLB lockdown support.
- If TLB lockdown support is implemented, the lockdown mechanism is IMPLEMENTATION DEFINED. However, key properties of the interaction of lockdown with the architecture must be documented as part of the implementation documentation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This means that:

- The TLB Type Register, TLBTR, does not define the lockdown scheme in use.
- In AArch32 state, a region of the CP15 c10 encodings is reserved for IMPLEMENTATION DEFINED TLB functions, such as TLB lockdown functions. The reserved encodings are those with:
  - `<CRm>` == {c0, c1, c4, c8}.
  - All values of `<opc2>` and `<opc1>`.

  See also *VMSAv8-32 CP15 c10 register summary* on page G4-4145.

An implementation might use some of the CP15 c10 encodings that are reserved for IMPLEMENTATION DEFINED TLB functions to implement additional TLB control functions. These functions might include:

- Unlock all locked TLB entries.
- Preload into a specific level of TLB. This is beyond the scope of the `PLI` and `PLD` hint instructions.

The inclusion of EL2 in an implementation does not affect the TLB lockdown requirements. However, in an implementation that includes EL2, exceptions generated by problems related to TLB lockdown, in a Non-secure PL1 mode, can be routed to either:

- Non-secure Abort mode, using the Non-secure Data Abort exception vector.
- Hyp mode, using the Hyp Trap exception vector.

For more information, see *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page G1-3860.

### G4.9.5 TLB conflict aborts

If an address matches multiple entries in the TLB, it is IMPLEMENTATION DEFINED whether a TLB conflict abort is generated.

An implementation can generate TLB conflict aborts on either or both instruction fetches and data accesses. A TLB conflict abort is classified as an MMU fault, see *MMU faults in AArch32 state* on page G4-4086. This means:

- A TLB conflict abort on an instruction fetch is reported as a Prefetch Abort exception,
- A TLB conflict abort on a data access is reported as a Data Abort exception,

Fault status encodings for TLB conflict aborts are defined for both the Short-descriptor and Long-descriptor translation table formats, see:

- *PL1 fault reporting with the Short-descriptor translation table format* on page G4-4096
- *PL1 fault reporting with the Long-descriptor translation table format* on page G4-4097.

On a TLB conflict abort, the fault address register returns the address that generated the fault. That is, it returns the address that was being looked up in the TLB.

It is IMPLEMENTATION DEFINED whether a TLB conflict abort is a stage 1 abort or a stage 2 abort.

**Note**

- An address can hit multiple entries in the TLB if the TLB has been invalidated inappropriately, for example if TLB invalidation required by this manual has not been performed.
- A stage 2 abort cannot be generated if the Non-secure PL1&0 stage 2 address translation is disabled.

The priority of the TLB conflict abort is IMPLEMENTATION DEFINED, because it depends on the form of any TLB that can generate the abort. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.

If an address matches multiple entries in the TLB and no TLB conflict abort not generated, the resulting behavior is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5427. The CONSTRAINED UNPREDICTABLE behavior must not permit access to regions of memory with permissions or attributes that mean they cannot be accessed in the current Security state at the current Privilege level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.10 TLB maintenance requirements

*Translation Lookaside Buffers (TLBs)* on page G4-4058 describes the ARM architectural provision for TLBs. Although the ARM architecture does not specify the form of any TLB structures, it does define the mechanisms by which TLBs can be maintained. The following sections describe the VMSAv8-32 TLB maintenance instructions:

- *General TLB maintenance requirements*.
- *Maintenance requirements on changing System register values* on page G4-4065.
- *Atomicity of register changes on changing virtual machine* on page G4-4066.
- *Synchronization of changes of ASID and TTBR* on page G4-4067.
- *The scope of TLB maintenance instructions* on page G4-4069.

### G4.10.1 General TLB maintenance requirements

TLB maintenance instructions provide a mechanism to invalidate entries from a TLB. As stated at the start of *Translation Lookaside Buffers (TLBs)* on page G4-4058, any translation table entry that does not generate a Translation fault, an Address size fault, or an Access flag fault might be allocated to an enabled TLB at any time. This means that software must perform TLB maintenance between updating translation table entries that apply in a particular context and accessing memory locations whose translation is determined by those entries in that context.

**Note**

This requirement applies to any translation table entry at any level of the translation tables, including an entry that points to further levels of the tables, provided that the entry in that level of the tables does not cause a Translation fault, an Address size fault, or an Access flag fault.

In addition to any TLB maintenance requirement, when changing the cacheability attributes of an area of memory, software must ensure that any cached copies of affected locations are removed from the caches. For more information see *Cache maintenance requirement created by changing translation table attributes* on page G4-4076.

Because a TLB never holds any translation table entry that generates a Translation fault, an Address size fault, or an Access flag fault, a change from a translation table entry that causes a Translation, Address size, or Access flag fault to one that does not fault, does not require any TLB or branch predictor invalidation.

Special considerations can apply to translation table updates that change the memory type, cacheability, or output address of an entry, see *Using break-before-make when updating translation table entries* on page G4-4062.

In addition, software must perform TLB maintenance after updating the System registers if the update means that the TLB might hold information that applies to a current translation context, but is no longer valid for that context. *Maintenance requirements on changing System register values* on page G4-4065 gives more information about this maintenance requirement.

Each of the translation regimes defined in Figure G4-1 on page G4-3994 is a different context, and:

- For the Non-secure PL1&0 regime, a change in the VMID or ASID value changes the context.
- For the Secure PL1&0 regime, a change in the ASID value changes the context.

For operation in Non-secure PL1 or PL0 modes, a change of HCR.VM, unless made at the same time as a change of VMID, requires the invalidation of all TLB entries for the Non-secure PL1&0 translation regime that apply to the current VMID. Otherwise, there is no guarantee that the effect of the change of HCR.VM is visible to software executing in the Non-secure PL1 and PL0 modes.

Any TLB maintenance instruction can affect any other TLB entries that are not locked down.

AArch32 state defines CP15 c8 functions for TLB maintenance instructions, and supports the following operations:

- Invalidate all unlocked entries in the TLB.
- Invalidate a single TLB entry, by VA, or VA and ASID for a non-global entry.
- Invalidate all TLB entries that match a specified ASID.
- Invalidate all TLB entries that match a specified VA, regardless of the ASID.
- Operations that apply across multiprocessors in the same Inner Shareable domain.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

An address-based TLB maintenance instruction that applies to the Inner Shareable domain does so regardless of the Shareability attributes of the address supplied as an argument to the instruction.

A TLB maintenance instruction that specifies a virtual address that would generate any MMU fault, including a virtual address that is not in the range of virtual addresses that can be translated, does not generate an abort.

EL2 provides additional TLB maintenance instructions for use in AArch32 state at PL2, and has some implications for the effect of the other TLB maintenance instructions, see *The scope of TLB maintenance instructions* on page G4-4069.

In an implementation that includes EL3, the TLB maintenance instructions take account of the current Security state, as part of the address translation required for the TLB maintenance instruction.

Some TLB maintenance instructions are defined as operating only on instruction TLBs, or only on data TLBs. ARMv8 AArch32 state includes these instructions for backwards compatibility. However, more recent TLB maintenance instructions do not support this distinction. From the introduction of ARMv7, ARM deprecates any use of Instruction TLB maintenance instructions, or of Data TLB maintenance instructions, and developers must not rely on this distinction being maintained in future revisions of the ARM architecture.

The ARM architecture does not dictate the form in which the TLB stores translation table entries. However, for TLB invalidate instructions, the minimum size of the table entry that is invalidated from the TLB must be at least the size that appears in the translation table entry.

*The scope of TLB maintenance instructions* on page G4-4069 describes the TLB maintenance instructions. The following subsections give more information about the general requirements for TLB maintenance:

- *Using break-before-make when updating translation table entries*
- *The interaction of TLB lockdown with TLB maintenance instructions* on page G4-4063
- *Ordering and completion of TLB maintenance instructions* on page G4-4064

## Using break-before-make when updating translation table entries

To avoid the effects of TLB caching possibly breaking coherency, ordering guarantees or uniprocessor semantics, or possibly failing to clear the exclusive monitors, ARM strongly recommends the use of a break-before-make when changing translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation entries involves any of:

- A change of the memory type.
- A change of the cacheability attributes.
- A change of the output address (OA), if the OA of at least one of the old translation table entries and the new translation table entry is writable.

A break-before-make approach on changing from an old translation table entry to a new translation table entry requires the following steps:

1. Replace the old translation table entry with an invalid entry, and execute a `DSB` instruction.
2. Invalidate the translation table entry with a broadcast TLB invalidation instruction, and execute a `DSB` instruction to ensure the completion of that invalidation.
3. Write the new translation table entry, and execute a `DSB` instruction to ensure that the new entry is visible.

This sequence ensures that at no time are both the old and new entries simultaneously visible to different threads of execution. This means the problems described at the start of this subsection cannot arise.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### The interaction of TLB lockdown with TLB maintenance instructions

The precise interaction of TLB lockdown with the TLB maintenance instructions is IMPLEMENTATION DEFINED. However, the architecturally-defined TLB maintenance instructions must comply with these rules:

- The effect on locked entries of a TLB invalidate all unlocked entries instruction or a TLB invalidate by VA all ASID instruction is IMPLEMENTATION DEFINED. However, these instructions must implement one of the following options:
  - Have no effect on entries that are locked down.
  - Generate an IMPLEMENTATION DEFINED Data Abort exception if an entry is locked down, or might be locked down. For an invalidate instruction performed in AArch32 state, the CP15 c5 fault status register definitions include a fault code for cache and TLB lockdown faults, see Table G4-27 on page G4-4096 for the codes used with the Short-descriptor translation table formats, or Table G4-28 on page G4-4097 for the codes used with the Long-descriptor translation table formats.

    In an implementation that includes EL2, if EL2 is using AArch32 and the value of HCR.TIDCP is 1, any such exceptions taken from a Non-secure PL1 mode are routed to Hyp mode, see *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page G1-3860.

  This permits a usage model for TLB invalidate routines, where the routine invalidates a large range of addresses, without considering whether any entries are locked in the TLB.
- The effect on locked entries of a TLB invalidate by VA instruction or a TLB invalidate by ASID match instruction is IMPLEMENTATION DEFINED. However, these instructions must implement one of the following options:
  - A locked entry is invalidated in the TLB.
  - The instruction has no effect on a locked entry in the TLB. In the case of the Invalidate single entry by VA, this means the PE treats the instruction as a NOP.
  - The instruction generates an IMPLEMENTATION DEFINED Data Abort exception if it operates on an entry that is locked down, or might be locked down. For an invalidate instruction performed in AArch32 state, the CP15 c5 fault status register definitions include a fault code for cache and TLB lockdown faults, see Table G4-27 on page G4-4096 and Table G4-28 on page G4-4097.

**Note**

Any implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required invalidation on entries that are not locked down.
- Implement one of the other specified alternatives for the locked entries.

ARM recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use the architecturally-defined maintenance instructions. This minimizes the number of customized maintenance operations required.

In addition, an implementation that uses an abort mechanism for handling TLB maintenance instructions on entries that can be locked down but are not actually locked down must also must provide a mechanism that ensures that no TLB entries are locked.

Similar rules apply to cache lockdown, see *The interaction of cache lockdown with cache maintenance instructions* on page G3-3972.

The architecture does not guarantee that any unlocked entry in the TLB remains in the TLB. This means that, as a side-effect of a TLB maintenance instruction, any unlocked entry in the TLB might be invalidated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Ordering and completion of TLB maintenance instructions

The following rules describe the relations between the memory order model and the TLB maintenance instructions:

- A TLB invalidate instruction is complete when all memory accesses using the invalidated TLB entries have been observed by all observers, to the extent that those accesses must be observed. The Shareability and Cacheability of the accessed memory locations determine the extent to which the accesses must be observed.

  **Note**

  For a TLB invalidate instruction that affects other PEs, the set of memory accesses that have been observed when the TLB maintenance instruction is complete must include the memory accesses from those processes that used the invalidated TLB entries.

  After the TLB invalidate instruction is complete, no new memory accesses using the invalidated TLB entries will be observed by those observers.

  **Note**

  This requirement does not mean that speculative memory accesses cannot be performed using those entries if it is impossible to tell that those memory accesses have been observed by the observers.

- A TLB maintenance instruction is only guaranteed to be complete after the execution of a `DSB` instruction.
- An `ISB` instruction, or a return from an exception, causes the effect of all completed TLB maintenance instructions that appear in program order before the `ISB` or return from exception to be visible to all subsequent instructions, including the instruction fetches for those instructions.
- An exception causes all completed TLB maintenance instructions, that appear in the instruction stream before the point where the exception is taken, to be visible to all subsequent instructions, including the instruction fetches for those instructions.
- All TLB maintenance instructions are executed in program order relative to each other.
- The execution of a Data or Unified TLB maintenance instruction is only guaranteed to be visible to a subsequent explicit load or store instruction after both:
  - The execution of a `DSB` instruction to ensure the completion of the TLB maintenance instruction.
  - Execution of a subsequent *Context synchronization operation*.
- The execution of an Instruction or Unified TLB maintenance instruction is only guaranteed to be visible to a subsequent instruction fetch after both:
  - The execution of a `DSB` instruction to ensure the completion of the TLB maintenance instruction.
  - Execution of a subsequent *Context synchronization operation*.

In all cases in this section, where a `DMB` or `DSB` is referred to, it refers to a `DMB` or `DSB` whose required access type is both loads and stores.

The following rules apply when writing translation table entries. They ensure that the updated entries are visible to subsequent accesses and cache maintenance instructions.

For TLB maintenance, the translation table walk is treated as a separate observer. This means:

- A write to the translation tables is only guaranteed to be seen by a translation table walk caused by an explicit load or store after the execution of both a `DSB` and an `ISB`.

  However, the architecture guarantees that any writes to the translation tables are not seen by any explicit memory access that occurs in program order before the write to the translation tables.

- A write to the translation tables is only guaranteed to be seen by a translation table walk caused by the instruction fetch of an instruction that follows the write to the translation tables after both a `DSB` and an `ISB`.

Therefore, in a uniprocessor system, an example instruction sequence for writing a translation table entry, covering changes to the instruction or data mappings is:

```
STR rx, [Translation table entry]          ; write new entry to the translation table
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
DSB            ; ensures visibility of the new entry
Invalidate TLB entry by VA (and ASID if non-global) [page address]
Invalidate BTC
DSB            ; ensure completion of the Invalidate TLB instruction
ISB            ; ensure table changes visible to instruction fetch
```

### G4.10.2 Maintenance requirements on changing System register values

The TLB contents can be influenced by control bits in a number of System registers. This means the TLB must be invalidated after any changes to these bits, unless the changes are accompanied by a change to the VMID or ASID that defines the context to which the bits apply. The general form of the required invalidation sequence is as follows:

```
; Change control bits in System registers
ISB        ; Synchronize changes to the control bits
; Perform TLB invalidation of all entries that might be affected by the changed control bits
```

The System register changes that this applies to are:

- Any change to the NMRR, PRRR, MAIR0,MAIR1, HMAIR0 or HMAIR1 registers.
- Any change to the SCTLR.AFE bit, see *Changing the Access flag enable*.
- Any change to any of the SCTLR.{TRE, WXN, UWXN} bits.
- Any change to the Translation table base 0 address in TTBR0.
- Any change to the Translation table base 1 address in TTBR1.
- Any change to HTTBR.BADDR.
- Any change to VTTBR.BADDR.
- Changing TTBCR.EAE, see *Changing the current Translation table format* on page G4-4066.
- In an implementation that includes EL3, any change to the SCR.SIF bit.
- In an implementation that includes EL2:
  - Any change to the HCR.VM bit.
  - Any change to HCR.PTW bit, see *Changing HCR.PTW* on page G4-4066.
- When using the Short-descriptor translation table format:
  - Any change to the RGN, IRGN, S, or NOS fields in TTBR0 or TTBR1.
  - Any change to the N, EAE, PD0 or PD1 fields in TTBCR
- When using the Long-descriptor translation table format:
  - Any change to the EAE, T*n*SZ, ORGN*n*, IRGN*n*, SH*n*, or EPD*n* fields in the TTBCR, where *n* is 0 or 1.
  - Any change to the T0SZ, ORGN0, IRGN0, or SH0 fields in the HTCR.
  - Any change to the T0SZ, ORGN0, IRGN0, or SH0 fields in the VTCR.

#### Changing the Access flag enable

In a PE that is using the Short-descriptor translation table format, it is UNPREDICTABLE whether the TLB caches the effect of the SCTLR.AFE bit on translation tables. This means that, after changing the SCTLR.AFE bit software must invalidate the TLB before it relies on the effect of the new value of the SCTLR.AFE bit, otherwise behavior is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5427.

**Note**

There is no enable bit for use of the Access flag when using the Long-descriptor translation table format.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Changing HCR.PTW

When EL2 is using AArch32 and the value of the Protected table walk bit, HCR.PTW, is 1, a stage 1 translation table access in the Non-secure PL1&0 translation regime, to an address that is mapped to any type of Device memory by its stage 2 translation, generates a stage 2 Permission fault. A TLB associated with a particular VMID might hold entries that depend on the effect of HCR.PTW. Therefore, if the value of HCR.PTW is changed without a change to the VMID value, all TLB entries associated with the current VMID must be invalidated before executing software in a Non-secure PL1 or PL0 mode. If this is not done, behavior is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5427.

### Changing the current Translation table format

The effect of changing TTBCR.EAE when executing in the translation regime affected by TTBCR.EAE with any stage of address translation for that translation regime enabled is CONSTRAINED UNPREDICTABLE. This means that, when TTBCR.EAE is changed for a given context, the TLB must be invalidated before resuming execution in that context, otherwise the effect is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5427.

## G4.10.3 Atomicity of register changes on changing virtual machine

From the viewpoint of software executing in a Non-secure PL1 or PL0 mode, when there is a switch from one virtual machine to another, the registers that control or affect address translation must be changed atomically. This applies to the registers for:

- Non-secure PL1&0 stage 1 address translations. This means that all of the following registers must change atomically:
  - PRRR and NMRR, if using the Short-descriptor translation table format.
  - MAIR0 and MAIR1, if using the Long-descriptor translation table format.
  - TTBR0, TTBR1, TTBCR, DACR, and CONTEXTIDR.
  - The SCTLR.
- Non-secure PL1&0 stage 2 address translations. When EL2 is using AArch32, this means that all of the following registers and register fields must change atomically:
  - VTTBR and VTCR.
  - HMAIR0 and HMAIR1.
  - The HSCTLR.

**Note**

Only some bits of SCTLR affect the stage 1 translation, and only some bits of HSCTLR affect the stage 2 translation. However, in each case, changing these bits requires a write to the register, and that write must be atomic with the other register updates.

These registers apply to execution in Non-secure PL1&0 modes. However, when updated as part of a switch of virtual machines they are updated by software executing in Hyp mode. This means the registers are *out of context* when they are updated, and no synchronization precautions are required.

**Note**

By contrast, a translation table change associated with a change of ASID, made by software executing at PL1, can require changes to registers that are *in context*. *Synchronization of changes of ASID and TTBR* on page G4-4067 describes appropriate precautions for such a change.

Software executing in Hyp mode, or in Secure state, must not use the registers associated with the Non-secure PL1&0 translation regime for speculative memory accesses.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.10.4 Synchronization of changes of ASID and TTBR

A common virtual memory management requirement is to change the ASID and TTBR together to associate the new ASID with different translation tables, without any change to the current translation regime. When using the Short-descriptor translation table format, different registers hold the ASID and the translation table base address, meaning these two values cannot be updated atomically. Since a PE can perform a speculative memory access at any time, this lack of atomicity is a problem that software must address. Such a change is complicated by:

- The depth of speculative fetch being IMPLEMENTATION DEFINED.
- The use of branch prediction.

When using the Short-descriptor translation table format, the virtual memory management operations must ensure the synchronization of changes of the ContextID and the translation table registers. For example, some or all of the TLBs, branch predictors, and other caching of ASID and translation information might become corrupt with invalid translations. Synchronization is required to avoid either:

- The old ASID being associated with translation table walks from the new translation tables.
- The new ASID being associated with translation table walks from the old translation tables.

There are a number of possible solutions to this problem, and the most appropriate approach depends on the system. Example G4-3, Example G4-4, and Example G4-5 on page G4-4068 describe three possible approaches.

**Note**

Another instance of the synchronization problem occurs if a branch is encountered between changing the ASID and performing the synchronization. In this case the value in the branch predictor might be associated with the incorrect ASID. Software can address this possibility using any of these approaches, but instead software might be written in a way that avoids such branches.

**Example G4-3 Using a reserved ASID to synchronize ASID and TTBR changes**

In this approach, a particular ASID value is reserved for use by the operating system, and is used only for the synchronization of the ASID and TTBR. This example uses the value of 0 for this purpose, but any value could be used.

This approach can be used only when the size of the mapping for any given virtual address is the same in the old and new translation tables.

The maintenance software uses the following sequence, that must be executed from memory marked as global:

```
Change ASID to 0
ISB
Change TTBR
ISB
Change ASID to new value
```

This approach ensures that any non-global pages fetched at a time when it is uncertain whether the old or new translation tables are being accessed are associated with the unused ASID value of 0. Since the ASID value of 0 is not used for any normal operations these entries cannot cause corruption of execution.

**Example G4-4 Using translation tables containing only global mappings when changing the ASID**

A second approach involves switching the translation tables to a set of translation tables that only contain global mappings while switching the ASID.

The maintenance software uses the following sequence, that must be executed from memory marked as global:

```
Change TTBR to the global-only mappings
ISB
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
Change ASID to new value
ISB
Change TTBR to new value
```

This approach ensures that no non-global pages can be fetched at a time when it is uncertain whether the old or new ASID value will be used.

This approach works without the need for TLB invalidations in systems that have caching of intermediate levels of translation tables, as described in *General TLB maintenance requirements* on page G4-4061, provided that the translation tables containing only global mappings have only level 1 translation table entries of the following kinds:

- Entries that are global.
- Pointers to level 2 tables that hold only global entries, and that are the same level 2 tables that are used for accessing global entries by both:
  - The set of translation tables that were used under the old ASID value.
  - The set of translation tables that will be used with the new ASID value.
- Invalid level 1 entries.

In addition, all sets of translation tables in this example should have the same Shareability and Cacheability attributes, as held in the TTBR0.{ORGN, IRGN} or TTBR1.{ORGN, IRGN} fields.

If these rules are not followed, then the implementation might cache level 1 translation table entries that require explicit invalidation.

---

**Example G4-5 Disabling non-global mappings when changing the ASID**

---

In systems where only the translation tables indexed by TTBR0 hold non-global mappings, maintenance software can use the TTBCR.PD0 field to disable use of TTBR0 during the change of ASID. This means the system does not require a set of global-only mappings.

The maintenance software uses the following sequence, that must be executed from a memory region with a translation that is accessed using the base address in the TTBR1 register, and is marked as global:

```
Set TTBCR.PD0 = 1
ISB
Change ASID to new value
Change TTBR to new value
ISB
Set TTBCR.PD0 = 0
```

This approach ensures that no non-global pages can be fetched at a time when it is uncertain whether the old or new ASID value will be used.

---

When using the Long-descriptor translation table format, TTBCR.A1 holds the number, 0 or 1, of the TTBR that holds the current ASID. This means the current TTBR can also hold the current ASID, and the current translation table base address and ASID can be updated atomically when:

- TTBR0 is the only TTBR being used. TTBCR.A1 must be set to 0.
- TTBR0 points to the only translation tables that hold non-global entries, and TTBCR.A1 is set to 0.
- TTBR1 points to the only translation tables that hold non-global entries, and TTBCR.A1 is set to 1.

In these cases, software can update the current translation table base address and ASID atomically, by updating the appropriate TTBR, and does not require a specific routine to ensure synchronization of the change of ASID and base address.

However, in all other cases using the Long-descriptor format, the synchronization requirements are identical to those when using the Short-descriptor formats, and the examples in this section indicate how synchronization might be achieved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

When using the Long-descriptor translation table format, CONTEXTIDR.ASID has no significance for address translation, and is only an extension of the Context ID value.

### G4.10.5 The scope of TLB maintenance instructions

TLB maintenance instructions provide a mechanism for invalidating entries from TLB caching structures, to ensure that changes to the translation tables are reflected correctly in the TLB caching structures. To support TLB maintenance in multiprocessor systems, there are maintenance operations that apply to the TLBs of all PEs in the same Inner Shareable domain.

The architecture permits the caching of any translation table entry that has been returned from memory without a fault and that does not, itself, cause a Translation Fault, an Address size fault, or an Access Flag fault. This means the TLB:

- Cannot hold an entry that, when used for a translation table lookup, causes a Translation fault, an Address size fault, or an Access Flag fault.
- Can hold an entry for a translation table lookup for a translation that causes a Translation Fault, an Address size fault, or an Access Flag fault at a subsequent level of translation table lookup. For example, it can hold an entry for the level 1 lookup of a translation that causes a Translation fault, an Address size fault, or an Access Flag fault at level 2 or level 3 of lookup.

This means that entries cached in the TLB can include:

- Translation table entries that point to a subsequent table to be used in the current stage of translation.
- In an implementation that includes EL2:
  - Stage 2 translation table entries that are used as part of a stage 1 translation table walk.
  - Stage 2 translation table entries for translating the output address of a stage 1 translation.

Such entries might be held in intermediate TLB caching structures that are distinct from the data caches, in that they are not required to be invalidated as the result of writes of the data. The architecture makes no restriction on the form of these intermediate TLB caching structures.

The architecture does not intend to restrict the form of TLB caching structures used for holding translation table entries. In particular for translation regimes that involve two stages of translation, it recognizes that such caching structures might contain:

- At any level of the translation table walk, entries containing information from stage 1 translation table entries.
- In an implementation that includes EL2:
  - At any level of the translation table walk, entries containing information from stage 2 translation table entries.
  - At any level of the translation table walk, entries combining information from both stage 1 and stage 2 translation table entries.

**Note**

For the purpose of TLB maintenance, the term *TLB entry* denotes any structure, including temporary working registers in translation table walk hardware, that holds a translation table entry.

Where a TLB maintenance instruction is:

- Required to apply to stage 1 entries, then it must apply to any cached entry in the caching structures that includes any stage 1 information that would be used to translate the address being invalidated, including any entry that combines information from both stage 1 and stage 2 translation table entries.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

— Where stage 1 information has been cached in multiple TLB entries, as could occur from splintering a page when caching in the TLB, then the invalidation must apply to each cached entry containing stage 1 information from the page that is used to translate the address being invalidated, regardless of whether or not that cached entry would be used to translate the address being invalidated.

— As stated in *Global and process-specific translation table entries* on page G4-4058, translation table entries from levels of translation other than the final level are treated as being non-global. ARM expects that, in at least some implementations, cached copies of levels of the translation table walk other than the last level are tagged with their ASID, regardless of whether the final level is global. This means that TLB invalidations that involve the ASID require the ASID to match such entries to perform the required invalidation.

- Required to apply to stage 2 entries only, then:
  — It is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.
  — It must apply to caching structures that contain information only from stage 2 translation table entries.
- Required to apply to both stage 1 and stage 2 entries, then it must apply to any entry in the caching structures that includes information from either a stage 1 translation table entry or a stage 2 translation table entry, including any entry that combines information from both stage 1 and stage 2 translation table entries.

Table G4-25 on page G4-4071 summarizes the required effect of the preferred TLB operations, for execution in AArch32 state, that operate only on TLBs on the PE that executes the instruction. Additional TLB operations:

- Apply across all PEs in the same Inner Shareable domain. Each operation shown in the table has an Inner Shareable equivalent, identified by an IS suffix. For example, the Inner Shareable equivalent of TLBIALL is TLBIALLIS. See also *EL2 upgrading of TLB maintenance instructions* on page G4-4072.
- Can apply to separate Instruction or Data TLBs, as indicated by a footnote to the table. ARM deprecates any use of these operations.

**Note**

- The architecture permits a TLB invalidation operation to affect any unlocked entry in the TLB. Table G4-25 on page G4-4071 defines only the entries that each operation must invalidate.
- All TLB operations, including those that operate on a VA match, operate regardless of the value of SCTLR.M.

When interpreting the table:

**Related operations** Each operation description applies also to any equivalent operation that either:
- Applies to all PEs in the same Inner Shareable domain.
- Applies only to a data TLB, or only to an instruction TLB.

So, for example, the TLBIALL description applies also to TLBIALLIS, ITLBIALL, and DTLBIALL.

*TLB maintenance instructions, functional group* on page G4-4171 lists all of the TLB maintenance instructions.

**Matches the VA** Means the VA argument for the operation must match the VA value in the TLB entry.

**Matches the ASID** Means the ASID argument for the operation must match the ASID in use when the TLB entry was assigned.

**Matches the current VMID**

Means the current VMID must match the VMID in use when the TLB entry was assigned.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The dependency on the VMID applies even when the value of HCR.VM is 0, including situations where there is no use of virtualization. However, VTTBR.VMID resets to zero, meaning there is a valid VMID from reset.

**Execution at PL2** Descriptions of operations at PL2 apply only to implementations that include EL2.

For the definitions of the translation regimes referred to in the table see *About VMSAv8-32* on page G4-3992.

**Table G4-25 Effect of the TLB maintenance instructions**

| Operation | Executed from | | Effect, must invalidate any entry that matches all stated conditions |
|---|---|---|---|
| | State | Mode | |
| TLBIALL[a] | Secure | PL1 | All entries for the Secure PL1&0 translation regime. That is, all entries that were allocated in Secure state. |
| | Non-secure | PL1 | All entries for stage 1 of the Non-secure PL1&0 translation regime that match the current VMID. |
| | | Hyp | All entries for stage 1 or stage 2 of the Non-secure PL1&0 translation regime that match the current VMID. |
| TLBIMVA[a] | Secure | PL1 | Any entry for the Secure PL1&0 translation regime that both:<br>• Matches the VA argument.<br>• Matches the ASID argument, or is global. |
| | Non-secure | PL1 or Hyp | Any entry for stage 1 of the Non-secure PL1&0 translation regime to which all of the following apply. The entry:<br>• Matches the VA argument.<br>• Matches the ASID argument, or is global.<br>• Matches the current VMID. |
| TLBIASID[a] | Secure | PL1 | Any entry for the Secure PL1&0 translation regime that matches the specified ASID and either:<br>• Is from a level of lookup above the final level.<br>• Is a non-global entry from the final level of lookup. |
| | Non-secure | PL1 or Hyp | Any entry for stage 1 of the Non-secure PL1&0 translation regime that both:<br>• Matches the specified ASID and either:<br>— Is from a level of lookup above the final level.<br>— Is a non-global entry from the final level of lookup.<br>• Matches the current VMID. |
| TLBIMVAA | Secure | PL1 | Any entry for the Secure PL1&0 translation regime that matches the VA argument. |
| | Non-secure | PL1 or Hyp | Any entry for stage 1 of the Non-secure PL1&0 translation regime that both:<br>• Matches the VA argument.<br>• Matches the current VMID. |
| TLBIALLNSNH[b] | Secure | Monitor | All entries for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID. |
| | Non-secure | Hyp | |
| TLBIALLH[b] | Secure | Monitor | All entries for the Non-secure PL2 translation regime. That is, any entry that was allocated in Non-secure state from Hyp mode. |
| | Non-secure | Hyp | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-25 Effect of the TLB maintenance instructions (continued)**

| Operation | Executed from | | Effect, must invalidate any entry that matches all stated conditions |
|---|---|---|---|
| | State | Mode | |
| TLBIMVAL | Secure | PL1 | Any entry for stage 1 of the Secure PL1&0 translation regime that is from the last level of the translation table walk and both:<br>• Matches the VA argument.<br>• Matches the ASID argument, or is global. |
| | Non-secure | PL1 or Hyp | Any entry for stage 1 of the Non-secure PL1&0 translation regime that is from the last level of the translation table walk and to which all of the following apply. The entry:<br>• Matches the VA argument.<br>• Matches the ASID argument, or is global.<br>• Matches the current VMID. |
| TLBIMVAAL | Secure | PL1 | Any entry for stage 1 of the Secure PL1&0 translation regime that is from the last level of the translation table walk and matches the VA argument. |
| | Non-secure | PL1 or Hyp | Any entry for stage 1 of the Non-secure PL1&0 translation regime that is from the last level of the translation table walk and both:<br>• Matches the VA argument.<br>• Matches the current VMID. |
| TLBIMVAH[b] | Secure | Monitor | Any entry for the Non-secure PL2 translation regime that matches the VA argument. |
| | Non-secure | Hyp | |
| TLBIMVALH[b] | Secure | Monitor | Any entry for the Non-secure PL2 translation regime that is from the last level of the translation table walk and matches the VA argument. |
| | Non-secure | Hyp | |
| TLBIIPAS2[b] | Secure | Monitor[c] | Any entry for stage 2 of the PL1&0 translation regime that both:<br>• Matches the IPA argument.<br>• Matches the current VMID. |
| | Non-secure | Hyp[d] | |
| TLBIIPAS2L[b] | Secure | Monitor[c] | Any entry for stage 2 of the PL1&0 translation regime that is from the last level of translation and both:<br>• Matches the IPA argument.<br>• Matches the current VMID. |
| | Non-secure | Hyp[d] | |

a. The architecture defines variants of these operations that apply only to instruction TLBs, and only to data TLBs. ARM deprecates any use of these variants. For more information, see the referenced description of the operation.

b. Available only in an implementation that includes EL2. See also *EL2 upgrading of TLB maintenance instructions*.

c. This instruction executes as a NOP when SCR.NS == 0.

d. This instruction is CONSTRAINED UNPREDICTABLE if executed in any AArch32 Secure privileged mode, see *Hyp mode TLB maintenance instructions* on page K1-5499.

### EL2 upgrading of TLB maintenance instructions

In an implementation that includes EL2, when the value of HCR.FB is 1, the TLB maintenance instructions that are not broadcast across the Inner Shareable domain are upgraded to operate across the Inner Shareable domain when performed in a Non-secure PL1 mode. For example, when the value of HCR.FB is 1, a TLBIMVA operation performed in a Non-secure PL1 mode operates as a TLBIMVAIS operation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### TLB maintenance with different translation granule sizes

If a TLB maintenance instruction specifying a virtual address affecting the PL2 translation regime is broadcast from a PE using AArch32 to a PE using AArch64 using a translation granule size that is different from the AArch32 translation granule size for that same translation regime, the TLB maintenance instruction is not required to perform any invalidation on the recipient PE.

If a TLB maintenance instruction specifying a virtual address affecting the PL1 translation regime is broadcast from a PE using AArch32 using one translation granule size for that translation regime for a particular ASID, VMID (if applicable), and Security state, to a PE using AArch64 where EL1 for the same ASID, VMID (if applicable), and Security state, is using a translation granule size that is different from the AArch32 translation granule size, the TLB maintenance instruction is not required to perform any invalidation on the recipient PE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.11 Caches in VMSAv8-32

The ARM architecture describes the required behavior of an implementation of the architecture. As far as possible it does not restrict the implemented microarchitecture, or the implementation techniques that might achieve the required behavior.

Maintaining this level of abstraction is difficult when describing the relationship between memory address translation and caches, especially regarding the indexing and tagging policy of caches. This section:

- Summarizes the architectural requirements for the interaction between caches and memory translation.
- Gives some information about the likely implementation impact of the required behavior.

The following sections give this information:

- *Data and unified caches*.
- *Instruction caches*.

In addition *Cache maintenance requirement created by changing translation table attributes* on page G4-4076 describes the cache maintenance required after updating the translation tables to change the attributes of an area of memory.

For more information about cache maintenance see:

- *AArch32 cache and branch predictor support* on page G3-3952. This section describes the ARM cache maintenance instructions.
- *Cache maintenance instructions, functional group* on page G4-4169. This section summarizes the System register encodings used for these operations when executing in AArch32 state.

### G4.11.1 Data and unified caches

For data and unified caches, the use of memory address translation is entirely transparent to any data access that is not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE.

This means that the behavior of accesses from the same observer to different VAs, that are translated to the same PA with the same memory attributes, is fully coherent. This means these accesses behave as follows, regardless of which VA is accessed:

- Two writes to the same PA occur in program order.
- A read of a PA returns the value of the last successful write to that PA.
- A write to a PA that occurs, in program order, after a read of that PA, has no effect on the value returned by that read.

The memory system behaves in this way without any requirement to use barrier or cache maintenance instructions.

In addition, if cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

These properties are consistent with implementing all caches that can handle data accesses as *Physically-indexed, physically-tagged* (PIPT) caches.

### G4.11.2 Instruction caches

In the ARM architecture, an instruction cache is a cache that is accessed only as a result of an instruction fetch. Therefore, an instruction cache is never written to by any load or store instruction executed by the PE.

The ARM architecture supports three different behaviors for instruction caches. For ease of reference and description these are identified by descriptions of the associated expected implementation, as follows:

- PIPT instruction caches.
- *Virtually-indexed, physically-tagged* (VIPT) instruction caches.
- ASID and VMID tagged *Virtually-indexed, virtually-tagged* (VIVT) instruction caches.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In AArch32 state, the CTR identifies the form of the instruction caches, see *CTR, Cache Type Register* on page G6-4276.

The following subsections describe the behavior associated with these cache types, including any occasions where explicit cache maintenance is required to make the use of memory address translation transparent to the instruction cache:

- *PIPT instruction caches*.
- *VIPT instruction caches*.
- *ASID and VMID tagged VIVT instruction caches*.
- *The IVIPT architecture Extension* on page G4-4076.

**Note**

For software to be portable between implementations that might use any of PIPT instruction caches, VIPT instruction caches, or ASID and VMID tagged VIVT instruction caches, the software must invalidate the instruction cache whenever any condition occurs that would require instruction cache maintenance for at least one of the instruction cache types.

### PIPT instruction caches

For PIPT instruction caches, the use of memory address translation is entirely transparent to all instruction fetches that are not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE.

If cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

An implementation that provides PIPT instruction caches implements the IVIPT Extension, see *The IVIPT architecture Extension* on page G4-4076.

### VIPT instruction caches

For VIPT instruction caches, the use of memory address translation is transparent to all instruction fetches that are not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, except for the effect of memory address translation on instruction cache invalidate by address operations.

**Note**

Cache invalidation is the only cache maintenance instruction that can be performed on an instruction cache.

If instruction cache invalidation by address is performed on a memory location, the effect of that invalidation is visible only to the virtual address supplied with the operation. The effect of the invalidation might not be visible to any other virtual address aliases of that physical memory location.

The only architecturally-guaranteed way to invalidate all aliases of a physical address from a VIPT instruction cache is to invalidate the entire instruction cache.

An implementation that provides VIPT instruction caches implements the IVIPT Extension, see *The IVIPT architecture Extension* on page G4-4076.

### ASID and VMID tagged VIVT instruction caches

For ASID and VMID tagged VIVT instruction caches, if the instructions at any virtual address change, for a given translation regime and a given ASID and VMID, as appropriate, then instruction cache maintenance is required to ensure that the change is visible to subsequent execution. This maintenance is required when writing new values to instruction locations. It can also be required as a result of any of the following situations that change the translation of a virtual address to a physical address, if, as a result of the change to the translation, the instructions at the virtual addresses change:

- For any translation regime other than the Non-secure PL1&0 translation regime, enabling or disabling stage 1 translations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For the Non-secure PL1&0 translation regime:
  - When stage 2 translations are enabled, enabling or disabling stage 1 translations unless accompanied by a change of VMID.
  - When stage 2 translations are disabled, enabling or disabling stage 1 translations.
  - Enabling or disabling stage 2 translations.
- Writing new mappings to the translation tables.
- In AArch32 state, any change to the TTBR0, TTBR1, or TTBCR registers, unless:
  - For a change to the Secure PL1&0 translation regime, the change is accompanied by a change to the ASID.
  - For a change to the stage 1 translations of the Non-secure PL1&0 translation regime, the change is accompanied by a change to the ASID or a change to the VMID.
- In AArch32 state, any change to the VTTBR or VTCR registers, unless accompanied by a change to the VMID.

**Note**

For ASID and VMID tagged VIVT instruction caches only, for a given translation regime and a given ASID and VMID, as appropriate, invalidation is not required if a change to the translations is such that the instructions associated with the non-faulting translations of a virtual address remain unchanged through the change to the translations, even if the physical locations being mapped to by the changed translation have been written as part of changing the translation.

Examples of situations where this might occur include:

- Copy-on-Write.
- Demand Paging of memory locations to/from disk.

This does not apply for VIPT or PIPT instruction caches, because those caches hold copies of physical addresses, and therefore must be invalidated when the contents are written to, to avoid the use of stale entries.

If instruction cache invalidation by address is performed on a memory location, the effect of that invalidation is visible only to the virtual address supplied with the operation. The effect of the invalidation might not be visible to any other virtual address aliases of that physical memory location.

The only architecturally-guaranteed way to invalidate all aliases of a physical address from an ASID and VMID tagged VIVT instruction cache is to invalidate the entire instruction cache.

### The IVIPT architecture Extension

An implementation in which the instruction cache exhibits the behaviors described in *PIPT instruction caches* on page G4-4075, or those described in *VIPT instruction caches* on page G4-4075, is said to implement the *IVIPT Extension* to the ARM architecture.

The formal definition of the IVIPT Extension to the ARM architecture is that it reduces the instruction cache maintenance requirement to the following condition:

- Instruction cache maintenance is required only after writing new data to a physical address that holds an instruction.

## G4.11.3 Cache maintenance requirement created by changing translation table attributes

Any change to the translation tables to change the attributes of an area of memory can require maintenance of the translation tables, as described in *General TLB maintenance requirements* on page G4-4061. If the change affects the cacheability attributes of the area of memory, including any change between Write-Through and Write-Back attributes, software must ensure that any cached copies of affected locations are removed from the caches, typically


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

by cleaning and invalidating the locations from the levels of cache that might hold copies of the locations affected by the attribute change. Any of the following changes to the inner cacheability or outer cacheability attribute creates this maintenance requirement:

- Write-Back to Write-Through.
- Write-Back to Non-cacheable.
- Write-Through to Non-cacheable.
- Write-Through to Write-Back.

The cache clean and invalidate avoids any possible coherency errors caused by mismatched memory attributes.

Similarly, to avoid possible coherency errors caused by mismatched memory attributes, the following sequence must be followed when changing the Shareability attributes of a cacheable memory location:

1. Make the memory location Non-cacheable, Outer Shareable.
2. Clean and invalidate the location from them cache.
3. Change the Shareability attributes to the required new values.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.12 VMSAv8-32 memory aborts

In a VMSAv8-32 implementation, the following mechanisms cause a PE to take an exception on a failed memory access:

**Debug exception** An exception caused by the debug configuration, see Chapter G2 *AArch32 Self-hosted Debug* and *Exception Catch debug event* on page H3-4875.

**Alignment fault** An Alignment fault is generated if the address used for a memory access does not have the required alignment for the operation. For more information see *Unaligned data access* on page E2-2363 and *Alignment faults* on page G4-4085.

**MMU fault** An MMU fault is a fault generated by the fault checking sequence for the current translation regime. See *MMU faults in AArch32 state* on page G4-4086.

**External abort** Any memory system fault other than a Debug exception, an Alignment fault, or an MMU fault.

Collectively, these mechanisms are called *aborts*. Chapter G2 *AArch32 Self-hosted Debug* and Chapter H3 *Halting Debug Events* describe Debug exceptions, and the remainder of this section describes Alignment faults, MMU faults, and External aborts.

The exception generated on a synchronous memory abort:

- On an instruction fetch is called the Prefetch Abort exception.
- On a data access is called the Data Abort exception.

**Note**

The Prefetch Abort exception applies to any synchronous memory abort on an instruction fetch. It is not restricted to speculative instruction fetches.

In AArch32 state, asynchronous memory aborts are a type of External abort, and are treated as a type of Data Abort exception.

The following sections describe the abort mechanisms:

- *Routing of aborts taken to AArch32 state*.
- *VMSAv8-32 MMU fault terminology* on page G4-4081.
- *The MMU fault-checking sequence* on page G4-4081.
- *Alignment faults* on page G4-4085.
- *MMU faults in AArch32 state* on page G4-4086.
- *External abort on a translation table walk* on page G4-4088.
- *AArch32 prioritization of synchronous aborts from a single stage of address translation* on page G4-4088.

An access that causes an abort is said to be aborted. On an abort, System registers are used to record context information. For more information see *Exception reporting in a VMSAv8-32 implementation* on page G4-4091.

### G4.12.1 Routing of aborts taken to AArch32 state

A memory abort is either a Data Abort exception or a Prefetch Abort exception. When executing in AArch32 state, depending on the cause of the abort, and possibly on configuration settings, an abort is taken either:

- To the Exception level of the PE mode from which the abort is taken. In this case the abort is taken to AArch32 state.
- To a higher Exception level. In this case the Exception level to which the abort is taken is either:
  - Using AArch32. In this case, this chapter describes how the abort is handled.
  - Using AArch64. In this case, Chapter D4 *The AArch64 Virtual Memory System Architecture* describes how the abort is handled.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an abort taken to an Exception level that is using AArch32, the mode to which a memory abort is taken depends on the reason for the exception, the mode the PE is in when it takes the exception, and configuration settings, as follows:

**Memory aborts taken to Monitor mode**

If an implementation includes EL3, when the value of SCR.EA is 1, all External aborts are taken to EL3, and if EL3 is using AArch32 they are taken to Monitor mode. This applies to aborts taken from Secure modes and from Non-secure modes. For more information see *Asynchronous exception routing controls* on page G1-3803.

**Note**

- Although the referenced section mostly describes the routing of asynchronous exceptions, it includes the SCR.EA control that applies to both synchronous and asynchronous external aborts.
- The SCR is implemented only as part of EL3.

**Memory aborts taken to Secure Abort mode**

If an implementation includes EL3, when the PE is executing in Secure state, all memory aborts that are not routed to EL3 are taken to Secure Abort mode.

**Note**

The only memory aborts that can be routed to Monitor mode are External aborts.

**Memory aborts taken to Hyp mode**

If an implementation includes EL2, when the PE is executing in Non-secure state, the following aborts are taken to EL2. If EL2 is using AArch32 this means they are taken to Hyp mode:

- Alignment faults taken:
  - When the PE is in Hyp mode.
  - When the PE is in a Non-secure PL1 or PL0 mode and the exception is generated because the Non-secure PL1&0 stage 2 translation identifies the target of an unaligned access as any type of Device memory.
  - When the PE is in Non-secure User mode and HCR.TGE is set to 1. For more information see *Abort exceptions, when the value of HCR.TGE is 1* on page G1-3792.
- When the PE is using the Non-secure PL1&0 translation regime:
  - MMU faults from stage 2 translations, for which the stage 1 translation did not cause an MMU fault.
  - Any abort taken during the stage 2 translation of an address accessed in a stage 1 translation table walk that is not routed to Secure Monitor mode, see *Stage 2 fault on a stage 1 translation table walk* on page G4-4085.
- When the PE is using the Non-secure PL2 translation regime, MMU faults from stage 1 translations.

  **Note**

  The Non-secure PL2 translation regime has only one stage of translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- External aborts, if SCR.EA is set to 0 and any of the following applies:
  - The PE was executing in Hyp mode when it took the exception.
  - The PE was executing in a Non-secure PL0 or PL1 mode when it took the exception, the abort is asynchronous, and HCR.AMO is set to 1. For more information see *Asynchronous exception routing controls* on page G1-3803.
  - The PE was executing in the Non-secure User mode when it took the exception, the abort is synchronous, and HCR.TGE is set to 1. For more information see *Abort exceptions, when the value of HCR.TGE is 1* on page G1-3792.
  - The abort occurred on a stage 2 translation table walk.
- Debug exceptions, if HDCR.TDE is set to 1. For more information, see *Routing debug exceptions to EL2* on page G1-3795.

**Memory aborts taken to Non-secure Abort mode**

In an implementation that does not include EL3, all memory aborts that are taken to an Exception level that is using AArch32 are taken to Abort mode.

Otherwise, when the PE is executing in Non-secure state, the following aborts are taken to Non-secure Abort mode:

- When the PE is in a Non-secure PL1 or PL0 mode, Alignment faults taken for any of the following reasons:
  - SCTLR.A is set to 1.
  - An instruction that does not support unaligned accesses is committed for execution, and the instruction accesses an unaligned address.
  - The PL1&0 stage 1 translation identifies the target of an unaligned access as any type of Device memory.

    **Note**

    In an implementation that does not include EL2, this case results in a CONSTRAINED UNPREDICTABLE memory access, see *Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE* on page E2-2364 and *Loads and Stores to unaligned locations* on page K1-5424.

  If an implementation includes EL2 and the PE is in Non-secure User mode, these exceptions are taken to Abort mode only if the value of HCR.TGE is 0.
- When the PE is using the Non-secure PL1&0 translation regime, an MMU fault from a stage 1 translation.
- External aborts, if all of the following apply:
  - The abort is not on a stage 2 translation table walk.
  - The PE is not in Hyp mode.
  - The value of SCR.EA is 0.
  - The abort is asynchronous, and HCR.AMO is set to 0.
  - The abort is synchronous, and HCR.TGE is set to 0.
- Virtual Aborts, see *Virtual exceptions when an implementation includes EL2* on page G1-3801.
- When the value of HDCR.TDE is 0, Debug exceptions. For more information, see *Routing debug exceptions to EL2* on page G1-3795.

**Note**

If EL0 is using AArch32 and EL1 is using AArch64 then any of these memory aborts taken from User mode are taken to EL1 as described in Chapter D4 *The AArch64 Virtual Memory System Architecture*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Memory aborts with IMPLEMENTATION DEFINED behavior**

In addition, a PE can generate an abort for an IMPLEMENTATION DEFINED reason associated with lockdown. In an implementation that includes EL2, whether such an abort is taken to Non-secure Abort mode or is taken to EL2 is IMPLEMENTATION DEFINED, and an implementation might include a mechanism to select whether the abort is routed to Non-secure Abort mode or to EL2.

When the PE is in a Non-secure mode other than Hyp mode, if multiple factors cause an Alignment fault, the abort is taken to Non-secure Abort mode if any of the factors require the abort to be taken to Abort mode. For example, if the SCTLR.A bit is set to 1, and the access is an unaligned access to an address that the stage 2 translation tables mark as Device-nGnRnE, then the abort is taken to Non-secure Abort mode.

For more information see *Handling exceptions that are taken to an Exception level using AArch32* on page G1-3774.

### G4.12.2 VMSAv8-32 MMU fault terminology

The ARMv7 Large Physical Address Extension introduced new terminology for faults on a stage of address translation, to provide consistent terminology across all implementations. Table G4-26 shows the terminology used in this manual for an MMU faults, compared with older ARM documentation. The current terms are the same for faults that occur with the Short-descriptor translation table format and with the Long-descriptor format, and also apply to faults in a level 3 lookup when using the Long-descriptor translation table format.

**Table G4-26 MMU fault terminology**

| Current term | Old term | Note |
|---|---|---|
| Level 1 Translation fault | Section Translation fault | - |
| Level 2 Translation fault | Page Translation fault | - |
| Level 3 Translation fault | - | Long-descriptor translation table format only. |
| Level 1 Access flag fault | Section Access flag fault | - |
| Level 2 Access flag fault | Page Access flag fault | - |
| Level 3 Access flag fault | - | Long-descriptor translation table format only. |
| Level 1 Domain fault | Section Domain fault | Short-descriptor translation table format only, except for reporting faults on address translation instructions in the 64-bit PAR, see *Determining the PAR format* on page G4-4113.<br>Cannot occur at level 3. |
| Level 2 Domain fault | Page Domain fault | |
| Level 1 Permission fault | Section Permission fault | - |
| Level 2 Permission fault | Page Permission fault | - |
| Level 3 Permission fault | - | Long-descriptor translation table format only. |

In an implementation that includes EL2, MMU faults are also classified by the translation stage at which the fault is generated. This means that a memory access from a Non-secure PL1 or PL0 mode can generate:

- A stage 1 MMU fault, for example, a stage 1 Translation fault.
- A stage 2 MMU fault, for example, a stage 2 Translation fault.

### G4.12.3 The MMU fault-checking sequence

This section describes the MMU checks made for the memory accesses required for instruction fetches and for explicit memory accesses:

- If an instruction fetch faults it generates a Prefetch Abort exception.
- If an data memory access faults it generates a Data Abort exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For more information about Prefetch Abort exceptions and Data Abort exceptions see *Handling exceptions that are taken to an Exception level using AArch32* on page G1-3774.

In VMSAv8-32, all memory accesses require VA to PA translation. Therefore, when a corresponding stage of address translation is enabled, each access requires a lookup of the translation table descriptor for the accessed VA. For more information, see *Translation tables* on page G4-4004 and subsequent sections of this chapter. MMU fault checking is performed for each level of translation table lookup. If an implementation includes EL2 and is operating in Non-secure state, MMU fault checking is performed for each stage of address translation.

**Note**

In an implementation that includes EL2, if a PE is executing in Non-secure state, the operating system or similar Non-secure system software defines the stage 1 translation tables in the IPA address space, and typically is unaware of the stage 2 translation from IPA to PA. However, each Non-secure stage 1 translation table access is subject to stage 2 address translation, and might be faulted at that stage.

The MMU fault checking sequence is largely independent of the translation table format, as the figures in this section show. The differences are:

**When using the Short-descriptor format**

- There are one or two levels of lookup.
- Lookup always starts at level 1.
- The final level of lookup checks the Domain field of the descriptor and:
  - Faults if there is no access to the Domain.
  - Checks the access permissions only for Client domains.

**When using the Long-descriptor format**

- There are one, two, or three levels of lookup.
- Lookup starts at either level 1 or level 2.
- Domains are not supported. All accesses are treated as Client domain accesses.

The fault-checking sequence shows a translation from an Input address to an Output address. For more information about this terminology, see *About address translation for VMSAv8-32* on page G4-3995.

**Note**

The descriptions in this section do not include the possibility that the attempted address translation generates a TLB conflict abort, as described in *TLB conflict aborts* on page G4-4060.

*MMU faults in AArch32 state* on page G4-4086 describes the faults that an MMU fault-checking sequence can report.

Figure G4-17 on page G4-4083 shows the process of fetching a descriptor from the translation table. For the top-level fetch for any translation, the descriptor is fetched only if the input address passes any required alignment check. As the figure shows, in an implementation that includes EL2, if the translation is stage 1 of the Non-secure PL1&0 translation regime, then the descriptor address is in the IPA address space, and is subject to a stage 2 translation to obtain the required PA. This stage 2 translation requires a recursive entry to the fault checking sequence.

**Note**

Figure G4-17 on page G4-4083 and Figure G4-18 on page G4-4084 give an overview of the fault checking performed by the MMU. See *AArch32 prioritization of synchronous aborts from a single stage of address translation* on page G4-4088 for the complete set of possible faults and their prioritization.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

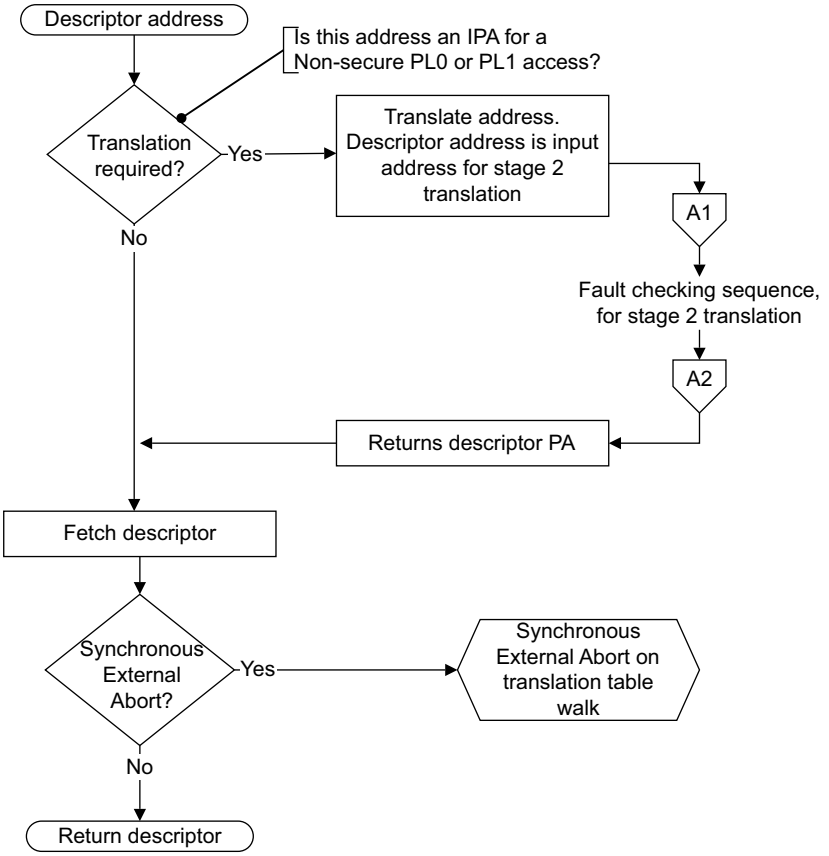



**Figure G4-17 Fetching the descriptor in a VMSAv8-32 translation table walk**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Figure G4-18 shows the full VMSAv8-32 fault checking sequence, including the alignment check on the initial access.

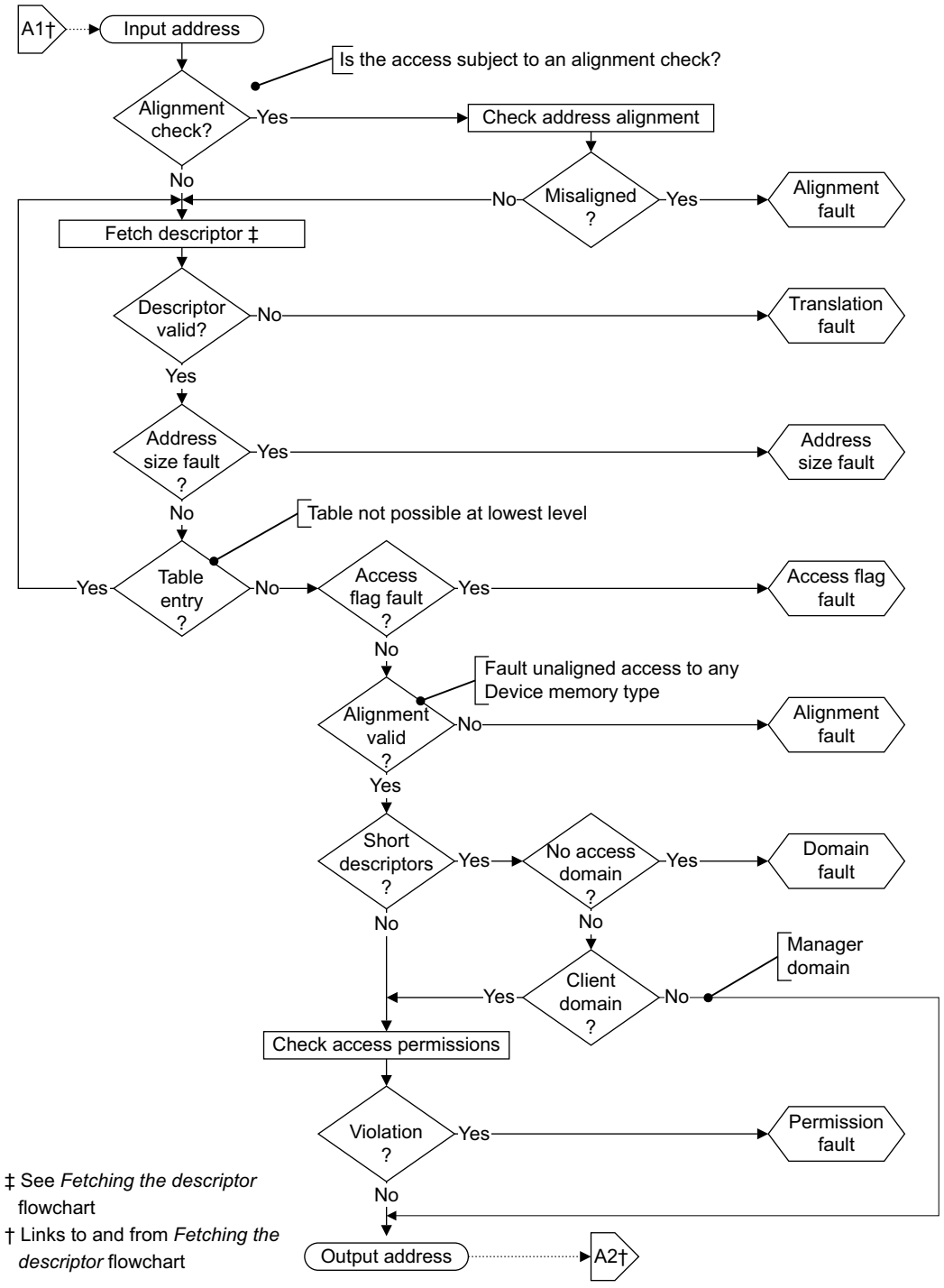



**Figure G4-18 VMSAv8-32 fault checking sequence**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Stage 2 fault on a stage 1 translation table walk

When an implementation that includes EL2 is operating in a Non-secure PL1 or PL0 mode, any memory access goes through two stages of translation:

- Stage 1, from VA to IPA.
- Stage 2, from IPA to PA.

**Note**

In a virtualized system that is using AArch32, typically, a Guest OS operating in a Non-secure PL1 mode defines the translation tables and translation table register entries controlling the Non-secure PL1&0 stage 1 translations. A Guest OS has no awareness of the stage 2 address translation, and therefore believes it is specifying translation table addresses in the physical address space. However, it actually specifies these addresses in its IPA space. Therefore, to support virtualization, translation table addresses for the Non-secure PL1&0 stage 1 translations are always defined in the IPA address space.

On performing a translation table walk for the stage 1 translations, the descriptor addresses must be translated from IPA to PA, using a stage 2 translation. This means that a memory access made as part of a stage 1 translation table lookup might generate, on a stage 2 translation:

- A Translation fault, Access flag fault, or Permission fault.
- A synchronous external abort on the memory access.

If SCR.EA is set to 1, a synchronous external abort is taken to EL3, and if EL3 is using AArch32 it is taken to Secure Monitor mode. Otherwise, these faults are reported as stage 2 memory aborts. When EL2 is using AArch32, HSR.ISS[7] is set to 1, to indicate a stage 2 fault during a stage 1 translation table walk, and the part of the ISS field that might contain details of the instruction is invalid. For more information see *Use of the HSR* on page G4-4105.

Alternatively, a memory access made as part of a stage 1 translation table lookup might target an area of memory with the any type of Device memory attribute assigned on the stage 2 translation of the address accessed. When the value of the HCR.PTW bit is 1, such an access generates a stage 2 Permission fault.

**Note**

- On most systems, such a mapping to a Device memory type on the stage 2 translation is likely to indicate a Guest OS error, where the stage 1 translation table is corrupted. Therefore, it is appropriate to trap this access to the hypervisor.

A TLB might hold entries that depend on the effect of HCR.PTW. Therefore, if HCR.PTW is changed without changing the current VMID, the TLBs must be invalidated before executing in a Non-secure PL1 or PL0 mode. For more information see *Changing HCR.PTW* on page G4-4066.

A cache maintenance instruction performed from a Non-secure PL1 mode can cause a stage 1 translation table walk that might generate a stage 2 Permission fault, as described in this section. This is an exception to the general rule that a cache maintenance instruction cannot generate a Permission fault.

## G4.12.4 Alignment faults

The ARM memory architecture requires support for strict alignment checking. This checking is controlled by SCTLR.A. In addition, some instructions do not support unaligned accesses, regardless of the value of SCTLR.A. *Unaligned data access* on page E2-2363 defines when Alignment faults are generated, for both values of SCTLR.A.

An Alignment fault can occur on an access for which the stage of address translation is disabled.

Any unaligned access to memory region with any Device memory type attribute generates an Alignment fault.

*Routing of aborts taken to AArch32 state* on page G4-4078 defines the mode to which an Alignment fault is taken.

The prioritization of Alignment faults depends on whether the fault was generated because of an access to a Device memory type, or for another reason. For more information see *AArch32 prioritization of synchronous aborts from a single stage of address translation* on page G4-4088.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.12.5 MMU faults in AArch32 state

This section describes the faults that might be detected during one of the fault-checking sequences described in *The MMU fault-checking sequence* on page G4-4081. Unless indicated otherwise, information in this section applies to the fault checking sequences for both the Short-descriptor translation table format and the Long-descriptor translation table format.

MMU faults are always synchronous.

When an MMU fault generates an abort for a region of memory, no memory access is made if that region is or could be marked as any type of Device memory.

The following subsections describe the MMU faults that might be detected during a fault checking sequence:

- *Permission fault*.
- *Translation fault* on page G4-4087.
- *Address size fault* on page G4-4087.
- *Access flag fault* on page G4-4087.
- *Domain fault, Short-descriptor format translation tables only* on page G4-4088.
- *TLB conflict aborts* on page G4-4060.

See also *External abort on a translation table walk* on page G4-4088.

**Note**

- Although the TLB conflict abort is classified as an MMU fault, it is described in the section *Translation Lookaside Buffers (TLBs)* on page G4-4058.
- In VMSAv8-64 an external abort on a translation table walk is classified as an MMU fault. However, in VMSAv8-32, for consistency with earlier versions of the architecture these aborts are not classified as MMU faults.

#### Permission fault

A Permission fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. See *Access permissions* on page G4-4037 for information about conditions that cause a Permission fault.

**Note**

When using the Short-descriptor translation table format, the translation table descriptors are checked for Permission faults only for accesses to memory regions in Client domains.

A TLB might hold a translation table entry that cause a Permission fault. Therefore, if the handling of a Permission fault results in an update to the associated translation tables, the software that updates the translation tables must invalidate the appropriate TLB entry, to prevent the stale information in the TLB being used on a subsequent memory access. For more information, see the translation table entry update examples in *Ordering and completion of TLB maintenance instructions* on page G4-4064.

**Note**

In an implementation that includes EL2, this maintenance requirement applies to Permission faults in both stage 1 and stage 2 translations.

Cache or branch predictor maintenance operations cannot cause a Permission fault, except that:

- A stage 1 translation table walk performed as part of a cache or branch predictor maintenance operation can generate a stage 2 Permission fault as described in *Stage 2 fault on a stage 1 translation table walk* on page G4-4085.
- A DCIMVAC issued in Non-secure state that attempts to update a location for which it does not have stage 2 write access can generate a stage 2 Permission fault, as described in *AArch32 data cache maintenance instructions (DC*)* on page G3-3963.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Translation fault

A Translation fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. A Translation fault is generated if bits[1:0] of a translation table descriptor identify the descriptor as either a Fault encoding or a reserved encoding. For more information see:

- *VMSAv8-32 Short-descriptor translation table format descriptors* on page G4-4010.
- *VMSAv8-32 Long-descriptor translation table format descriptors* on page G4-4019.

In addition, a Translation fault is generated if the input address for a translation either does not map on to an address range of a TTBR, or the TTBR range that it maps on to is disabled. In these cases the fault is reported as a level 1 Translation fault on the translation stage at which the mapping to a region described by a TTBR failed.

The architecture guarantees that any translation table entry that causes a Translation fault is not cached, meaning the TLB never holds such an entry. Therefore, when a Translation fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

A data or unified cache maintenance instruction by VA can generate a Translation fault. It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate a Translation fault.

It is IMPLEMENTATION DEFINED whether a branch predictor maintenance operation can generate a Translation fault.

### Address size fault

An Address size fault can be generated at any level of lookup, and the reported fault code identifies the lookup level.

An Address size fault is generated if the translation table entries or the TTBR for the stage of translation have nonzero address bits above the most significant bit of the maximum output address size. Because VMSAv8-32 supports a maximum PA and IPA size of 40 bits, this means any case where a translation table entry or the TTBR holds an address for which A[47:40] is nonzero generates an Address size fault.

A data or unified cache maintenance instruction by VA can generate an Address size fault. It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate an Address size fault.

It is IMPLEMENTATION DEFINED whether a branch predictor maintenance operation can generate an Address size fault.

The architecture guarantees that any translation table entry that causes an Address size fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Address size fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

### Access flag fault

An Access flag fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. An Access flag fault is generated only if all of the following apply:

- The translation tables support an Access flag bit:
  - The Short-descriptor format supports an Access flag only when SCTLR.AFE is set to 1.
  - The Long-descriptor format always supports an Access flag.
- A translation table descriptor with the Access flag bit set to 0 is loaded.

For more information about the Access flag bit see:

- *VMSAv8-32 Short-descriptor translation table format descriptors* on page G4-4010
- *VMSAv8-32 Long-descriptor translation table format descriptors* on page G4-4019.

The architecture guarantees that any translation table entry that causes an Access flag fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Access flag fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

Whether any cache maintenance instruction by VA can generate Access flag faults is IMPLEMENTATION DEFINED.

Whether branch predictor invalidate by VA operations can generate Access flag faults is IMPLEMENTATION DEFINED.

For more information, see *The Access flag* on page G4-4043.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Domain fault, Short-descriptor format translation tables only

When using the Short-descriptor translation table format, a Domain fault can be generated at level 1or level 2 of lookup. The reported fault code identifies the lookup level. The conditions for generating a Domain fault are:

**Level 1** When a level 1 descriptor fetch returns a valid Section level 1 descriptor, the domain field of that descriptor is checked against the DACR. A level 1 Domain fault is generated if this check fails.

**Level 2** When a level 2 descriptor fetch returns a valid level 2 descriptor, the domain field of the level 1 descriptor that required the level 2 fetch is checked against the DACR, and a level 2 Domain fault is generated if this check fails.

For more information, see *Domains, Short-descriptor format only* on page G4-4042.

Domain faults cannot occur on cache or branch predictor maintenance operations.

A TLB might hold a translation table entry that cause a Domain fault. Therefore, if the handling of a Domain fault results in an update to the associated translation tables, the software that updates the translation tables must invalidate the appropriate TLB entry, to prevent the stale information in the TLB being used on a subsequent memory access. For more information, see the translation table entry update examples in *Ordering and completion of TLB maintenance instructions* on page G4-4064.

Any change to the DACR must be synchronized by a context synchronization operation. For more information see *Synchronization of changes to System registers* on page G4-4132.

## G4.12.6 External abort on a translation table walk

An external abort on a translation table walk can be either synchronous or asynchronous. For more information on external aborts, see *External aborts* on page G3-3976.

An external abort on a translation table walk is reported:

- If the external abort is synchronous, using:
  - A synchronous Prefetch Abort exception if the translation table walk is for an instruction fetch.
  - A synchronous Data Abort exception if the translation table walk is for a data access.
- If the external abort is asynchronous, using an asynchronous Data Abort exception.

If an implementation reports the error in the translation table walk asynchronously from executing the instruction whose instruction fetch or memory access caused the translation table walk, these aborts behave essentially as interrupts. The aborts are masked when PSTATE.A is set to 1, otherwise they are reported using the Data Abort exception.

### Behavior of external aborts on a translation table walk caused by address translation instructions

The address translation instructions summarized in *Address translation instructions, functional group* on page G4-4172 require translation table walks. An external abort can occur in the translation table walk. The abort generates a Data Abort exception, and can be synchronous or asynchronous. For more information, see *Handling of faults and aborts during an address translation instruction* on page G4-4114.

## G4.12.7 AArch32 prioritization of synchronous aborts from a single stage of address translation

*Exception prioritization for exceptions taken to AArch32 state* on page G1-3778 describes the prioritization of exceptions taken from an Exception level that is using AArch32. This section gives additional information about the prioritization of MMU faults from VMSAv8-32 translation regimes.

If a single instruction generates aborts on more than one memory access, the architecture does not define any prioritization between those aborts.

In general, the ARM architecture does not define when asynchronous events are taken, and therefore the prioritization of asynchronous events is IMPLEMENTATION DEFINED.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Note**

The priority numbering in this list only shows the relative priorities of aborts from a single stage of address translation in a VMSAv8-32 translation regime. This numbering has no global significance and, for example, does not correlate with the equivalent AArch64 list in *AArch64 prioritization of synchronous aborts from a single stage of address translation* on page D4-1830.

For a single stage of translation in a VMSAv8-32 translation regime, the priority of the memory management faults on a memory access is as follows, ordered from highest priority to lowest priority. For memory accesses that undergo two stages of translation, the *italic entries show where the faults from second stage translations can occur*. A second stage fault within a second stage translation follows the same priority of faults:

1. Alignment fault not caused by memory type. This is possible for a stage 1 translation only.
2. Translation fault due to the input address being out of the address range to be translated or requiring an AArch32 TTBR that is disabled. This includes VTCR.T0SZ being inconsistent with VTCR.SL0.
3. Address size fault on an AArch32 TTBR caused by the physical address being out of the range implemented.
4. *Second stage abort on a level 1 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the physical address being out of the range implemented*. This is second stage abort during a first stage translation table walk.
5. Synchronous parity or ECC error on a level 1 lookup of a translation table walk.
6. Synchronous external abort on a level 1 lookup level of a translation table walk.
7. Translation fault on a level 1 translation table entry.
8. Address size fault on a level 1 lookup translation table entry caused by the output address being out of the range implemented.
9. *Second stage abort on a level 2 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the physical address being out of the range implemented*. This is second stage abort during a first stage translation table walk.
10. Synchronous parity or ECC error on a level 2 lookup of a translation table walk.
11. Synchronous external abort on a level 2 lookup level of a translation table walk.
12. Translation fault on a level 2 translation table entry.
13. Address size fault on a level 2 lookup translation table entry caused by the output address being out of the range implemented.
14. *Second stage abort on a level 3 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the physical address being out of the range implemented*. This is second stage abort during a first stage translation table walk.
15. Synchronous parity or ECC error on a level 3 lookup of a translation table walk.
16. Synchronous external abort on a level 3 lookup level of a translation table walk.
17. Translation fault on a level 3 translation table entry.
18. Address size fault on a level 3 lookup translation table entry caused by the output address being out of the range implemented.
19. Access Flag fault.
20. Alignment fault caused by the memory type.
21. Domain fault.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Domain faults are possible only when using the VMSAv8-32 Short-descriptor translation table format, see *Domain fault, Short-descriptor format translation tables only* on page G4-4088.

22. Permission fault.
23. *A fault from the stage 2 translation of the memory access. When stage 2 address translation is enabled this includes an Address size fault caused by the physical address being out of the range implemented*.
24. Synchronous parity or ECC error on the memory access.
25. Synchronous External Abort on the memory access.

**Note**

- The prioritization of TLB Conflict aborts is IMPLEMENTATION DEFINED, as the exact cause of these aborts depends on the form of TLBs implemented. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.
- The prioritization of IMPLEMENTATION DEFINED MMU faults for a Load-Exclusive or Store-Exclusive to an unsupported memory type is IMPLEMENTATION DEFINED.

See also *The MMU fault-checking sequence* on page G4-4081.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G4.13 Exception reporting in a VMSAv8-32 implementation

This section describes exception reporting, in AArch32 state, in a VMSAv8-32 implementation. That is, it describes only the reporting of exceptions that are taken to an Exception level that is using AArch32. EL2 provides an enhanced reporting mechanism for exceptions taken to the Non-secure EL2 mode, Hyp mode. This means that, for VMSAv8-32, the exception reporting depends on the mode to which the exception is taken.

**Note**

The enhanced reporting mechanism for exceptions that are taken to Hyp mode is generally similar to the reporting of exceptions that are taken to an Exception level that is using AArch64.

*About exception reporting* introduces the general approach to exception reporting, and the following sections then describe exception reporting at different privilege levels:

- *Reporting exceptions taken to PL1 modes* on page G4-4092.
- *Fault reporting in PL1 modes* on page G4-4095.
- *Summary of register updates on faults taken to PL1 modes* on page G4-4099.
- *Reporting exceptions taken to Hyp mode* on page G4-4101.
- *Use of the HSR* on page G4-4105.
- *Summary of register updates on exceptions taken to Hyp mode* on page G4-4107.

**Note**

The registers used for exception reporting also report information about debug exceptions. For more information see:

- *Data Abort exceptions, taken to a PL1 mode* on page G4-4093.
- *Prefetch Abort exceptions, taken to a PL1 mode* on page G4-4095.
- *Reporting exceptions taken to Hyp mode* on page G4-4101.

### G4.13.1 About exception reporting

In an implementation that includes EL2 and EL3, exceptions can be taken to:

- Monitor mode, if EL3 is using AArch32.
- Hyp mode, if EL2 is using AArch32.
- A Secure or Non-secure PL1 mode.

Monitor mode is a PL1 mode, but:

- It is accessible only when EL3 is using AArch32.
- It is present only in Secure state.
- When EL3 is using AArch32, System register controls route some exceptions from Non-secure state to Monitor mode. These are the only cases where taking an exception to an Exception level that is using AArch32 changes the Security state of the PE.

Exception reporting in Hyp mode differs significantly from that in the other modes, but in general, exception reporting returns:

- Information about the exception:
  - On taking an exception to Hyp mode, the *Hyp Syndrome Register*, HSR, returns syndrome information.
  - On taking an exception to any other mode, a *Fault Status Register* (FSR) returns status information.
- For synchronous exceptions, one or more addresses associated with the exceptions, returned in *Fault Address Registers* (FARs). For a permitted exception to this requirement see *Fault address reporting on synchronous external aborts* on page G4-4092.

In all modes, additional IMPLEMENTATION DEFINED registers can provide additional information about exceptions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- *PE mode for taking exceptions* on page G1-3784 describes how the mode to which an exception is taken is determined.
- EL2 provides:
  - Specific exception types, that can only be taken from Non-secure PL1 and PL0 modes, and are always taken to Hyp mode.
  - Routing controls that can route some exceptions from Non-secure PL1 and PL0 modes to Hyp mode.

  These exceptions are reported using the same mechanism as the Hyp mode reporting of VMSAv8-32 memory aborts, as described in this section.

Memory system faults generate either a Data Abort exception or a Prefetch Abort exception, as summarized in:

- *Reporting exceptions taken to PL1 modes*.
- *Memory fault reporting in Hyp mode* on page G4-4103.

On an access that might have multiple aborts, the MMU fault checking sequence and the prioritization of aborts determine which abort occurs. For more information, see *The MMU fault-checking sequence* on page G4-4081 and *AArch32 prioritization of synchronous aborts from a single stage of address translation* on page G4-4088.

### Fault address reporting on synchronous external aborts

The general architectural requirement is that, on a synchronous abort, the faulting address is recorded in a *Fault Address Register* (FAR). This requirement is relaxed for the case of a synchronous external abort that is not a synchronous external abort on a translation table walk. In this case only:

- It is IMPLEMENTATION DEFINED whether the faulting address is recorded in a FAR.
- A bit in a fault reporting register, the FnV bit, indicates whether a valid address is recorded.

For exceptions taken to an Exception level that is using AArch32, the details of this reporting depend on whether the exception is taken to:

- A PL1 mode, as described in *Reporting exceptions taken to PL1 modes*.
- Hyp mode, as described in *Reporting exceptions taken to Hyp mode* on page G4-4101.

## G4.13.2 Reporting exceptions taken to PL1 modes

The following sections give general information about the reporting of exceptions when they are taken to a Secure or Non-secure PL1 mode:

- *Registers used for reporting exceptions taken to PL1 modes* on page G4-4093.
- *Data Abort exceptions, taken to a PL1 mode* on page G4-4093.
- *Prefetch Abort exceptions, taken to a PL1 mode* on page G4-4095.

*Fault reporting in PL1 modes* on page G4-4095 then describes the fault reporting in these modes, including the encodings used for reporting the faults.

**Note**

*Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990 describes how the Secure and Non-secure PL1 modes map onto the Exception levels.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Registers used for reporting exceptions taken to PL1 modes

AArch32 state defines the following registers, and register encodings, for exceptions taken to PL1 modes:

- The DFSR holds information about a Data Abort exception.
- The DFAR holds the faulting address for some synchronous Data Abort exceptions.
- The IFSR holds information about a Prefetch Abort exception.
- The IFAR holds the faulting address for some synchronous Prefetch Abort exceptions.

In addition, if implemented, the optional ADFSR and AIFSR can provide additional fault information, see *Auxiliary Fault Status Registers*.

### *Auxiliary Fault Status Registers*

AArch32 state defines the following Auxiliary Fault Status Registers:

- The Auxiliary Data Fault Status Register, ADFSR.
- The Auxiliary Instruction Fault Status Register, AIFSR.

The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED. An implementation can use these registers to return additional fault status information. An example use of these registers is to return more information for diagnosing parity or ECC errors.

An implementation that does not need to report additional fault information must implement these registers as UNK/SBZP. This ensures that an attempt to access these registers from software executing at PL1 does not cause an Undefined Instruction exception.

## Data Abort exceptions, taken to a PL1 mode

On taking a Data Abort exception to a PL1 mode:

- If the exception is on an instruction cache or branch predictor maintenance operation by VA, its reporting depends on the value of TTBCR.EAE. For more information about the registers used when reporting the exception, see *Data Abort on an instruction cache or branch predictor maintenance instruction by VA* on page G4-4094.

- Otherwise, the DFSR is updated with details of the fault, including the appropriate fault status code. If the Data Abort exception is synchronous, DFSR.WnR is updated to indicate whether the faulted instruction was a read or a write. However, if the fault is on a cache maintenance instruction, or on an address translation instruction, WnR is set to 1, to indicate a fault on a write instruction, and the CM bit is set to 1.

  See the register description for more information about the returned fault information. See also *Data Abort on a Watchpoint exception* on page G4-4094.

  If the Data Abort exception is

  — Synchronous, the DFAR is updated with the VA that caused the exception, but see *Fault address reporting on synchronous external aborts* on page G4-4092 for a permitted exception to this requirement.

  — Asynchronous, the DFAR becomes UNKNOWN.

DFSR.WnR is UNKNOWN on an asynchronous Data Abort exception.

For all Data Abort exceptions, if the implementation includes EL3, the Security state of the PE in the mode to which the Data Abort exception is taken determines whether the Secure or Non-secure DFSR and DFAR are updated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Data Abort on an instruction cache or branch predictor maintenance instruction by VA*

If an instruction cache invalidation by VA or branch predictor invalidation by VA operation generates a Data Abort exception that is taken to a PL1 mode, the DFAR is updated to hold the faulting VA. However, the reporting of the fault depends on the value of TTBCR.EAE:

**TTBCR.EAE == 0**

When the value of TTBCR.EAE is 0, it is IMPLEMENTATION DEFINED which of the following is used when reporting the fault:

- The DFSR indicates an Instruction cache maintenance instruction fault, and the IFSR is valid and indicates the cause of the fault, a Translation fault or Access flag fault.
- The DFSR indicates the cause of the fault, a Translation fault or Access flag fault. The IFSR is UNKNOWN.

In either case:

- DFSR.WnR is set to 1.
- DFSR.CM is set to 1, to indicate a fault on a cache maintenance instruction.

**TTBCR.EAE == 1**

When the value of TTBCR.EAE is 1:

- DFSR.CM is set to 1, to indicate a fault on a cache maintenance instruction.
- DFSR.STATUS indicates the cause of the fault, a Translation or Access flag fault.
- DFSR.WnR is set to 1.
- The IFSR is UNKNOWN.

#### *Data Abort on a Watchpoint exception*

On taking a Data Abort exception caused by a watchpoint:

- DFSR.FS is updated to indicate a debug event.
- DFSR.{WnR, Domain} are UNKNOWN.
- DFAR is set to the address that generated the watchpoint

**Note**

- LR_abt indicates the address of the instruction that triggered the watchpoint.
- In some ARMv7 AArch32 implementations, the DBGWFAR is set to the address of the instruction that triggered the watchpoint. In ARMv8 this register is RES0.

A watchpointed address can be any byte-aligned address. The address reported in DFAR might not be the watchpointed address, and:

- For a watchpoint due to an operation other than a Data Cache maintenance instruction, can be any address between and including:
  - The lowest address accessed by the instruction that triggered the watchpoint.
  - The highest watchpointed address accessed by that instruction.

  If multiple watchpoints are set in this range, there is no guarantee of which watchpoint is generated.

  The address must also be within a naturally-aligned block of memory of an IMPLEMENTATION DEFINED power-of-two size, containing a watchpoint address accessed by that location.

  **Note**

  - In particular, there is no guarantee of generating the watchpoint with the lowest address in the range.
  - The IMPLEMENTATION DEFINED power-of-two size must be no larger than the block size of the AArch64 DC ZVA operation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For a watchpoint due to a Data Cache operation, the address is the address passed to the instruction. This might be an address that is above the watchpointed location.

### Prefetch Abort exceptions, taken to a PL1 mode

For a Prefetch Abort exception generated by an instruction fetch, the Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

- If the PE attempts to execute the instruction a Prefetch Abort exception is generated.
- If an instruction fetch is issued but the PE does not attempt to execute the prefetched instruction, no Prefetch Abort exception is generated for that instruction. For example, if the execution flow branches round a prefetched instruction, no Prefetch Abort exception is generated.

In addition, Software Breakpoint Instruction, Breakpoint, and Vector Catch exceptions, generate a Prefetch Abort exception, see *Breakpoint debug events and Vector Catch exception* on page H2-4825.

On taking a Prefetch Abort exception to a PL1 mode:

- The IFSR is updated with details of the fault, including the appropriate fault code. If appropriate, the fault code indicates that the exception was generated by a debug exception.

  See the register description for more information about the returned fault information.
- For a Prefetch Abort exception generated by an instruction fetch, the IFAR is updated with the VA that caused the exception, but see *Fault address reporting on synchronous external aborts* on page G4-4092 for a permitted exception to this requirement.
- For a Prefetch Abort exception generated by a debug exception, the IFAR is UNKNOWN.

If the implementation includes EL3, the security state of the PE in the mode to which it takes the Prefetch Abort exception determines whether the exception updates the Secure or Non-secure IFSR and IFAR.

## G4.13.3 Fault reporting in PL1 modes

The FSRs provide fault information, including an indication of the fault that occurred. The following subsections describe fault reporting in PL1 modes for each of the translation table formats:

- *PL1 fault reporting with the Short-descriptor translation table format* on page G4-4096.
- *PL1 fault reporting with the Long-descriptor translation table format* on page G4-4097.

*Reserved encodings in the IFSR and DFSR encodings tables* on page G4-4099 gives some additional information about the encodings for both formats.

*Summary of register updates on faults taken to PL1 modes* on page G4-4099 shows which registers are updated on each of the reported faults.

*Reporting of External aborts taken from Non-secure state to Monitor mode* describes how the fault status register format is determined for those aborts. For all other aborts, the current translation table format determines the format of the fault status registers.

### Reporting of External aborts taken from Non-secure state to Monitor mode

When an External abort is taken from Non-secure state to Monitor mode:

- For a Data Abort exception, the Secure DFSR and DFAR hold information about the abort.
- For a Prefetch Abort exception, the Secure IFSR and IFAR hold information about the abort.
- The abort does not affect the contents of the Non-secure copies of the fault reporting registers.

Normally, the current translation table format determines the format of the DFSR and IFSR. However, when SCR.EA is set to 1, to route external aborts to Monitor mode, and an external abort is taken from Non-secure state, this section defines the DFSR and IFSR format.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an External abort taken from Non-secure state to Monitor mode, the DFSR or IFSR uses the format associated with the Long-descriptor translation table format, as described in *PL1 fault reporting with the Long-descriptor translation table format* on page G4-4097, if any of the following applies:

- The Secure TTBCR.EAE bit is set to 1.
- The External abort is synchronous and either:
  - It is taken from Hyp mode.
  - It is taken from a Non-secure PL1 or PL0 mode, and the Non-secure TTBCR.EAE bit is set to 1.

Otherwise, the DFSR or IFSR uses the format associated with the Short-descriptor translation table format, as described in *PL1 fault reporting with the Short-descriptor translation table format*.

### PL1 fault reporting with the Short-descriptor translation table format

This subsection describes the fault reporting for a fault taken to a PL1 when address translation is using the Short-descriptor translation table format.

On taking an exception, bit[9] of the FSR is RAZ, or set to 0, if the PE is using this FSR format.

An FSR encodes the fault in a 5-bit FS field, that comprises FSR[10, 3:0]. Table G4-27 shows the encoding of that field. *Summary of register updates on faults taken to PL1 modes* on page G4-4099 shows:

- Whether the corresponding FAR is updated on the fault. That is:
  - For a fault reported in the IFSR, whether the IFAR holds a valid address.
  - For a fault reported in the DFSR, whether the DFAR holds a valid address.
- For faults that update DFSR, whether DFSR.Domain is valid

When reading Table G4-27:

- FS values not shown in the table are reserved.
- FS values shown as DFSR only are reserved for the IFSR.

**Table G4-27 FSR encodings when using the Short-description translation table format**

| FS | Source | | Notes |
|---|---|---|---|
| 00001 | Alignment fault | | DFSR only. Fault on initial lookup |
| 00100 | Fault on instruction cache maintenance | | DFSR only |
| 01100<br>01110 | Synchronous external abort on translation table walk | Level 1<br>Level 2 | - |
| 11100<br>11110 | Synchronous parity or ECC error on translation table walk | Level 1<br>Level 2 | - |
| 00101<br>00111 | Translation fault | Level 1<br>Level 2 | MMU fault |
| 00011[a]<br>00110 | Access flag fault | Level 1<br>Level 2 | MMU fault |
| 01001<br>01011 | Domain fault | Level 1<br>Level 2 | MMU fault |
| 01101<br>01111 | Permission fault | Level 1<br>Level 2 | MMU fault |
| 00010 | Debug event | | See Chapter G2 *AArch32 Self-hosted Debug* |
| 01000 | Synchronous external abort | | - |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-27 FSR encodings when using the Short-description translation table format (continued)**

| FS | Source | Notes |
|---|---|---|
| 10000 | TLB conflict abort | See *TLB conflict aborts* on page G4-4060 |
| 10100 | IMPLEMENTATION DEFINED | Lockdown |
| 10101 | IMPLEMENTATION DEFINED | Unsupported Exclusive access |
| 11001 | Synchronous parity or ECC error on memory access | - |
| 10110 | Asynchronous external abort[b] | DFSR only |
| 11000 | Asynchronous parity or ECC error on memory access[c] | DFSR only |

a. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in VMSAv8-32 mean there should be no possibility of confusing the new use of this encoding with its previous use
b. Including asynchronous data external abort on translation table walk or instruction fetch.
c. Including asynchronous parity or ECC error on translation table walk.

#### *The Domain field in the DFSR*

The DFSR includes a Domain field. This is inherited from previous versions of the VMSA. The IFSR does not include a Domain field. *Summary of register updates on faults taken to PL1 modes* on page G4-4099 describes when DFSR.Domain is valid.

ARM deprecates any use of the Domain field in the DFSR. The Long-descriptor translation table format does not support a Domain field, and future versions of the ARM architecture might not support a Domain field in the Short-descriptor translation table format. ARM strongly recommends that new software does not use this field.

For both Data Abort exceptions and Prefetch Abort exceptions, software can find the domain information by performing a translation table read for the faulting address and extracting the Domain field from the translation table entry.

### PL1 fault reporting with the Long-descriptor translation table format

This subsection describes the fault reporting for a fault taken to a PL1mode when address translation is using the Long-descriptor translation table format.

When the PE takes an exception, bit[9] of the FSR is set to 1 if the PE is using this FSR format.

The FSRs encode the fault in a 6-bit STATUS field, that comprises FSR[5:0]. Table G4-28 shows the encoding of that field. In addition:

- For a fault taken to a PL1 mode, *Summary of register updates on faults taken to PL1 modes* on page G4-4099 shows whether the corresponding FAR is updated on the fault. That is:
  - For a fault reported in the IFSR, whether the IFAR holds a valid address.
  - For a fault reported in the DFSR, whether the DFAR holds a valid address.
- For a fault taken to the Hyp mode, *Summary of register updates on exceptions taken to Hyp mode* on page G4-4107 shows what registers are updated on the fault.

**Table G4-28 FSR encodings when using the Long-descriptor translation table format**

| STATUS[a] | Source | Notes |
|---|---|---|
| 0000LL | Address size fault. LL bits indicate level[b]. | MMU fault |
| 0001LL | Translation fault. LL bits indicate level[b]. | MMU fault |
| 0010LL | Access flag fault. LL bits indicate level[b]. | MMU fault |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-28 FSR encodings when using the Long-descriptor translation table format (continued)**

| STATUS[a] | Source | Notes |
|---|---|---|
| 0011LL | Permission fault. LL bits indicate level[b]. | MMU fault |
| 010000 | Synchronous external abort. | - |
| 011000 | Synchronous parity or ECC error on memory access. | - |
| 010001 | Asynchronous external abort. | DFSR only |
| 011001 | Asynchronous parity or ECC error on memory access. | DFSR only |
| 0101LL | Synchronous external abort on translation table walk.<br>LL bits indicate level[b]. | - |
| 0111LL | Synchronous parity or ECC error on memory access on translation table walk.<br>LL bits indicate level[b]. | - |
| 100001 | Alignment fault. | Fault on initial lookup |
| 100010 | Debug event. | See Chapter G2 *AArch32 Self-hosted Debug* |
| 110000 | TLB conflict abort. | See *TLB conflict aborts* on page G4-4060 |
| 110100 | IMPLEMENTATION DEFINED. | Lockdown, DFSR only |
| 110101 | IMPLEMENTATION DEFINED. | Unsupported Exclusive access |
| 1111LL | Domain fault.<br>LL bits indicate level[b]. | MMU fault. 64-bit PAR only, level 1 or level 2 only. Never used in DFSR, IFSR, or HSR[c] |

a. STATUS values not shown in this table are reserved. STATUS values not supported in the IFSR or DFSR are reserved for the register or registers in which they are not supported.

b. See *The level associated with MMU faults*.

c. A Domain fault can be reported using the Long-descriptor STATUS encodings only as a result of a fault on an address translation instruction. For more information see *MMU fault on an address translation instruction* on page G4-4114.

### *The level associated with MMU faults*

For MMU faults, Table G4-29 shows how the LL bits in the xFSR.STATUS field encode the lookup level associated with the fault.

**Table G4-29 Use of LL bits to encode the lookup level at which the fault occurred**

| LL bits | Meaning | |
|---|---|---|
| 00 | **Address size fault** | Address size fault in TTBR0 or TTBR1. |
| | **All other faults** | Reserved. |
| 01 | Level 1. | |
| 10 | Level 2. | |
| 11 | Level 3. When xFSR.STATUS indicates a Domain fault, this value is reserved. | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

### Reserved encodings in the IFSR and DFSR encodings tables

With both the Short-descriptor and the Long-descriptor FSR format, the fault encodings reserve a single encoding for each of:

- Cache and TLB lockdown faults. The details of these faults and any associated subsidiary registers are IMPLEMENTATION DEFINED.
- Aborts associated with coprocessors. The details of these faults are IMPLEMENTATION DEFINED.

## G4.13.4 Summary of register updates on faults taken to PL1 modes

For faults that generate exceptions that are taken to a PL1 mode, Table G4-30 shows the registers affected by each fault. In this table:

- Yes indicates that the register is updated.
- UNK indicates that the fault makes the register value UNKNOWN.
- A null entry, -, indicates that the fault does not affect the register.

For faults that update the DFSR using the Short-descriptor format FSR encodings, Table G4-31 on page G4-4100 shows whether DFSR.Domain is valid.

**Table G4-30 Effect of a fault taken to a PL1 mode on the reporting registers**

| Fault | IFSR | IFAR | DFSR | DFAR |
|---|---|---|---|---|
| Faults reported as Prefetch Abort exceptions: | | | | |
| MMU fault, always synchronous | Yes | Yes | - | - |
| Synchronous external abort on translation table walk | Yes | Yes | - | - |
| Synchronous parity or ECC error on translation table walk | Yes | Yes | - | - |
| Synchronous external abort | Yes | IMP DEF[a] | - | - |
| Synchronous parity or ECC error on memory access | Yes | Yes | - | - |
| TLB conflict abort | Yes | Yes | - | - |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-30 Effect of a fault taken to a PL1 mode on the reporting registers (continued)**

| Fault | | IFSR | IFAR | DFSR | DFAR |
|---|---|---|---|---|---|
| Fault reported as Data Abort exception: | | | | | |
| Alignment fault, always synchronous | | - | - | Yes | Yes |
| MMU fault, always synchronous | | - | - | Yes | Yes |
| Fault on instruction cache maintenance, when using Long-descriptor translation table format[b] | | UNK | - | Yes | Yes |
| Fault on instruction cache maintenance, when using Short descriptor translation table format[c] | *either* | Yes | - | Yes | Yes |
| | *or* | UNK | - | Yes | Yes |
| Synchronous external abort on translation table walk | | - | - | Yes | Yes |
| Synchronous parity or ECC error on translation table walk | | - | - | Yes | Yes |
| Synchronous external abort | | - | - | Yes | IMP DEF[a] |
| Synchronous parity or ECC error on memory access | | - | - | Yes | Yes |
| Asynchronous external abort | | - | - | Yes | UNK |
| Asynchronous parity or ECC error on memory access | | - | - | Yes | UNK |
| TLB conflict abort | | - | - | Yes | Yes |
| Debug exceptions: | | | | | |
| Breakpoint, Software Breakpoint Instruction, or Vector Catch[d] | | Yes | UNK | - | - |
| Watchpoint[e] | | - | - | Yes | Yes |

a. IMPLEMENTATION DEFINED. The IFSR.FnV or DFSR.FnV bit indicates whether the register holds a valid address. See *Fault address reporting on synchronous external aborts* on page G4-4092.

b. When using the Long-descriptor translation table format, there is not a specific fault code for a fault on an instruction cache maintenance instruction. For more information see *Data Abort on an instruction cache or branch predictor maintenance instruction by VA* on page G4-4094.

c. The two lines of this entry show the alternative ways of reporting the fault when using the Short-descriptor translation table format. It is IMPLEMENTATION DEFINED which methods is used, see *Data Abort on an instruction cache or branch predictor maintenance instruction by VA* on page G4-4094.

d. Generates a Prefetch Abort exception.

e. Generates a Data Abort exception.

For those faults for which Table G4-30 on page G4-4099 shows that the DFSR is updated, if the fault is reported using the Short-descriptor FSR encodings, Table G4-31 shows whether DFSR.Domain is valid. In this table, UNK indicates that the fault makes DFSR.Domain UNKNOWN.

**Table G4-31 Validity of Domain field on faults that update the DFSR when using the Short-descriptor encodings**

| DFSR.FS | Source | | DFSR.Domain | Notes |
|---|---|---|---|---|
| 00001 | Alignment fault | | UNK | - |
| 00100 | Fault on instruction cache maintenance instruction | | UNK | - |
| 01100 | Synchronous external abort on translation table walk | Level 1 | UNK | - |
| 01110 | | Level 2 | Valid | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-31 Validity of Domain field on faults that update the DFSR when using the Short-descriptor encodings**

| DFSR.FS | Source | | DFSR.Domain | Notes |
|---|---|---|---|---|
| 11100<br>11110 | Synchronous parity or ECC error on translation table walk | Level 1<br>Level 2 | UNK<br>Valid | - |
| 00101<br>00111 | Translation fault | Level 1<br>Level 2 | UNK<br>Valid | MMU fault |
| 00011[a]<br>00110 | Access flag fault | Level 1<br>Level 2 | UNK<br>Valid | MMU fault |
| 01001<br>01011 | Domain fault | Level 1<br>Level 2 | Valid<br>Valid | MMU fault |
| 01101<br>01111 | Permission fault | Level 1<br>Level 2 | UNK<br>UNK | MMU fault |
| 01000 | Synchronous external abort | | UNK | - |
| 10000 | TLB conflict abort | | UNK | - |
| 11001 | Synchronous parity or ECC error on memory access | | UNK | - |
| 10110 | Asynchronous external abort[b] | | UNK | - |
| 11000 | Asynchronous parity or ECC error on memory access[c] | | UNK | - |
| 00010 | Watchpoint | | UNK | |

a. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in VMSAv8-32 mean there should be no possibility of confusing the new use of this encoding with its previous use

b. Including asynchronous data external abort on translation table walk or instruction fetch.

c. Including asynchronous parity or ECC error on translation table walk.

### G4.13.5 Reporting exceptions taken to Hyp mode

Hyp mode is the Non-secure EL2 mode. It is entered by taking an exception to Hyp mode.

**Note**

Software executing in Monitor mode, or at EL3 when EL3 is using AArch64, can perform an exception return to Hyp mode. This means Hyp mode can be entered either by taking an exception, or by a permitted exception return.

When EL2 is using AArch32, the following exceptions are taken to Hyp mode:

- Asynchronous external aborts, IRQ exceptions, and FIQ exceptions, from Non-secure PL0 and PL1 modes, if not routed to Secure Monitor mode, and if configured by the AMO, FMO or IMO bits. For more information see *Asynchronous exception routing controls* on page G1-3803.
- When HCR.TGE is set to 1, all exceptions that would be routed to Non-secure PL1 modes.

  For more information, see *Routing exceptions from Non-secure EL0 to EL2* on page G1-3790.
- When HDCR.TDE is set to 1, any debug exception that would otherwise be taken to a Non-secure PL1 mode, see *Routing debug exceptions to EL2* on page G1-3795.
- The privilege rules for taking exceptions mean that any exception taken from Hyp mode, if not routed to EL3, must be taken to Hyp mode.
- Hypervisor Call exceptions, and Hyp Trap exceptions, are always taken to Hyp mode. These exceptions are supported only as part of EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When EL2 is implemented, various operations from Non-secure PL0 and PL1 modes can be *trapped* to Hyp mode, using the Hyp Trap exception. For more information, see *EL2 configurable controls* on page G1-3855.

These exceptions include any memory system fault that occurs:

- On a memory access from Hyp mode.
- On memory access from a Non-secure PL0 or PL1 mode:
  - On a stage 2 translation, from IPA to PA.
  - On the stage 2 translation of an address accessed in performing a stage 1 translation table walk.

*Memory fault reporting in Hyp mode* on page G4-4103 gives more information about these faults.

The following exceptions provide *syndrome* information in the HSR:

- Any synchronous exception taken to Hyp mode.
- Some exceptions taken from Debug state that would be taken to Hyp mode if the PE was in Non-debug state, see *Exceptions in Debug state* on page H2-4851.

**Note**

- In Debug state, the PE does not change mode on taking an exception.
- As *Exceptions in Debug state* on page H2-4851 describes, some other exceptions taken from Debug state make the HSR UNKNOWN.

The syndrome information in the HSR includes the fault status code otherwise provided by the fault status register, and extends the fault reporting compared to that available for an exception taken to a PL1 mode. For more information, see *Use of the HSR* on page G4-4105.

In addition, for a Debug exception taken to Hyp mode, DBGDSCRint.MOE or DBGDTRRXext.MOE shows what caused the Debug exception. This bit is valid regardless of whether the Debug exception was taken from Hyp mode or from another Non-secure mode.

*Registers used for reporting exceptions taken to Hyp mode* lists all of the registers used for exception reporting in Hyp mode.

## Registers used for reporting exceptions taken to Hyp mode

The following registers are used for reporting exceptions taken to Hyp mode:

- The HSR holds syndrome information for the exception.
- The HDFAR holds the VA associated with a Data Abort exception.
- The HIFAR holds the VA associated with a Prefetch Abort exception.
- The HPFAR holds bits[39:12] of the IPA associated with some aborts on stage 2 address translations.

In addition, if implemented, the optional HADFSR and HAIFSR can provide additional fault information, see *Hyp Auxiliary Fault Syndrome Registers*.

### *Hyp Auxiliary Fault Syndrome Registers*

EL2 also defines encodings for the following Hyp Auxiliary Fault Syndrome Registers:

- The Hyp Auxiliary Data Fault Syndrome Register, HADFSR.
- The Hyp Auxiliary Instruction Fault Syndrome Register, HAIFSR.

An implementation can use these registers to return additional fault status information for aborts taken to Hyp mode. They are the Hyp mode equivalents of the registers described in *Auxiliary Fault Status Registers* on page G4-4093. An example use of these registers is to return more information for diagnosing parity or ECC errors.

The architectural requirements for the HADFSR and HAIFSR are:

- The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- An implementation with no requirement for additional fault reporting can implement these registers as UNK/SBZP, but the architecture does not require it to do so.

### Memory fault reporting in Hyp mode

Prefetch Abort and Data Abort exceptions taken to Hyp mode report memory faults. For these aborts, the HSR contains the following fault status information:

- The HSR.EC field indicates the type of abort, as Table G4-32 shows.
- The HSR.ISS field holds more information about the abort. In particular:
  - Bits[5:0] of this field hold the STATUS field for the abort, using the encodings defined in *PL1 fault reporting with the Long-descriptor translation table format* on page G4-4097.
  - Other subfields of the ISS give more information about the exception, equivalent to the information returned in the FSR for a memory fault reported at PL1.

  See the descriptions of the ISS fields for the memory faults, referenced from the *Syndrome description* column of Table G4-32, for information about the returned fault information.

**Table G4-32 HSR.EC encodings for aborts taken to Hyp mode**

| HSR.EC | Abort | Syndrome description |
|---|---|---|
| 0x20 | Prefetch Abort taken from Non-secure PL0 or PL1 mode | *ISS encoding for an exception from a Prefetch abort* on page G6-4388 |
| 0x21 | Prefetch Abort taken from Hyp mode | |
| 0x24 | Data Abort taken from Non-secure PL0 or PL1 mode | *ISS encoding for an exception from a Data abort* on page G6-4391 |
| 0x25 | Data Abort taken from Hyp mode | |

For more information, see *Use of the HSR* on page G4-4105.

A Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

- If the PE attempts to execute the instruction a Prefetch Abort exception is generated.
- If an instruction fetch is issued but the PE does not attempt to execute the prefetched instruction, no Prefetch Abort exception is generated for that instruction. For example, if the execution flow branches round a prefetched instruction that would abort if the PE attempted to execute it, no Prefetch Abort exception is generated.

#### *Register updates on exception reporting in Hyp mode*

The use of the HSR, and of the other registers listed in *Registers used for reporting exceptions taken to Hyp mode* on page G4-4102, depends on the cause of the Abort. In reporting these faults, in general:

- If the fault generates a synchronous Data Abort exception, the HDFAR holds the associated VA, but see *Fault address reporting on synchronous external aborts* on page G4-4092 for a permitted exception to this requirement.
- If the fault generates a Prefetch Abort exception, the HIFAR holds the associated VA, but see *Fault address reporting on synchronous external aborts* on page G4-4092 for a permitted exception to this requirement.
- In the following cases, the HPFAR holds the faulting IPA:
  - A Translation or Access flag fault on a stage 2 translation.
  - A Translation, Access flag, or Permission fault on the stage 2 translation of an address accessed in a stage 1 translation table walk.
  - A stage 2 Address size fault.

  In all other cases, the HPFAR is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- On a Data Abort exception that is taken to Hyp mode, the HIFAR is UNKNOWN.
- On a Prefetch Abort exception that is taken to Hyp mode, the HDFAR is UNKNOWN.

In addition, the reporting of particular aborts is as follows:

**Abort on the stage 1 translation for a memory access from Hyp mode**

The HDFAR or HIFAR holds the VA that caused the fault. The STATUS subfield of HSR.ISS indicates the type of fault, Translation, Address size, Access flag, or Permission. The HPFAR is UNKNOWN.

**Abort on the stage 2 translation for a memory access from a Non-secure PL1 or PL0 mode**

This includes aborts on the stage 2 translation of a memory access made as part of a translation table walk for a stage 1 translation. The HDFAR or HIFAR holds the VA that caused the fault. The STATUS subfield of HSR.ISS indicates the type of fault, Translation, Address size, Access flag, or Permission.

For any Access flag fault or Translation fault, and also for any Permission fault on the stage 2 translation of a memory access made as part of a translation table walk for a stage 1 translation, the HPFAR holds the IPA that caused the fault. Otherwise, the HPFAR is UNKNOWN.

**Abort caused by a synchronous external abort, or synchronous parity or ECC error, and taken to Hyp mode**

The HDFAR or HIFAR holds the VA that caused the fault, but see *Fault address reporting on synchronous external aborts* on page G4-4092 for a permitted exception to this requirement. The HPFAR is UNKNOWN.

**Data Abort caused by a Watchpoint exception and routed to Hyp mode because HDCR.TDE is set to 1**

When HDCR.TDE is set to 1, a Watchpoint exception generated in a Non-secure PL1 or PL0 mode, that would otherwise generate a Data Abort exception, is routed to Hyp mode and generates a Hyp Trap exception.

HDFAR is set to the address that generated the watchpoint.

**Note**

ELR_hyp indicates the address of the instruction that triggered the watchpoint.

A watchpointed address can be any byte-aligned address. The address reported in HDFAR might not be the watchpointed address, and, for a watchpoint due to an operation other than a Data Cache maintenance instruction, can be any address between and including:

- The lowest address accessed by the instruction that triggered the watchpoint.
- The highest watchpointed address accessed by that instruction.

If multiple watchpoints are set in this range, there is no guarantee of which watchpoint is generated.

**Note**

In particular, there is no guarantee of generating the watchpoint with the lowest address in the range.

The address must also be within a naturally-aligned block of memory of an IMPLEMENTATION DEFINED power-of-two size, containing a watchpoint address accessed by that location.

**Note**

The IMPLEMENTATION DEFINED power-of-two size must be no larger than the block size of the AArch64 DC ZVA operation.

See also *Watchpoint exceptions* on page G2-3923.

In all cases, HPFAR is UNKNOWN.

**Prefetch Abort caused by a Software Breakpoint Instruction exception and taken to Hyp mode**

This abort is generated if a `BKPT` instruction is executed in Hyp mode. The abort leaves the HIFAR and HPFAR UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See also *Breakpoint debug events and Vector Catch exception* on page H2-4825.

**Prefetch Abort caused by a Software Breakpoint Instruction, Breakpoint, or Vector Catch exception, and routed to Hyp mode because HDCR.TDE is set to 1**

When HDCR.TDE is set to 1, a debug exception, generated in a Non-secure PL1 or PL0 mode, that would otherwise generate a Prefetch Abort exception, is routed to Hyp mode and generates a Hyp Trap exception.

The abort leaves the HIFAR and HPFAR UNKNOWN. This is identical to the reporting of a Prefetch Abort exception caused by a Debug exception on a `BKPT` instruction that is executed in Hyp mode.

**Note**

The difference between these two cases is:

- The Debug exception on a `BKPT` instruction executed in Hyp mode generates a Prefetch Abort exception, taken to Hyp mode, and reported in the HSR using EC value `0x21`.
- Aborts generated because HDCR.TDE is set to 1 generate a Hyp Trap exception, and are reported in the HSR using EC value `0x20`.

## Use of the HSR

The HSR holds syndrome information for any synchronous exception taken to Hyp mode. Compared with the reporting of exceptions taken to PL1 modes, the HSR:

- Always provides details of the fault. The DFSR and IFSR are not used.
- Provides more extensive information, for a wider range of exceptions.

**Note**

IRQ and FIQ exceptions taken to Hyp mode do not report any syndrome information in the HSR.

*HSR, Hyp Syndrome Register* on page G6-4376 describes the HSR, this section summarizes the general form of the register, to show how it encodes exception syndrome information. The register comprises:

- A 6-bit Exception class field, EC, that indicates the cause of the exception.
- An instruction length bit, IL. When an exception is caused by trapping an instruction to Hyp mode, this bit indicates the length of the trapped instruction, as follows:

  **0** 16-bit instruction trapped.

  **1** 32-bit instruction trapped.

  In other cases the IL field is not valid and is RES1.
- An instruction specific syndrome field, ISS. Architecturally, this field could be defined independently for each defined Exception class (EC), but in practice several ISS formats are common to more than one EC.

The format of the HSR depends on the value of the EC field, as follows:

**0b000000<EC≤0b001100**

The ISS part of the returned value includes the CV and COND fields described in *Encoding of ISS[24:20] when 0b000000<EC≤0b001100* on page G4-4106. Figure G4-19 shows the HSR format in this case.

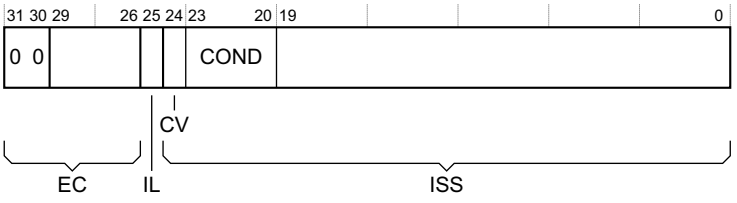



**Figure G4-19 HSR format when the ISS includes CV and COND fields**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EC==0b000000 or EC0b001110** There are no generic fields within the ISS. Figure G4-20 shows the HSR format in this case.

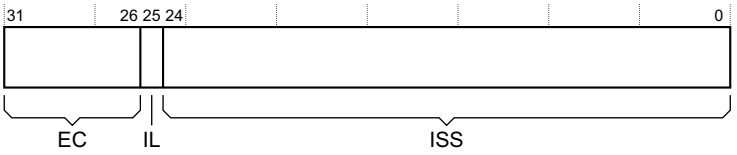



**Figure G4-20 HSR format when the ISS does not include a COND field**

### *Encoding of ISS[24:20] when 0b000000<EC≤0b001100*

For EC values that are nonzero and less than or equal to `0b001100`, ISS[24:20] provides the condition code field for the trapped instruction, together with a valid flag for this field. The encoding of this part of the ISS field is:

**CV, ISS[24]** Condition code valid. Possible values of this bit are:

**0** The COND field is not valid.

**1** The COND field is valid

**COND, ISS[23:20]**

The condition code for the trapped instruction. This field is valid only when CV is set to 1.

If CV is set to 0, this field is UNK/SBZP.

The full descriptions of the HSR.ISS formats give more information about the CV field.

**Note**

In some circumstances, it is IMPLEMENTATION DEFINED whether a conditional instruction that fails its condition code check generates an Undefined Instruction exception, see *Conditional execution of undefined instructions* on page G1-3813.

### *HSR exception classes*

Table G4-33 shows the encoding of the HSR exception class field, EC. Values of EC not shown in the table are reserved. For each EC value, the table references a subsection of the description of the HSR that describes the associated ISS format and gives information about the cause of the exception, for example the configuration required to enable the associated trap.

**Table G4-33 HSR.EC field encoding**

| EC | Exception class | ISS description, or notes |
|---|---|---|
| `0b000000` | Unknown reason | *ISS encoding for exceptions with an unknown reason* on page G6-4379. |
| `0b000001` | Trapped `WFI` or `WFE` instruction | *ISS encoding for an exception from a WFI or WFE instruction* on page G6-4379. |
| `0b000011` | Trapped `MCR` or `MRC` access to CP15 | *ISS encoding for an exception from an MCR or MRC access* on page G6-4380. |
| `0b000100` | Trapped `MCRR` or `MRRC` access to CP15 | *ISS encoding for an exception from an MCRR or MRRC access* on page G6-4382. |
| `0b000101` | Trapped `MCR` or `MRC` access to CP14 | *ISS encoding for an exception from an MCR or MRC access* on page G6-4380. |
| `0b000110` | Trapped `LDC` or `STC` access to CP14 | *ISS encoding for an exception from an LDC or STC access to CP14* on page G6-4383. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-33 HSR.EC field encoding (continued)**

| EC | Exception class | ISS description, or notes |
|---|---|---|
| 0b000111 | HCPTR-trapped access to CP10 or CP11 | *ISS encoding for an exception from an access to SIMD or floating-point functionality, resulting from HCPTR* on page G6-4385. Includes trap of use of Advanced SIMD functionality. |
| 0b001000 | Trapped MRC or VMRS access to CP10, for ID group traps | *ISS encoding for an exception from an MCR or MRC access* on page G6-4380.<br>This trap is not taken if the HCPTR settings trap the access. |
| 0b001100 | Trapped MRRC access to CP14 | *ISS encoding for an exception from an MCRR or MRRC access* on page G6-4382. |
| 0b001110 | Illegal exception return to AArch32 state | *ISS encoding for an exception from an Illegal state or PC alignment fault* on page G6-4390. |
| 0b010001 | Supervisor Call exception routed to Hyp mode | *ISS encoding for an exception from HVC or SVC instruction execution* on page G6-4386. |
| 0b010010 | Hypervisor Call | |
| 0b010011 | Trapped SMC instruction | *ISS encoding for an exception from SMC instruction execution* on page G6-4387. |
| 0b100000 | Prefetch Abort routed to Hyp mode | *ISS encoding for an exception from a Prefetch abort* on page G6-4388. |
| 0b100001 | Prefetch Abort taken from Hyp mode | |
| 0b100010 | PC alignment exception. | *ISS encoding for an exception from an Illegal state or PC alignment fault* on page G6-4390. |
| 0b100100 | Data Abort routed to Hyp mode | *ISS encoding for an exception from a Data abort* on page G6-4391. |
| 0b100101 | Data Abort taken from Hyp mode | |

All EC encodings not shown in Table G4-32 on page G4-4103 are reserved by ARM.

### G4.13.6 Summary of register updates on exceptions taken to Hyp mode

For memory system faults that generate exceptions that are taken to Hyp mode, Table G4-34 on page G4-4108 shows the registers affected by each fault. In this table:

- Yes indicates that the register is updated.
- UNK indicates that the fault makes the register value UNKNOWN.
- A null entry, -, indicates that the fault does not affect the register.

**Note**

For a list of the MMU faults see *MMU faults in AArch32 state* on page G4-4086.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-34 Effect of an exception taken to Hyp mode on the reporting registers**

| Fault | HSR | HIFAR | HDFAR | HPFAR |
|---|---|---|---|---|
| Faults reported as Prefetch Abort exceptions: | | | | |
| MMU fault[a] at stage 1. | Yes | Yes | UNK | UNK |
| Translation or Access flag MMU fault[a] at stage 2. | Yes | Yes | UNK | Yes |
| Other[b] MMU fault[a] at stage 2. | Yes | Yes | UNK | UNK |
| Stage 2 MMU fault[a] on a stage 1 translation. | Yes | Yes | UNK | Yes |
| Synchronous external abort on translation table walk. | Yes | Yes | UNK | UNK |
| Synchronous parity or ECC error on translation table walk. | Yes | Yes | UNK | UNK |
| Synchronous external abort. | Yes | IMP DEF[c] | UNK | UNK |
| Synchronous parity or ECC error on memory access. | Yes | Yes | UNK | UNK |
| Fault reported as Data Abort exception: | | | | |
| MMU fault[a] at stage 1. | Yes | UNK | Yes | UNK |
| Translation or Access flag MMU fault[a] at stage 2. | Yes | UNK | Yes | Yes |
| Other[b] MMU fault[a] at stage 2. | Yes | UNK | Yes | UNK |
| Stage 2 MMU fault[a] on a stage 1 translation. | Yes | UNK | Yes | Yes |
| Synchronous external abort on translation table walk. | Yes | UNK | Yes | UNK |
| Synchronous parity or ECC error on translation table walk. | Yes | UNK | Yes | UNK |
| Synchronous external abort. | Yes | UNK | IMP DEF[c] | UNK |
| Synchronous parity or ECC error on memory access. | Yes | UNK | Yes | UNK |
| Asynchronous external abort. | Yes | UNK | UNK | UNK |
| Asynchronous parity or ECC error on memory access. | Yes | UNK | UNK | UNK |
| Debug exception: | | | | |
| Software Breakpoint Instruction[d], generates a Prefetch Abort exception. | Yes | UNK | - | UNK |
| Debug exception routed to Hyp mode because HDCR.TDE is set to 1. Generates a Hyp Trap exception. | | | | |
| Breakpoint Software Breakpoint Instruction or Vector Catch | Yes | UNK | - | UNK |
| Watchpoint | Yes | - | Yes | UNK |

a. For more information see *Classification of MMU faults taken to Hyp mode* on page G4-4109
b. MMU fault other than a Translation fault or an Access flag fault.
c. IMPLEMENTATION DEFINED. The FnV bit in the HSR.ISS field indicates whether the register holds a valid address. See *Fault address reporting on synchronous external aborts* on page G4-4092.
d. All other debug exceptions are not permitted in Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Unlike Table G4-30 on page G4-4099, the Hyp mode fault reporting table does not include an entry for a fault on an instruction cache maintenance instruction. That is because, when the fault is taken to Hyp mode, the reporting indicates the cause of the fault, for example a Translation fault, and ISS.CM is set to 1 to indicate that the fault was on a cache maintenance instruction, see *ISS encoding for an exception from a Data abort* on page G6-4391.

### Classification of MMU faults taken to Hyp mode

This subsection gives more information about the MMU faults shown in Table G4-34 on page G4-4108.

**Note**

All MMU faults are synchronous.

The table uses the following descriptions for MMU faults taken to Hyp mode:

**MMU fault at stage 1**

This is an MMU fault generated on a stage 1 translation performed in the Non-secure PL2 translation regime.

**MMU fault at stage 2**

This is an MMU fault generated on a stage 2 translation performed in the Non-secure PL1&0 translation regime.

As the table shows, for the faults in this group:

- Translation and Access flag faults update the HPFAR
- Permission faults leave the HPFAR UNKNOWN.

**MMU stage 2 fault on a stage 1 translation**

This is an MMU fault generated on the stage 2 translation of an address accessed in a stage 1 translation table walk performed in the Non-secure PL1&0 translation regime. For more information about these faults see *Stage 2 fault on a stage 1 translation table walk* on page G4-4085.

Figure G4-1 on page G4-3994 shows the different translation regimes and associated stages of translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.14 Address translation instructions

The System register space includes encodings for instructions that either:

- Translate a *virtual address* (VA) to a *physical address* (PA).
- Translate a *virtual address* (VA) to an *intermediate physical address* (IPA).

*Address translation instructions, functional group* on page G4-4172 summarizes these instructions.

When using the Short-descriptor translation table format, all translations performed by these instructions take account of TEX remap when this is enabled, see *Short-descriptor format memory region attributes, with TEX remap* on page G4-4049.

An address translation instruction that executes successfully returns the output address, a PA or an IPA, in the PAR. This is a 64-bit register, that can hold addresses of up to 40 bits.

It is IMPLEMENTATION DEFINED whether the Address translation instructions return the values held in a TLB or the result of a translation table walk. Therefore, ARM recommends that these instructions are not used at a time when the TLB entries might be different from the underlying translation tables held in memory.

The following sections give more information about these instructions:

- *Address translation instruction naming and operation summary*.
- *Encoding and availability of the address translation instructions* on page G4-4112.
- *Determining the PAR format* on page G4-4113.
- *Handling of faults and aborts during an address translation instruction* on page G4-4114.

### G4.14.1 Address translation instruction naming and operation summary

Some older documentation uses the original names for the address translation instructions that were included in the original ARMv7 documentation. Table G4-35 summarizes the instructions that are available in AArch32 state, and relates the old instruction names to the current names.

**Table G4-35 Naming of address translation instructions**

| Name | Old name | Description |
|---|---|---|
| ATS1CPR, ATS1CPW, ATS1CUR, ATS1CUW | V2PCWPR, V2PCWPW, V2PCWUR, V2PCWUW | See *ATS1Cxx, Address translation stage 1, current security state* on page G4-4111 |
| ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, ATS12NSOUW | V2POWPR, V2POWPW, V2POWUR, V2POWUW | See *ATS12NSOxx, Address translation stages 1 and 2, Non-secure state only* on page G4-4111 |
| ATS1HR, ATS1HW | Not applicable[a] | See *ATS1Hx, Address translation stage 1, Hyp mode* on page G4-4112 |

a. Instructions are part of EL2 and have no equivalent in the older descriptions.

In an implementation that does not include EL2, there is no distinction between stage 1 translations and stage 1 and 2 combined translations.

For the *stage 1 current state* and *stages 1 and 2 Non-secure state only* instructions, the meanings of the last two letters of the names are:

**PR** PL1 mode, read operation.
**PW** PL1 mode, write operation.
**UR** User mode, read operation.
**UW** User mode, write operation.

**Note**

User mode can be described as the unprivileged mode. It is the only PL0 mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the *stage 1 Hyp mode* instructions, the last letter of the instruction name is **R** for the read operation and **W** for the write operation.

See also *Encoding and availability of the address translation instructions* on page G4-4112.

### ATS1Cxx, Address translation stage 1, current security state

Any VMSAv8-32 implementation supports the ATS1Cxx instructions. They can be executed by any software executing at PL1 or higher, in either Security state.

The ATS1Cxx instructions are ATS1CPR, ATS1CPW, ATS1CUR, and ATS1CUW. These instructions perform the address translations of the PL1&0 translation regime.

In an implementation that includes EL2, when executed in Non-secure state, these instructions return the IPA that is the output address of the stage 1 translation. Figure G4-1 on page G4-3994 shows the different translation regimes.

**Note**

The Non-secure PL1 and PL0 modes have no visibility of the stage 2 address translations, that can be defined only at PL2, and translate IPAs to be PAs.

See *Determining the PAR format* on page G4-4113 for the format used when returning the result of these instructions.

### ATS12NSOxx, Address translation stages 1 and 2, Non-secure state only

A VMSAv8-32 implementation supports the ATS12NSOxx instructions only if it includes EL2. In an implementation that includes EL2, in AArch32 state, they can be executed:

- By software executing in Non-secure state at EL2. This means by software executing in Hyp mode.
- If the implementation includes EL3, when EL3 is using AArch32, by software executing in Secure state at PL1.

The ATS12NSOxx instructions are ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, and ATS12NSOUW.

In an implementation that includes EL3, when EL3 is using AArch64 and EL1 is using AArch32, any execution of an ATS12NSOxx instruction at Secure EL1 is trapped as an exception that is taken to EL3.

In an implementation that does not include EL2, but includes EL3, when EL3 is using AArch32 these instructions are not UNDEFINED but each instruction behaves in the same way as the equivalent ATS1Cxx instruction.

If an implementation does not include EL2 and does not include EL3 then these instructions are CONSTRAINED UNPREDICTABLE, with the permitted behavior that the instructions are UNDEFINED, see *Unallocated CP14 and CP15 instructions* on page K1-5426.

ARM deprecates use of these instructions from any Secure PL1 mode other than Monitor mode.

In Secure state and in Non-secure Hyp mode these instructions perform the translations made by the Non-secure PL1&0 translation regime.

These instructions always return the PA and final attributes generated by the translation. That is, for an implementation that includes EL2, they return:

- The result of the two stages of address translation for the specified Non-secure input address.
- The memory attributes obtained by the combination of the stage 1 and stage 2 attributes.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

From Hyp mode, the ATS1Cxx and ATS12NSOxx instructions both return the results of address translations that would be performed in the Non-secure modes other than Hyp mode. The difference is:

- The ATS1Cxx instructions return the Non-secure PL1 view of the associated address translation. That is, they return the IPA output address corresponding to the VA input address.
- The ATS12NSOxx instructions return the EL2, or Hyp mode, view of the associated address translation. That is, they return the PA output address corresponding to the VA input address, generated by two stages of translation.

See *Determining the PAR format* on page G4-4113 for the format used when returning the result of these instructions.

### ATS1Hx, Address translation stage 1, Hyp mode

A VMSAv8-32 implementation supports the ATS1Hx instructions only if it includes EL2. They can be executed by:

- Software executing in Non-secure state at PL2. This means by software executing in Hyp mode.
- Software executing in Secure state in Monitor mode.

The ATS1Hx instructions are ATS1HR and ATS1HW. In an implementation that includes EL3, these instructions are CONSTRAINED UNPREDICTABLE if executed in a Secure PL1 mode other than Monitor mode, see *Hyp mode VA to PA address translation instructions* on page K1-5499.

If an implementation does not include EL2 then these instructions are CONSTRAINED UNPREDICTABLE, with the permitted behavior that the instructions are UNDEFINED, see *Unallocated CP14 and CP15 instructions* on page K1-5426.

These instructions perform the translations made by the Non-secure EL2 translation regime. The instruction takes a VA input address and returns a PA output address.

These instructions always return a result in a 64-bit format PAR.

## G4.14.2 Encoding and availability of the address translation instructions

Software executing at PL0 never has any visibility of the address translation instructions, but software executing at PL1 or higher can use the unprivileged address translation instructions to find the address translations used for memory accesses by software executing at PL0 and PL1.

**Note**

For information about translations when the stage of address translation is disabled see *The effects of disabling address translation stages on VMSAv8-32 behavior* on page G4-4000.

Table G4-36 shows the encodings for the address translation instructions, and their availability in different implementations in different PE modes and states.

**Table G4-36 Address translation instructions in AArch32 state**

| opc1 | CRm | opc2 | Name | Type | Description |
|---|---|---|---|---|---|
| All VMSAv8-32 implementations, in all modes, at PL1 or higher, see *ATS1Cxx, Address translation stage 1, current security state* on page G4-4111 | | | | | |
| 0 | c8 | 0 | ATS1CPR | WO | PL1 stage 1 read translation, current state |
| | | 1 | ATS1CPW | WO | PL1 stage 1 write translation, current state |
| | | 2 | ATS1CUR | WO | Unprivileged stage 1 read translation, current state |
| | | 3 | ATS1CUW | WO | Unprivileged stage 1 write translation, current state |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-36 Address translation instructions in AArch32 state (continued)**

| opc1 | CRm | opc2 | Name | Type | Description |
|---|---|---|---|---|---|
| Implementation includes EL2, in Non-secure Hyp mode and Secure PL1 modes, see *ATS12NSOxx, Address translation stages 1 and 2, Non-secure state only* on page G4-4111 | | | | | |
| 0 | c8 | 4 | ATS12NSOPR | WO | Non-secure PL1 stage 1 and 2 read translation |
| | | 5 | ATS12NSOPW | WO | Non-secure PL1 stage 1 and 2 write translation |
| | | 6 | ATS12NSOUR | WO | Non-secure unprivileged stage 1 and 2 read translation |
| | | 7 | ATS12NSOUW | WO | Non-secure unprivileged stage 1 and 2 write translation |
| Implementation includes EL2, in Non-secure Hyp mode and Secure Monitor mode, see *ATS1Hx, Address translation stage 1, Hyp mode* on page G4-4112 | | | | | |
| 4 | c8 | 0 | ATS1HR | WO | Hyp mode stage 1 read translation |
| | | 1 | ATS1HW | WO | Hyp mode stage 1 write translation |

The result of an instruction is always returned in the PAR. The PAR is a RW register and:

- In all implementations, the 32-bit format PAR is accessed using an `MCR` or `MRC` instruction with CRn set to c7, CRm set to c4, and opc1 and opc2 both set to 0.
- The 64-bit format PAR is accessed using an `MCRR` or `MRRC` instruction with CRm set to c7, and opc1 set to 0.

Address translation instructions that are not available in a particular implementation are reserved and CONSTRAINED UNPREDICTABLE. For example:

- In an implementation that does not include EL2, the encodings with an `opc1` value of 4 are reserved and CONSTRAINED UNPREDICTABLE. These are the ATS1Hx operations.
- In an implementation that does not include either EL2 or EL3, the encodings with `opc2` values of 4-7 are reserved and CONSTRAINED UNPREDICTABLE. These are the ATS12NSOxx operations.

The CONSTRAINED UNPREDICTABLE behavior of these encodings is that they are UNDEFINED, see *Unallocated CP14 and CP15 instructions* on page K1-5426.

### G4.14.3 Determining the PAR format

The PAR is a 64-bit register, that supports both 32-bit and 64-bit PAR formats. This section describes how the PAR format is determined, for returning a result from each of the groups of address translation instructions. The returned result might be the translated address, or might indicate a fault on the translation, see *Handling of faults and aborts during an address translation instruction* on page G4-4114.

**ATS1Cxx instructions**

Address translations for the current state. From modes other than Hyp mode:

- TTBCR.EAE determines whether the result is returned using the 32-bit or the 64-bit PAR format.
- If the implementation includes EL3, the translation performed is for the current security state and, depending on that state:
  - The Secure or Non-secure TTBCR.EAE determines the PAR format.
  - The result is returned to the Secure or Non-secure instance of the PAR

Instructions executed in Hyp mode always return a result to the Non-secure PAR, using the 64-bit format.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**ATS12NSOxx instructions**

Address translations for the Non-secure PL1 and PL0 modes. These instructions return a result using the 64-bit PAR format if at least one of the following is true:

- The Non-secure TTBCR.EAE bit is set to 1.
- The implementation includes EL2, and the value of HCR.VM is 1.

Otherwise, the instruction returns a result using the 32-bit PAR format.

Instructions executed in a Secure PL1 mode return a result to the Secure PAR. Instructions executed in Hyp mode return a result to the Non-secure PAR.

**ATS1Hx instructions**

Address translations from Hyp mode. These instructions always return a result using the 64-bit PAR format.

Instructions executed in Secure Monitor mode return a result to the Secure PAR. Instructions executed in Non-secure Hyp mode return a result to the Non-secure PAR.

### G4.14.4 Handling of faults and aborts during an address translation instruction

When a stage of address translation is enabled, any corresponding address translation instruction requires a translation table lookup, and this might require a translation table walk. However, the input address for the translation might be a faulting address, either because:

- The translation table entries used for the translation indicate a fault.
- A stage 2 fault or an external abort occurs on the required translation table walk.

*VMSAv8-32 memory aborts* on page G4-4078 describes the faults that might occur on a translation table walk in AArch32 state.

How the fault is handled, and whether it generates an exception, depends on the cause of the fault, as described in:

- *MMU fault on an address translation instruction*.
- *External abort during an address translation instruction* on page G4-4115.
- *Stage 2 fault on a current state address translation instruction* on page G4-4115.

#### MMU fault on an address translation instruction

In the following cases, an MMU fault on an address translation is reported in the PAR, and no abort is taken. This applies:

- For a faulting address translation instruction executed in Hyp mode, or in a Secure PL1 mode.
- For a faulting address translation instruction executed in a Non-secure PL1 mode, for cases where the fault would generate a stage 1 abort if it occurred on the equivalent load or store operation.

*Using the PAR to report a fault on an address translation instruction* on page G4-4115 gives more information about how these faults are reported.

**Note**

- The Domain fault encodings shown in Table G4-28 on page G4-4097 are used only for reporting a fault on an address translation instruction that uses the 64-bit PAR format. That is, they are used only in an implementation that includes EL2, and are used for reporting a Domain fault on either:
  - An ATS1Cxx operation from Hyp mode.
  - An ATS12NSOxx operation when HCR.VM is set to 1.

  These encodings are never used for fault reporting in the DFSR, IFSR, or HSR.
- For an address translation instruction executed in a Non-secure PL1 mode, for a fault that would generate a stage 2 abort if it occurred on the equivalent load or store operation, the stage 2 abort is generated as described in *Stage 2 fault on a current state address translation instruction* on page G4-4115.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

#### *Using the PAR to report a fault on an address translation instruction*

For a fault on an address translation instruction for which no abort is taken, the PAR is updated with the following information, to indicate the fault:

- The fault code, that would normally be written to the Fault status register. The code used depends on the current translation table format, as described in either:
  - — *PL1 fault reporting with the Short-descriptor translation table format* on page G4-4096.
  - — *PL1 fault reporting with the Long-descriptor translation table format* on page G4-4097.

  See also the Note at the start of *Determining the PAR format* on page G4-4113 about the Domain fault encodings shown in Table G4-28 on page G4-4097.
- A status bit, that indicates that the translation operation failed.

The fault does not update any Fault Address Register.

### External abort during an address translation instruction

As stated in *External abort on a translation table walk* on page G4-4088, an external abort on a translation table walk generates a Data Abort exception. The abort can be synchronous or asynchronous, and behaves as follows:

**Synchronous external abort on a translation table walk**

The fault status and fault address registers of the Security state to which the abort is taken are updated. The fault status register indicates the appropriate external abort on a Translation fault, and the fault address register indicates the input address for the translation.

The PAR is UNKNOWN.

**Asynchronous external abort on a translation table walk**

The fault status register of the Security state to which the abort is taken is updated, to indicate the asynchronous external abort. No fault address registers are updated.

The PAR is UNKNOWN.

### Stage 2 fault on a current state address translation instruction

If the PE is in a Non-secure PL1 mode and performs one of the ATS1C** operations, then a fault in the stage 2 translation of an address accessed in a stage 1 translation table lookup generates an exception. This is equivalent to the case described in *Stage 2 fault on a stage 1 translation table walk* on page G4-4085. When this fault occurs on an ATS1C** address translation instruction:

- A Hyp Trap exception is taken to Hyp mode.
- The PAR is UNKNOWN.
- The HSR indicates that:
  - — The fault occurred on a translation table walk.
  - — The operation that faulted was a cache maintenance instruction.
- The HPFAR holds the IPA that faulted.
- The HDFAR holds the VA that the executing software supplied to the address translation instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.15 About the System registers for VMSAv8-32

In AArch32 state, the System registers are in the coprocessor encoding space, and comprise:

- Registers in the CP15 (coprocessor 15) encoding space, that was called the System Control Coprocessor interface in earlier versions of the architecture.
- Registers in the CP14 encoding space, including:
  - Debug registers.
  - Trace registers.
  - Legacy execution environment registers.
- A small number of registers in the {CP10, CP11} encoding space, used for floating-point and Advanced SIMD functionality.

*Organization of the CP14 registers in VMSAv8-32* on page G4-4138 summarizes the CP14 registers, and indicates where the CP14 registers are described, either in this manual or in other architecture specifications.

*Organization of the CP15 registers in VMSAv8-32* on page G4-4141 summarizes the CP15 registers, and indicates where in this manual the CP15 registers are described.

This section gives general information about the AArch32 System registers, and the conventions used in describing these registers.

**Note**

Many implementations include other interfaces to some System registers, for example a memory-mapped interface to some debug System registers. These are described in the appropriate sections of this manual.

This section is organized as follows:

- *About System register accesses*.
- *General behavior of System registers* on page G4-4118.
- *Classification of System registers* on page G4-4122.
- *Synchronization of changes to System registers* on page G4-4132.
- *Meaning of fixed bit values in register diagrams* on page G4-4137.

### G4.15.1 About System register accesses

Most AArch32 System registers are 32 bits wide. *Accessing 32-bit System registers* on page G4-4117 describes how these registers are accessed.

A small number of the AArch32 System registers are 64 bits wide. *Accessing 64-bit System registers* on page G4-4118 describes how these registers are accessed.

#### Ordering of reads of System registers

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that the data dependencies between the instructions, specified in *Synchronization of changes to System registers* on page G4-4132, are met.

**Note**

In particular, System registers holding self-incrementing counts, for example the Performance Monitors counters or the Generic Timer counter or timers, can be read *early*. This means that, for example, if a memory communication is used to communicate a read of the Generic Timer counter, an `ISB` must be inserted between the read of the memory location used for this communication and the read of the Generic Timer counter if it is required that the Generic Timer counter returns a count value that is later than the memory communication.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing 32-bit System registers

Software accesses most 32-bit System registers using the generic MCR and MRC coprocessor instructions, specifying:

- A coprocessor identifier, coproc, identifying a conceptual coprocessor, as a value p14 or p15, corresponding to CP14 or CP15.

  **Note**

  A general description of MCR and MRC supports coproc values of p0-p15, corresponding to CP0 to CP15. However, ARMv8:

  — Only supports the use of p10, p11, p14, and p15.

  — Only uses p14 and p15 in its specification of MCR and MRC accesses to System registers.

- Two coprocessor registers, CRn and CRm, as values in the range c0-c15. These specify a conceptual coprocessor register within the coprocessor specified by coproc. CRn specifies the primary coprocessor register.
- Two opcodes, opc1 and opc2, as values in the range 0-7.
- A general-purpose register to hold a 32-bit value to transfer to or from the coprocessor.

This means an MCR or MRC access to a specific 32-bit System register uses:

- A unique combination of coproc, CRn, opc1, CRm, and opc2, to specify the required System register.
- A general-purpose register for the transferred 32-bit value.

A small number of AArch32 System registers are accessed using LDC or STC instructions. In these cases, the register to be accessed is identified by {coproc, CRd}, where CRd is the conceptual coprocessor register.

A small number of AArch32 System registers are accessed using MRS, MSR, VMRS, or VMSR instructions, see the appropriate register and instruction description for more information.

**Note**

For example:

- The APSR, CPSR, and SPSR are accessed using MRS or MSR instructions.
- The MVFR0, MVFR1, and MVFR2 are accessed using VMRS or VMSR instructions.

This means a PE can access a 32-bit System register using:

- An MCR instruction to write to a System register, see *MCR, MCR2* on page F6-2801.
- An MRC instruction to read a System register, see *MCR, MCR2* on page F6-2801.
- An LDC access to load data from memory to DBGDTRTXint, see *LDC, LDC2 (immediate)* on page F6-2704 and *LDC, LDC2 (literal)* on page F6-2708.
- An STC access to store data to memory from DBGDTRRXint, see *STC, STC2* on page F6-3026.
- An MRS or VMRS instruction to read a System register, see *MRS* on page F6-2830 and *VMRS* on page F7-3495.
- An MSR or VMSR instruction to write to a System register, see *MSR (immediate)* on page F6-2838, *MSR (register)* on page F6-2840, and *VMSR* on page F7-3497.

In addition, the banked register forms of the MRS and MSR instructions can be used to access some System registers associated with PE modes other than the mode in which the PE is currently executing, see *MRS (Banked register)* on page F6-2832 and *MSR (Banked register)* on page F6-2835.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing 64-bit System registers

Software accesses a 64-bit System register using the generic MCRR and MRRC coprocessor instructions, specifying:

- A coprocessor identifier, coproc, identifying a conceptual coprocessor, as a value p14 or p15, corresponding to CP14 or CP15.

  **Note**

  A general description of MCRR and MRRC supports coproc values of p0-p15, corresponding to CP0 to CP15. However, ARMv8:

  — Only supports the use of p10, p11, p14, and p15.

  — Only uses p14 and p15 in its specification of MCRR and MRRC accesses to System registers.

- A coprocessor register, CRm, as a value in the range c0-c15. This specifies a conceptual coprocessor register within the coprocessor specified by coproc. This is the primary coprocessor register.
- A single opcode, opc1, as a value in the range 0-7.
- Two general-purpose registers to hold two 32-bit values to transfer to or from the coprocessor.

This means an MCRR or MRRC access to a specific 64-bit System register uses:

- A unique combination of coproc, CRm and opc1, to specify the required 64-bit System register.
- Two general-purpose registers, each holding 32 bits of the value to transfer.

This means a PE can access a 64-bit System register using:

- An MCRR instruction to write to a System register, see *MCRR, MCRR2* on page F6-2803.
- An MRRC instruction to read a System register, see *MCRR, MCRR2* on page F6-2803.

When using an MCRR or MRRC instruction:

- Rt contains the least-significant 32 bits of the transferred value, and Rt2 contains the most-significant 32 bits of that value.
- The access is 64-bit atomic.

Some 64-bit registers also have an MCR and MRC encoding. The MCR and MRC encodings for these registers access the least significant 32 bits of the register. For example, to access the PAR, software can:

- Use the following instructions to access all 64 bits of the register:

```
MRRC p15, 0, <Rt>, <Rt2>, c7    ; Read 64-bit PAR into Rt (low word) and Rt2 (high word)
MCRR p15, 0, <Rt>, <Rt2>, c7    ; Write Rt (low word) and Rt2 (high word) to 64-bit PAR
```

- Use the following instructions to access the least-significant 32 bits of the register:

```
MRC p15, 0, <Rt>, c7, c4, 0    ; Read PAR[31:0] into Rt
MCR p15, 0, <Rt>, c7, c4, 0    ; Write Rt to PAR[31:0]
```

## G4.15.2 General behavior of System registers

Except where indicated, System registers are 32-bits wide. As stated in *About System register accesses* on page G4-4116, there are some 64-bit registers, and these include cases where software can access either a 32-bit view or a 64-bit view of a register. The register summaries, and the individual register descriptions, identify the 64-bit registers and how they can be accessed.

The following sections give information about the general behavior of these registers. Unless otherwise indicated, information applies to all AArch32 System registers:

- *Read-only bits in read/write registers* on page G4-4119.
- *UNPREDICTABLE and UNDEFINED behavior for AArch32 System register accesses* on page G4-4119.
- *Read-only and write-only register encodings* on page G4-4121.
- *Reset behavior of System registers* on page G4-4121.

See also *About System register accesses* on page G4-4116 and *Meaning of fixed bit values in register diagrams* on page G4-4137.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Read-only bits in read/write registers

Some read/write registers include bits that are read-only. These bits ignore writes.

An example of this is the SCTLR.NMFI bit, SCTLR[27].

### UNPREDICTABLE and UNDEFINED behavior for AArch32 System register accesses

This section defines UNPREDICTABLE and UNDEFINED behaviors for accesses to System registers, including those cases where the ARMv8 behavior is CONSTRAINED UNPREDICTABLE.

In AArch32 state the following operations are UNDEFINED:

- All `CDP`, `LDC` and `STC` operations to the CP14 and CP15 encoding space, except for the `LDC` access to DBGDTRTXint and the `STC` access to DBGDTRRXint specified in Table G4-47 on page G4-4140.
- All `MCRR` and `MRRC` operations to the CP14 and CP15 encoding space, except for those explicitly defined as accessing 64-bit CP14 and CP15 registers specified in Table G4-46 on page G4-4139 and Table G4-48 on page G4-4147.
- All `CDP2`, `MCR2`, `MRC2`, `MCRR2`, `MRRC2`, `LDC2`, `LDCL`, `LDC2L`, `STC2`, `STCL`, and `STC2L` operations to the CP14 and CP15 encoding space.

Unless otherwise indicated in the individual register descriptions:

- Reserved fields in registers are RES0.
- Assigning a reserved value to a field has a CONSTRAINED UNPREDICTABLE effect, see *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

The following subsections give more information about UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for CP14 and CP15 accesses:

- *Accesses to unallocated encodings in the CP14 and CP15 encoding space*.
- *Additional rules for MCR and MRC accesses to System registers*.
- *Effects of EL3 and EL2 on System register accesses* on page G4-4120.

#### *Accesses to unallocated encodings in the CP14 and CP15 encoding space*

In ARMv8-A, accesses to unallocated register encodings in the CP14 and CP15 encoding space are UNDEFINED, see *Unallocated CP14 and CP15 instructions* on page K1-5426.

#### *Additional rules for MCR and MRC accesses to System registers*

The following operations are CONSTRAINED UNPREDICTABLE for all encodings with `coproc` values `p14` and `p15`:

- All `MCR` operations from the PC.
- All `MRC` operations to `APSR_nzcv`, except for the CP14 `MRC` operation to `APSR_nzcv` from DBGDSCRint.

The CONSTRAINED UNPREDICTABLE behavior of these operations is that they are UNDEFINED, see *Unallocated CP14 and CP15 instructions* on page K1-5426.

For registers and operations that are accessible from a particular Privilege level, any attempt to access those registers from a lower Privilege level is UNDEFINED.

Some individual registers can be made inaccessible by setting configuration bits, possibly including IMPLEMENTATION DEFINED configuration bits, to disable access to the register. The effects of the architecturally-defined configuration bits are defined individually in this manual. Unless explicitly stated otherwise in this manual, setting a configuration bit to disable access to a register results in the register becoming UNDEFINED for `MRC` and `MCR` accesses.

See also *Read-only and write-only register encodings* on page G4-4121.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Effects of EL3 and EL2 on System register accesses*

EL2 and EL3 introduce classes of System registers, described in *Classification of System registers* on page G4-4122. Some of these classes of register are either:

- Accessible only from certain modes or states.
- Accessible from certain modes or states only when configuration settings permit the access.

Accesses to these registers that are not permitted are UNDEFINED, meaning execution of the register access instruction generates an Undefined Instruction exception.

**Note**

This section applies only to registers that are accessible from some modes and states. That is, it applies only to register access instructions using an encoding that, under some circumstances, would perform a valid register access.

The following register classes restrict access in this way:

**Restricted access System registers**

This register class is defined in any implementation that includes EL3.

Restricted access registers other than the NSACR are accessible only from Secure EL3 modes. All other accesses to these registers are UNDEFINED.

The NSACR is a special case of a Restricted access register and:

- The NSACR is:
  - Read/write accessible from Secure PL1 modes.
  - Is Read-only accessible from Non-secure PL2 and PL1 modes.
- All other accesses to the NSACR are UNDEFINED.

For more information, including behavior when EL3 is using AArch64 or is not implemented, see *Restricted access System registers* on page G4-4124.

**Configurable access System registers**

This register class is defined in any implementation that includes EL3.

Most Configurable access registers are accessible from Non-secure state only if control bits in the NSACR permit Non-secure access to the register. Otherwise, a Non-secure access to the register is UNDEFINED.

For other Configurable access registers, control bits in the NSACR control the behavior of bits or fields in the register when it is accessed from Non-secure state. That is, Non-secure accesses to the register are permitted, but the NSACR controls how they behave. The only architecturally-defined register of this type is the CPACR.

For more information, see *Configurable access System registers* on page G4-4125.

**EL2-mode System registers**

This register class is defined only in an implementation that includes EL2.

EL2-mode registers are accessible only from:

- The Non-secure EL2 mode, Hyp mode.
- Secure Monitor mode when SCR.NS is set to 1.

All other accesses to these registers are UNDEFINED.

For more information, see *Hyp mode CP15 read/write registers* on page G4-4125 and *Hyp mode encodings for shared CP15 registers* on page G4-4127.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**EL2-mode write-only operations**

This register class is defined only in an implementation that includes EL2.

EL2-mode write-only operations are accessible only from:

- The Non-secure EL2 mode, Hyp mode.
- Secure Monitor mode, regardless of the value of SCR.NS.

Write accesses to these operations are:

- CONSTRAINED UNPREDICTABLE in Secure EL3 modes other than Monitor mode.
- UNDEFINED in Non-secure modes other than Hyp mode.

For more information, see *Hyp mode CP15 write-only operations* on page G4-4127.

In addition, in any implementation that includes EL3, when EL3 is using AArch32, if write access to a register is disabled by the **CP15SDISABLE** signal then any `MCR` access to that register is UNDEFINED.

## Read-only and write-only register encodings

Some System registers are *read-only* (RO) or *write-only* (WO). For example:

- Most identification registers are read-only.
- Most encodings that perform an operation, such as a cache maintenance instruction, are write-only.

If a particular Privilege level defines a register to be:

- RO, then any attempt to write to that register, at that Privilege level, is UNDEFINED. This means that any access to that register with L == 0 is UNDEFINED.
- WO, then any attempt to read from that register, at that Privilege level, is UNDEFINED. This means that any access to that register with L== 1 is UNDEFINED.

For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED.

**Note**

This section applies only to registers that this manual defines as RO or WO. It does not apply to registers for which other access permissions are explicitly defined.

## Reset behavior of System registers

Reset values apply only to RW registers, although some RO status registers always return a known value. Otherwise, a RW or RO register might return status information about the PE, and this information might be UNKNOWN on reset.

After a reset, only a limited subset of the PE state is guaranteed to be set to defined values. Also, for debug and trace System registers, reset requirements must take account of different levels of reset. For more information about the reset behavior of System registers when the PE resets into an Exception Level that is using AArch32, see:

- *PE state on reset into AArch32 state* on page G1-3830.
- The appropriate Trace architecture specification, for the Trace System registers.
- *Pseudocode description of resetting System registers* on page G4-4122.

When the PE resets into an Exception Level that is using AArch64, PE state that relates to execution in AArch32 state, including the System register values, is UNKNOWN. The only exception to this is state that applies to execution in both AArch64 state and AArch32 state and that has a defined reset value on the reset into AArch64 state. An example of such PE state is the EDPRSR.SR bit.

For a PE reset into an Exception level that is using AArch32, the architecture defines which AArch32 System registers have a defined reset value, and when that defined reset value applies. The register descriptions include this information, and *PE state on reset into AArch32 state* on page G1-3830 summarizes these architectural requirements. Otherwise, RW registers reset to an architecturally UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

In an implementation that includes EL3, unless this manual explicitly states otherwise, only the Secure instance of a Banked register is reset to the defined value. This means that software must program the Non-secure instance of the register with the required values. Typically, this programming is part of the PE boot sequence.

#### *Pseudocode description of resetting System registers*

The `AArch32.ResetControlRegisters()` pseudocode function resets all System registers, and register fields, that have defined reset values, as described in this section and *PE state on reset into AArch32 state* on page G1-3830.

**Note**

For debug and trace System registers this function resets registers as defined for the appropriate level of reset.

### G4.15.3 Classification of System registers

Features provided by EL3 and EL2 integrate with many features of the architecture. Therefore, the descriptions of the individual System registers include information about how these Exception levels affect the register. This section:

- Summarizes how EL3 and EL2 affect the implementation of the System registers, and the classification of those registers.
- Summarizes how EL3 controls access to the System registers.
- Describes an EL3 signal that can control access to some CP15 registers.

It contains the following subsections:

- *Banked System registers* on page G4-4123.
- *Restricted access System registers* on page G4-4124.
- *Configurable access System registers* on page G4-4125.
- *EL2-mode System registers* on page G4-4125.
- *Common System registers* on page G4-4128.
- *The CP15SDISABLE input signal* on page G4-4129.
- *Access to registers from Monitor mode* on page G4-4130.

**Note**

EL3 defines the register classifications of Banked, Restricted access, Configurable, and Common. EL2 defines the EL2-mode classification. Some of these classifications can apply to some CP10 and CP11 coprocessor registers, as well as to the CP14 and CP15 System registers.

It is IMPLEMENTATION DEFINED whether each IMPLEMENTATION DEFINED register is Banked, Restricted access, Configurable, EL2-mode, or Common.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Banked System registers

In an implementation that includes EL3 using AArch32, some System registers are Banked. Banked System registers have two copies, one Secure and one Non-secure. The SCR.NS bit selects the Secure or Non-secure instance of the register. Table G4-37 shows which CP15 registers are Banked in this way, and the permitted access to each register. No CP14 registers are Banked.

**Table G4-37 Banked CP15 registers**

| Banked register | Permitted accesses[a] |
|---|---|
| CSSELR, Cache Size Selection Register | Read/write only at EL1 or higher |
| SCTLR, System Control Register[b] | Read/write only at EL1 or higher |
| ACTLR, Auxiliary Control Register | Read/write only at EL1 or higher |
| ACTLR2, Auxiliary Control Register 2 | Read/write only at EL1 or higher. |
| TTBR0, Translation Table Base 0 | Read/write only at EL1 or higher |
| TTBR1, Translation Table Base 1 | Read/write only at EL1 or higher |
| TTBCR, Translation Table Base Control | Read/write only at EL1 or higher |
| DACR, Domain Access Control Register | Read/write only at EL1 or higher |
| DFSR, Data Fault Status Register | Read/write only at EL1 or higher |
| IFSR, Instruction Fault Status Register | Read/write only at EL1 or higher |
| ADFSR, Auxiliary Data Fault Status Register[c] | Read/write only at EL1 or higher |
| AIFSR, Auxiliary Instruction Fault Status Register[c] | Read/write only at EL1 or higher |
| DFAR, Data Fault Address Register | Read/write only at EL1 or higher |
| IFAR, Instruction Fault Address Register | Read/write only at EL1 or higher |
| PAR, Physical Address Register | Read/write only at EL1 or higher |
| PRRR, Primary Region Remap Register | Read/write only at EL1 or higher |
| NMRR, Normal Memory Remap Register | Read/write only at EL1 or higher |
| VBAR, Vector Base Address Register | Read/write only at EL1 or higher |
| FCSEIDR, FCSE PID Register[c] | Read/write only at EL1 or higher |
| CONTEXTIDR, Context ID Register | Read/write only at EL1 or higher |
| TPIDRURW, User Read/Write Thread ID | Read/write at all privilege levels, including EL0 |
| TPIDRURO, User Read-only Thread ID | Read-only at EL0<br>Read/write at EL1 or higher |
| TPIDRPRW, EL1 only Thread ID | Read/write only at EL1 or higher |

a. Any attempt to execute an access that is not permitted results in an Undefined Instruction exception.
b. Some bits are common to the Secure and the Non-secure copies of the register, see *SCTLR, System Control Register* on page G6-4527.
c. Banked only in an implementation that includes the FCSE. The FCSE PID Register is RAZ/WI if the FCSE is not implemented.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A Banked CP15 register can contain a mixture of:

- Fields that are Banked.
- Fields that are read-only in Non-secure PL1 or PL2 modes but read/write in the Secure state.

The System Control Register SCTLR is an example of a register of that contains this mixture of fields.

The Secure copies of the Banked CP15 registers are sometimes referred to as the Secure Banked CP15 registers. The Non-secure copies of the Banked CP15 registers are sometimes referred to as the Non-secure Banked CP15 registers.

### Restricted access System registers

In an implementation that includes EL3, some System registers are present only in the Secure security state. These are called *Restricted access* registers, and their read/write access permissions are:

- In Non-secure state, software cannot modify Restricted access registers.
- For the NSACR, in Non-secure state:
  - Software running at PL1 or higher can read the register.
  - Unprivileged software, meaning software running at PL0, cannot read the register.

  This means that Non-secure software running at PL1 or higher can read the access permissions for System registers that have Configurable access.

  If EL3 is using AArch64 then any read of the NSACR from Non-secure EL2 using AArch32, or Non-secure EL1 using AArch32, returns the value `0x00000C00`.
- For all other Restricted access registers, Non-secure software cannot read the register.

In an implementation that does not include EL3:

- SDER is implemented only in Secure state.
- Any read of the NSACR returns the value `0x00000C00`.
- All other accesses to Restricted access System registers are UNDEFINED.

Table G4-38 shows the Restricted access CP15 registers. There are no Restricted access CP14 registers.

**Table G4-38 Restricted access CP15 registers**

| Register | Permitted accesses[a] |
|---|---|
| SCR, Secure Configuration | Read/write in Secure PL1 modes |
| SDCR, Secure Debug Configuration Register | Read/write in Secure PL1 modes |
| SDER, Secure Debug Enable | Read/write in Secure PL1 modes |
| NSACR, Non-Secure Access Control | Read/write in Secure PL1 modes<br>Read-only in Non-secure PL1 and PL2 modes |
| MVBAR, Monitor Vector Base Address | Read/write in Secure PL1 modes |

a. Any attempt to execute an access that is not permitted results in an Undefined Instruction exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Configurable access System registers

Secure software can configure the access to some System registers. These registers are called Configurable access registers, and the control can be:

- A bit in the control register determines whether the register is:
  - Accessible from Secure state only.
  - Accessible from both Secure and Non-secure states.
- A bit in the control register changes the accessibility of a register bit or field. For example, setting a bit in the control register might mean that a RW field behaves as RAZ/WI when accessed from Non-secure state.

Bits in the NSACR control access.

In an AArch32 implementation that includes EL3:

- There are no Configurable access CP14 registers.
- The only required Configurable access CP15 register is the CPACR, Coprocessor Access Control Register.
- The following registers in the CP10 and CP11 register space are Configurable access:
  - Floating-point Status and Control Register, FPSCR
  - Floating-point Exception register, FPEXC.
  - Floating-point System ID register, FPSID.
  - Media and VFP Feature Register 0, MVFR0.
  - Media and VFP Feature Register 1, MVFR1.
  - Media and VFP Feature Register 2, MVFR2.

## EL2-mode System registers

In an implementation that includes EL2, if EL2 can use AArch32, the implementation provides a number of registers for use in the EL2 mode, Hyp mode. As with other System register encodings, some of these register encodings provide write-only operations. When the implementation includes EL3 and EL3 is using AArch32, these registers are also accessible from Monitor mode when the value of SCR.NS is 1.

The following subsections describe the EL2-mode registers:

- *Hyp mode CP15 read/write registers*.
- *Hyp mode encodings for shared CP15 registers* on page G4-4127.
- *Hyp mode CP15 write-only operations* on page G4-4127.

There are no EL2-mode CP14 registers.

### *Hyp mode CP15 read/write registers*

These registers are implemented only in Non-secure state, and in Non-secure state they are accessible only from Hyp mode.

Except for accesses to CNTVOFF in an implementation that includes EL3 but not EL2, the behavior of accesses to these registers is as follows:

- In Secure state, the registers can be accessed from EL3 when SCR.NS is set to 1, see *Access to registers from Monitor mode* on page G4-4130.
- The following accesses are UNDEFINED:
  - Accesses from Non-secure PL1 modes.
  - Accesses in Secure state when SCR.NS is set to 0.

In an implementation that includes EL3 but not EL2, the behavior of accesses to CNTVOFF is as follows:

- Any access from Secure Monitor mode is CONSTRAINED UNPREDICTABLE, regardless of the value of SCR.NS. The CONSTRAINED UNPREDICTABLE behavior is that the access is UNDEFINED, see *Unallocated CP14 and CP15 instructions* on page K1-5426.
- All other accesses are UNDEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Except for CNTVOFF, the Hyp mode registers are part of EL2, meaning they are implemented only if the implementation includes EL2. However, conceptually, CNTVOFF is part of any implementation of the Generic Timer, see *Status of the CNTVOFF register* on page G5-4205. This means the behavior of CNTVOFF in an implementation that does not include EL2 is not covered by the general definition of the behavior of the Hyp mode CP15 read/write registers.

Table G4-39 shows the Hyp mode CP15 read/write registers:

**Table G4-39 Hyp mode CP15 read/write registers**

| Register | Width | Permitted accesses[a] |
|---|---|---|
| VPIDR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| VMPIDR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HSCTLR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HACTLR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HACTLR2 | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode. |
| HCR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HDCR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HCPTR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HSTR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HACR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HTCR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| VTCR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HTTBR | 64-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| VTTBR | 64-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HADFSR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HAIFSR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HSR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HPFAR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HMAIR0 | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HMAIR1 | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HAMAIR0 | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HAMAIR1 | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HVBAR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| HTPIDR | 32-bit | Read/write. In Non-secure state, accessible only from Hyp mode |
| CNTVOFF[b] | 64-bit | Read/write. In Non-secure state, accessible only from Hyp mode |

a. Any attempt to execute an access that is not permitted results in an Undefined Instruction exception.
b. Implemented in any implementation of the Generic Timer. See, also, the Note earlier in this section.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Hyp mode encodings for shared CP15 registers

Some Hyp mode registers share the Secure instance of an existing Banked register. In this case the implementation includes an encoding for the register that is accessible only in Hyp mode, or in Monitor mode when SCR.NS is set to 1.

For these registers, the following accesses are UNDEFINED:

- Accesses from Non-secure PL1 modes.
- Accesses in Secure state when SCR.NS is set to 0.

Table G4-40 lists the Hyp mode encodings for shared registers.

**Table G4-40 Hyp mode CP15 register encodings for shared registers**

| Register | Permitted accesses[a] | Shared register |
|---|---|---|
| HDFAR | Read/write. Accessible from Hyp mode, or from Monitor mode when SCR.NS is set to 1. | Secure DFAR |
| HIFAR | Read/write. Accessible from Hyp mode, or from Monitor mode when SCR.NS is set to 1. | Secure IFAR |

a. Any attempt to execute an access that is not permitted results in an Undefined Instruction exception.

In Monitor mode, the Secure copies of these registers can be accessed either:

- Using the DFAR or IFAR encoding with SCR.NS set to 0.
- Using the HDFAR or HIFAR encoding with SCR.NS set to 1.

However, between accessing a register using one alias and accessing the register using the other alias, a *Context synchronization operation* is required to ensure the ordering of the accesses.

### Hyp mode CP15 write-only operations

Architecturally, these encodings are an extension of the Banked register encodings described in *Banked System registers* on page G4-4123, where:

- The implementation does not implement the operation in Secure state.
- In Non-secure state, the operation is accessible only at EL2, that is, only from Hyp mode.

In Secure state:

- These operations can be accessed from Monitor mode regardless of the value of SCR.NS, see *Access to registers from Monitor mode* on page G4-4130.
- Accesses to these operations are CONSTRAINED UNPREDICTABLE if executed in a Secure mode other than Monitor mode, see *Hyp mode TLB maintenance instructions* on page K1-5499 and *Hyp mode VA to PA address translation instructions* on page K1-5499.

Accesses to these operations are UNDEFINED if accessed from a Non-secure PL1 mode.

Table G4-41 shows the EL2-mode CP15 write-only operations:

**Table G4-41 Hyp mode CP15 write-only operations**

| Register | Width | Permitted accesses[a] |
|---|---|---|
| ATS1HR | 32-bit | Write-only. Accessible only from Hyp mode and Monitor mode |
| ATS1HW | 32-bit | Write-only. Accessible only from Hyp mode and Monitor mode |
| TLBIALLHIS | 32-bit | Write-only. Accessible only from Hyp mode and Monitor mode |
| TLBIMVAHIS | 32-bit | Write-only. Accessible only from Hyp mode and Monitor mode |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-41 Hyp mode CP15 write-only operations (continued)**

| Register | Width | Permitted accesses[a] |
|---|---|---|
| TLBIALLNSNHIS | 32-bit | Write-only. Accessible only from Hyp mode and Monitor mode |
| TLBIALLH | 32-bit | Write-only. Accessible only from Hyp mode and Monitor mode |
| TLBIMVAH | 32-bit | Write-only. Accessible only from Hyp mode and Monitor mode |
| TLBIALLNSNH | 32-bit | Write-only. Accessible only from Hyp mode and Monitor mode |

a. This section describes the behavior of write accesses that are not permitted. See also *Read-only and write-only register encodings* on page G4-4121.

For more information about these operations, see *ATS1Hx, Address translation stage 1, Hyp mode* on page G4-4112.

## Common System registers

Some System registers and operations are common to the Secure and Non-secure Security states. These are described as the *Common access* registers, or simply as the *Common* registers. These registers include:

- Read-only registers that hold configuration information.
- Register encodings used for various memory system operations, rather than to access registers.
- The ISR.
- All CP14 registers.

Table G4-42 shows the Common CP15 System registers. These registers are not affected by whether EL3 is implemented.

**Table G4-42 Common CP15 registers**

| Register | Permitted accesses[a] |
|---|---|
| MIDR, Main ID Register | Read-only, only at EL1 or higher |
| CTR, Cache Type Register | Read-only, only at EL1 or higher |
| TCMTR, TCM Type Register[b] | Read-only, only at EL1 or higher |
| TLBTR, TLB Type Register[b] | Read-only, only at EL1 or higher |
| MPIDR, Multiprocessor Affinity Register | Read-only, only at EL1 or higher |
| REVIDR, Revision ID | Read-only, only at EL1 or higher |
| ID_PFRx, Processor Feature Registers | Read-only, only at EL1 or higher |
| ID_DFR0, Debug Feature Register 0 | Read-only, only at EL1 or higher |
| ID_AFR0, Auxiliary Feature Register 0 | Read-only, only at EL1 or higher |
| ID_MMFRx, Memory Model Feature Registers | Read-only, only at EL1 or higher |
| ID_ISARx, Instruction Set Attribute Registers | Read-only, only at EL1 or higher |
| CCSIDR, Cache Size ID Register | Read-only, only at EL1 or higher |
| CLIDR, Cache Level ID Register | Read-only, only at EL1 or higher |
| AIDR, Auxiliary ID Register[b] | Read-only, only at EL1 or higher |
| Cache maintenance instruction | See *Cache maintenance instructions, functional group* on page G4-4169 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-42 Common CP15 registers (continued)**

| Register | Permitted accesses[a] |
|---|---|
| Address translation instructions | See *Address translation instructions, functional group* on page G4-4172 |
| Data barrier operations | Write-only at all privilege levels, including EL0 |
| TLB maintenance instructions | Write-only, only at EL1 or higher |
| Performance monitors | See *Access permissions* on page D5-1893 |
| ISR, Interrupt Status Register | Read-only, only at EL1 or higher |

a. Any attempt to execute an access that is not permitted results in an Undefined Instruction exception.
b. Register or operation details are IMPLEMENTATION DEFINED.

### Secure CP15 registers

The Secure CP15 registers comprise:

- The Secure copies of the Banked CP15 registers.
- The Restricted access CP15 registers.
- The Configurable access CP15 registers that are configured to be accessible only from Secure state.

In an implementation that includes EL3, the Non-secure CP15 registers are the CP15 registers other than the Secure CP15 registers.

### The CP15SDISABLE input signal

When EL3 is using AArch32, it provides an input signal, **CP15SDISABLE**, that disables write access to some of the Secure registers when asserted HIGH. The **CP15SDISABLE** signal has no effect on:

- Register accesses from AArch64 state.
- Register accesses from Secure EL1 when EL3 is using AArch64 and EL1 is using AArch32.

**Note**

When EL3 is using AArch32, the interaction between **CP15SDISABLE** and any IMPLEMENTATION DEFINED register is IMPLEMENTATION DEFINED.

Table G4-43 shows the registers and operations affected.

**Table G4-43 Secure registers affected by CP15SDISABLE**

| Register name | Affected operation |
|---|---|
| SCTLR, System Control Register | `MCR p15, 0, <Rt>, c1, c0, 0` |
| TTBR0, Translation Table Base Register 0 | `MCR p15, 0, <Rt>, c2, c0, 0` |
| TTBCR, Translation Table Base Control Register | `MCR p15, 0, <Rt>, c2, c0, 2` |
| DACR, Domain Access Control Register | `MCR p15, 0, <Rt>, c3, c0, 0` |
| PRRR. Primary Region Remap Register | `MCR p15, 0, <Rt>, c10, c2, 0` |
| NMRR, Normal Memory Remap Register | `MCR p15, 0, <Rt>, c10, c2, 1` |
| AMAIR0, Auxiliary Memory Attribute Indirection Register 0 | `MCR p15, 0, <Rt>, c10, c3, 0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-43 Secure registers affected by CP15SDISABLE (continued)**

| Register name | Affected operation |
|---|---|
| AMAIR1, Auxiliary Memory Attribute Indirection Register 1 | `MCR p15, 0, <Rt>, c10, c3, 1` |
| VBAR, Vector Base Address Register | `MCR p15, 0, <Rt>, c12, c0, 0` |
| MVBAR, Monitor Vector Base Address Register | `MCR p15, 0, <Rt>, c12, c0, 1` |

On a reset by the external system that resets the PE into EL3 using AArch32, the **CP15SDISABLE** input signal must be taken LOW. This permits the Reset code to set up the configuration of EL3 features. When the input is asserted HIGH, any attempt to write to the Secure registers shown in Table G4-43 on page G4-4129 results in an Undefined Instruction exception.

The **CP15SDISABLE** input does not affect reading Secure registers, or reading or writing Non-secure registers. It is IMPLEMENTATION DEFINED how the input is changed and when changes to this input are reflected in the PE, and an implementation might not provide any mechanism for driving the **CP15SDISABLE** input HIGH. However, in an implementation in which the **CP15SDISABLE** input can be driven HIGH, changes in the state of **CP15SDISABLE** must be reflected as quickly as possible. Any change must occur before completion of an Instruction Synchronization Barrier operation, issued after the change, is visible to the PE with respect to instruction execution boundaries. Software must perform an Instruction Synchronization Barrier operation meeting the above conditions to ensure all subsequent instructions are affected by the change to **CP15SDISABLE**.

When EL3 is using AArch32, use of **CP15SDISABLE** means key Secure features that are accessible only at PL1 can be locked in a known state. This provides an additional level of overall system security. ARM expects control of **CP15SDISABLE** to reside in the system, in a block dedicated to security.

### Access to registers from Monitor mode

When the PE is in Monitor mode, the PE is in Secure state regardless of the value of the SCR.NS bit. In Monitor mode, the SCR.NS bit determines whether valid uses of the `MRC`, `MCR`, `MRRC` and `MCRR` instructions access the Secure Banked CP15 registers or the Non-secure Banked CP15 registers. That is, when:

**NS == 0** Common, Restricted access, and Secure Banked registers are accessed by CP15 `MRC`, `MCR`, `MRRC` and `MCRR` instructions.

If the implementation includes EL2, the registers listed in *Hyp mode CP15 read/write registers* on page G4-4125 and *Hyp mode encodings for shared CP15 registers* on page G4-4127 are not accessible, and any attempt to access them generates an Undefined Instruction exception.

**Note**

The operations listed in *Hyp mode CP15 write-only operations* on page G4-4127 are accessible in Monitor mode regardless of the value of SCR.NS.

CP15 operations use the Security state to determine all resources used, that is, all CP15-based operations are performed in Secure state.

**NS == 1** Common, Restricted access and Non-secure Banked registers are accessed by CP15 `MRC`, `MCR`, `MRRC` and `MCRR` instructions.

If the implementation includes EL2, all the registers and operations listed in the subsections of *EL2-mode System registers* on page G4-4125 are accessible, using the `MRC`, `MCR`, `MRRC`, or `MCRR` instructions required to access them from Hyp mode.

CP15 operations use the Security state to determine all resources used, that is, all CP15-based operations are performed in Secure state.

The Security state determines whether the Secure or Non-secure Banked registers determine the control state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Where the contents of a register select the value accessed by an `MRC` or `MCR` access to a different register, then the register that is used for selection is being used as control state. For example, CSSELR selects the current CCSIDR, and therefore CSSELR is used as control state. Therefore, in Monitor mode:

- SCR.NS determines whether the Secure or Non-secure CSSELR is accessible.
- Because the PE is in Secure state, the Secure CSSELR selects the current CCSIDR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.15.4 Synchronization of changes to System registers

In this section, *this PE* means the PE on which accesses are being synchronized.

**Note**

See *Definitions of direct and indirect reads and writes and their side-effects* on page G4-4135 for definitions of the terms *direct write*, *direct read*, *indirect write*, and *indirect read*.

A *direct write* to a System register might become visible at any point after the change to the register, but without a *Context synchronization operation* there is no guarantee that the change becomes visible.

Any direct write to a System register is guaranteed not to affect any instruction that appears, in program order, before the instruction that performed the direct write, and any direct write to a System register must be synchronized before any instruction that appears after the direct write, in program order, can rely on the effect of that write. The only exceptions to this are:

- All direct writes to the same register, using the same encoding, are guaranteed to occur in program order.
- All direct writes to a register are guaranteed to occur in program order relative to all direct reads of the same register using the same encoding.
- Any System register access that an ARM *Architecture Specification* or equivalent specification defines as not requiring synchronization.
- If an instruction that appears in program order before the direct write performs a memory access, such as a memory-mapped register access, that causes an indirect read or write to a register, that memory access is subject to the memory order model. In this case, if permitted by the memory order model, the instruction that appears in program order before the direct write can be affected by the direct write. For information about the memory order model, see *Memory ordering* on page E2-2372.

These rules mean that an instruction that writes to one of the address translation instructions described in *Address translation instructions* on page G4-4110 must be explicitly synchronized to guarantee that the result of the address translation instruction is visible in the PAR.

**Note**

In this case, the direct write to the encoding of the address translation instruction causes an *indirect write* to the PAR. Without a *Context synchronization operation* after the direct write there is no guarantee that the indirect write to the PAR is visible.

Conceptually, the explicit synchronization occurs as the first step of any *Context synchronization operation*. This means that if the operation uses the state that had been changed but not synchronized before the operation occurred, the operation is guaranteed to use the state as if it had been synchronized.

**Note**

- This explicit synchronization is applied as the first step of the execution of any instruction that causes the synchronization operation. This means it does not synchronize any effect of changes to the System registers that might affect the fetch and decode of the instructions that cause the operation, such as breakpoints or changes to translation tables.
- For a synchronous exception, the control state in use at the time the exception is generated determines the exception syndrome information, and this syndrome information is not changed by this synchronization at the start of taking the exception.

Except for the register reads listed in *Registers with some architectural guarantee of ordering or observability* on page G4-4134, if no context synchronization operation is performed, direct reads of System registers can occur in any order.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table G4-44 shows the synchronization requirement between two reads or writes that access the same System register. In the column headings, *First* and *Second* refer to:

- Program order, for any read or write caused by the execution of an instruction by this PE, other than a read or write caused by a memory access made by that instruction.
- The order of arrival of asynchronous reads or writes made by this PE relative to the execution of instructions by this PE.

In addition:

- For indirect reads or writes caused by an external agent, such as a debugger, the mechanism that determines the order of the reads or writes is defined by that external agent. The external agent can provide mechanisms that ensure that any read or write it makes arrives at the PE. These indirect reads and writes are asynchronous to software execution on the PE.
- For indirect reads or writes caused by memory-mapped reads or writes made by this PE, the ordering of the memory accesses is subject to the memory order model, including the effect of the memory type of the accessed memory address. This applies, for example, if this PE reads or writes one of its registers in a memory-mapped register interface.

  The mechanism for ensuring completion of these memory accesses, including ensuring the arrival of the asynchronous read or write at the PE, is defined by the system.

  **Note**

  Such accesses are likely to be given a Device memory attribute, but requiring this is outside the scope of the architecture.

- For indirect reads or writes caused by autonomous asynchronous events that are counted, for example events caused by the passage of time, the events are ordered so that:
  — Counts progress monotonically.
  — The events arrive at the PE in finite time and without undue delay.

**Table G4-44 Synchronization requirements for updates to System registers**

| First read or write | Second read or write | Context synchronization operation required |
|---|---|---|
| Direct read | Direct read | No |
| | Direct write | No |
| | Indirect read | No[a] |
| | Indirect write | No[a], but see text in this section for exceptions |
| Direct write | Direct read | No |
| | Direct write | No |
| | Indirect read | Yes[a] |
| | Indirect write | No, but see text in this section for exceptions |
| Indirect read | Direct read | No |
| | Direct write | No |
| | Indirect read | No |
| | Indirect write | No |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-44 Synchronization requirements for updates to System registers (continued)**

| First read or write | Second read or write | Context synchronization operation required |
|---|---|---|
| Indirect write | Direct read | Yes, but see text in this section for exceptions |
| | Direct write | No, but see text in this section for exceptions |
| | Indirect read | Yes, but see text in this section for exceptions |
| | Indirect write | No, but see text in this section for exceptions |

a. Although no synchronization is required between a Direct write and a Direct read, or between a Direct read and an Indirect write, this does not imply that a Direct read causes synchronization of a previous Direct write. This means that the sequence Direct write followed by Direct read followed by Indirect read, with no intervening context synchronization, does not guarantee that the Indirect read observes the result of the Direct write.

If the indirect write is to a register that *Registers with some architectural guarantee of ordering or observability* shows as having some guarantee of the visibility of an indirect write, synchronization might not be required.

If a direct read or a direct write to a register is followed by an indirect write to that register that is caused by an external agent, or by an autonomous asynchronous event, or as a result of a memory-mapped write, then synchronization is required to guarantee the ordering of the indirect write relative to the direct read or direct write.

If an indirect write caused by a direct write is followed by an indirect write caused by an external agent, or by an autonomous asynchronous event, or as a result of a memory-mapped write, then synchronization is required to guarantee the ordering of the two indirect writes.

If a direct read causes an indirect write, synchronization is required to guarantee that the indirect write is visible to subsequent direct or indirect reads or writes. This synchronization must be performed after the direct read, before any subsequent direct or indirect read or write.

If a direct write causes an indirect write, synchronization is required to guarantee that the indirect write is visible to subsequent direct or indirect reads or writes. This synchronization must be performed after the direct write, before any subsequent direct or indirect read or write.

**Note**

Where a register has more that one encoding, a direct write to the register using a particular encoding is not an indirect write to the same register with a different encoding.

Where an indirect write is caused by the action of an external agent, such as a debugger, or by a memory-mapped read or write by the PE, then an indirect write by that agent to a register using a particular access mechanism, followed by an indirect read by that agent to the same register using the same access mechanism and address does not need synchronization.

For information about the additional synchronization requirements for memory-mapped registers, see *Synchronization requirements for System registers* on page D7-1908.

To guarantee the visibility of changes to some registers, additional operations might be required before the context synchronization operation. For such a register, the definition of the register identifies these additional requirements.

In this manual, unless the context indicates otherwise:

- *Accessing* a System register refers to a direct read or write of the register.
- *Using* a System register refers to an indirect read or write of the register.

### Registers with some architectural guarantee of ordering or observability

For the registers for which Table G4-45 on page G4-4135 shows that the ordering of direct reads is guaranteed, multiple direct reads of a single register, using the same encoding, occur in program order without any explicit ordering.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the registers for which Table G4-45 shows that some observability of indirect writes is guaranteed, an indirect write to the register caused by an external agent, an autonomous asynchronous event, or as a result of a memory-mapped write, is both:

- Observable to direct reads of the register, in finite time, without explicit synchronization.
- Observable to subsequent indirect reads of the register without explicit synchronization.

These two sets of registers are similar, as Table G4-45 shows:

**Table G4-45 Registers with a guarantee of ordering or observability, VMSAv8-32**

| Register | Ordering of direct reads | Observability of indirect writes | Notes |
|---|---|---|---|
| ISR | Guaranteed | Guaranteed | Interrupt Status Register |
| DBGCLAIMCLR | Guaranteed | Guaranteed | Debug CLAIM registers |
| DBGCLAIMSET | - | Guaranteed | |
| DBGDTRRX | Guaranteed | Guaranteed | Debug Communication Channel registers |
| DBGDTRTX | Guaranteed | Guaranteed | |
| CNTPCT | Guaranteed | Guaranteed | Generic Timer registers |
| CNTP_TVAL | Guaranteed | Guaranteed | |
| CNTVCT | Guaranteed | Guaranteed | |
| CNTV_TVAL | Guaranteed | Guaranteed | |
| CNTHP_TVAL | Guaranteed | Guaranteed | |
| PMCCNTR | Guaranteed | Guaranteed | Performance Monitors Extension registers, if the implementation includes the extension |
| PMXEVCNTR | Guaranteed | Guaranteed | |
| PMOVSSET | Guaranteed | Guaranteed | |

For the specified registers, the observability requirement is more demanding than the observability requirements for other registers. However, the possibility that direct reads can occur *early*, in the absence of context synchronization, described in *Ordering of reads of System registers* on page G4-4116, still applies to these registers.

In Debug state, additional synchronization requirements can apply to the registers shown in Table G4-45. For more information, see *Synchronization of DCC and ITR accesses* on page H4-4897.

### Definitions of direct and indirect reads and writes and their side-effects

Direct and indirect reads and writes are defined as follows:

**Direct read** Is a read of a register, using an `MRC`, `MRC2`, `MRRC`, `MRRC2`, `LDC`, or `LDC2` instruction, that the architecture permits for the current PE state.

If a direct read of a register has a side-effect of changing the value of a register, the effect of a direct read on that register is defined to be an *indirect write*, and has the synchronization requirements of an indirect write. This means the indirect write is guaranteed to have occurred, and to be visible to subsequent direct or indirect reads and writes only if synchronization is performed after the direct read.

**Note**

The indirect write described here can affect either the register written to by the direct write, or some other register. The synchronization requirement is the same in both cases.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Direct write** Is a write to a register, using an `MCR`, `MCR2`, `MCRR`, `MCRR2`, `STC`, or `STC2` instruction, that the architecture permits for the current PE state.

In the following cases, the side-effect of the direct write is defined to be an indirect write of the affected register, and has the synchronization requirements of an indirect write:

- If the direct write has a side-effect of changing the value of a register other than the register accessed by the direct write.
- If the direct write has a side-effect of changing the value of the register accessed by the direct write, so that the value in that register might not be the value that the direct write wrote to the register.

In both cases, this means that the indirect write is not guaranteed to be visible to subsequent direct or indirect reads and writes unless synchronization is performed after the direct write.

**Note**

- As an example of a direct write to a register having an effect that is an indirect write of that register, writing 1 to a PMCNTENCLR.P*x* bit is also an indirect write, because if the P*x* bit had the value 1 before the direct write, the side-effect of the write changes the value of that bit to 0.
- The indirect write described here can affect either the register written to by the direct write, or some other register. The synchronization requirement is the same in both cases.

  For example, writing 1 to a PMCNTENCLR.P*x* bit that is set to 1 also changes the corresponding PMCNTENSET.P*x* bit from 1 to 0. This means that the direct write to the PMCNTENCLR defines indirect writes to both itself and to the PMCNTENSET.

**Indirect read** Is a use of the register by an instruction to establish the operating conditions for the instruction. Examples of operating conditions that might be determined by an indirect read are the translation table base address, or whether a cache is enabled.

Indirect reads include situations where the value of one register determines what value is returned by a second register. This means that any read of the second register is an indirect read of the register that determines what value is returned.

Indirect reads also include:

- Reads of the System registers by external agents, such as debuggers, as described in *Debug registers* on page G6-4648.
- Memory-mapped reads of the System registers made by the PE on which the System registers are implemented.

Where an indirect read of a register has a side-effect of changing the value of a register, that change is defined to be an indirect write, and has the synchronization requirements of an indirect write.

**Indirect write** Is an update to the value of a register as a consequence of either:

- An exception, operation, or execution of an instruction that is not a direct write to that register.
- The asynchronous operation of an external agent.

This can include:

- The passage of time, as seen in counters or timers, including performance counters.
- The assertion of an interrupt.
- A write from an external agent, such as a debugger.

However, for some registers, the architecture gives some guarantee of visibility without any explicit synchronization, see *Registers with some architectural guarantee of ordering or observability* on page G4-4134.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Taking an exception is a context-synchronizing operation. Therefore, any indirect write performed as part of an exception entry does not require additional synchronization. This includes the indirect writes to the registers that report the exception, as described in *Exception reporting in a VMSAv8-32 implementation* on page G4-4091.

### G4.15.5 Meaning of fixed bit values in register diagrams

In register diagrams, fixed bits are indicated by one of following:

**0** In any implementation:

- The bit must read as 0.
- Writes to the bit must be ignored.
- Software:
  - Can rely on the bit reading as 0.
  - Must use an SBZP policy to write to the bit.

**(0)** In AArch32 state there are a small number of cases where a bit is (0) in some contexts, and has a different defined behavior in other contexts. See RES0.

**1** In any implementation:

- The bit must read as 1.
- Writes to the bit must be ignored.
- Software:
  - Can rely on the bit reading as 1.
  - Must use an SBOP policy to write to the bit.

**(1)** In AArch32 state there are a small number of cases where a bit is (1) in some contexts, and has a different defined behavior in other contexts. See RES1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.16 Organization of the CP14 registers in VMSAv8-32

The CP14 registers provide a number of distinct control functions, covering:

- Debug.
- Trace.
- Execution environment control, for identification of the trivial Jazelle implementation.

Because these functions are distinct, the descriptions of these registers are distributed, as follows:

- In this manual *Debug registers* on page G6-4648 describes the Debug registers.
- The *Embedded Trace Macrocell Architecture Specification* describes the Trace registers.

This section summarizes the allocation of the CP14 registers between these different functions, and the CP14 register encodings that are reserved.

The CP14 32-bit register encodings are classified by the {opc1, CRn, opc2, CRm} values required to access them using an `MCR` or an `MRC` instruction. The CP14 64-bit register encodings are classified by the {opc1, CRm} values required to access them using an `MCRR` or an `MRRC` instruction. The opc1 value determines the primary allocation of these registers, as follows:

**opc1==0** Debug registers.

**opc1==1** Trace registers.

**opc1==7** Jazelle registers. Jazelle registers are implemented as required for a trivial Jazelle implementation.

**Other opc1 values**

Reserved.

**Note**

Primary allocation of CP14 register function by opc1 value differs from the allocation of CP15 registers, where primary allocation is by CRn value.

For the Debug and Jazelle registers, Table G4-46 on page G4-4139 defines:

- The {opc1, CRn, opc2, CRm} values used for accessing the 32-bit registers using the `MRC` and `MCR` instructions.
- The {opc1, CRm} values used for accessing the 64-bit register using the `MRRC` instruction.

Some Debug registers can also be accessed using the `LDC` and `STC` instructions. Table G4-47 on page G4-4140 defines the CRn values used for accessing the registers using these instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.16.1 CP14 interface instruction arguments

Table G4-46 shows the MCR, MRC, and MRRC instruction arguments required for accesses to each register that can be visible in the CP14 System Register interface.

**Table G4-46 Mapping of CP14 MCR, MRC, and MRRC instruction arguments to registers**

| opc1 | CRn | opc2 | CRm | Name | Width | Description |
|---|---|---|---|---|---|---|
| 0 | c0 | 0 | c0 | DBGDIDR | 32-bit | Debug ID, or unallocated[a] |
| | | | c1 | DBGDSCRint | 32-bit | Debug Status and Control Register, Internal View |
| | | | c2 | DBGDCCINT | 32-bit | DCC Interrupt Enable Register |
| | | | c5 | DBGDTRRXint | 32-bit | Debug Data Transfer Register, Receive, Internal View |
| | | | c5 | DBGDTRTXint | 32-bit | Debug Data Transfer Register, Transmit, Internal View |
| | | | c6 | - | 32-bit | Legacy DBGWFAR, RES0 |
| | | | c7 | DBGVCR | 32-bit | Debug Vector Catch Register |
| | | 2 | c0 | DBGDTRRXext | 32-bit | Debug Data Transfer Register, Receive, External View |
| | | | c2 | DBGDSCRext | 32-bit | Debug Status and Control Register, External View |
| | | | c3 | DBGDTRTXext | 32-bit | Debug Data Transfer Register, Transmit, External View |
| | | | c6 | DBGOSECCR | 32-bit | Debug OS Lock Exception Catch Register |
| | | 4 | c0-15[b] | DBGBVR<n> | 32-bit | Debug Breakpoint Value Registers, n = 0-15 |
| | | 5 | c0-15[b] | DBGBCR<n> | 32-bit | Debug Breakpoint Control Registers, n = 0-15 |
| | | 6 | c0-15[b] | DBGWVR<n> | 32-bit | Debug Watchpoint Value Registers, n =0-15 |
| | | 7 | c0-15[b] | DBGWCR<n> | 32-bit | Debug Watchpoint Control Registers, n = 0-15 |
| | c1 | 0 | c0 | DBGDRAR | 32-bit | Debug ROM Address Register |
| | - | - | c1 | | 64-bit | |
| | c1 | 1 | c0-15[b] | DBGBXVR<n> | 32-bit | Debug Breakpoint Extended Value Registers n = 0-15 |
| | | 4 | c0 | DBGOSLAR | 32-bit | Debug OS Lock Access Register |
| | | | c1 | DBGOSLSR | 32-bit | Debug OS Lock Status Register |
| | | | c3 | DBGOSDLR | 32-bit | Debug OS Double Lock Register |
| | | | c4 | DBGPRCR | 32-bit | Debug Power Control Register |
| | c2 | 0 | c0 | DBGDSAR | 32-bit | Debug Self Address Register or unallocated[a] |
| | - | - | c2 | | 64-bit | |
| | c4 | 0-3 | c0-15 | - | | IMPLEMENTATION DEFINED |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-46 Mapping of CP14 MCR, MRC, and MRRC instruction arguments to registers (continued)**

| opc1 | CRn | opc2 | CRm | Name | Width | Description |
|---|---|---|---|---|---|---|
| 0 | c7 | 6 | c8 | DBGCLAIMSET | 32-bit | Debug CLAIM Tag Set register |
| | | | c9 | DBGCLAIMCLR | 32-bit | Debug CLAIM Tag Clear register |
| | | | c14 | DBGAUTHSTATUS | 32-bit | Debug Authentication Status register |
| | | 7 | c0 | DBGDEVID2 | 32-bit | Debug Device ID register 2 |
| | | | c1 | DBGDEVID1 | 32-bit | Debug Device ID register 1 |
| | | | c2 | DBGDEVID | 32-bit | Debug Device ID register |
| 1 | c0-c7 | 0-7 | c0-c15 | - | 32-bit | Reserved for OPTIONAL Trace extension |
| 7 | c0 | 0 | c0 | JIDR | 32-bit | Jazelle ID Register[c] |
| | c1 | 0 | c0 | JOSCR | 32-bit | Jazelle OS Control Register[c] |
| | c2 | 0 | c0 | JMCR | 32-bit | Jazelle Main Configuration Register[c] |
| All other encodings | | | | - | 32-bit | Unallocated |

a. If EL1 cannot use AArch32 this register is OPTIONAL and deprecated. See the register description for details.

b. Not implemented breakpoint and watchpoint register access instructions are unallocated. If EL2 is not implemented or breakpoint *n* is not context-aware, DBGBXVR<n> is unallocated. CRm encodes <*n*>, the breakpoint or watchpoint number.

c. Legacy register, see *Legacy feature registers, functional group* on page G4-4179.

Table G4-47 shows the LDC and STC instruction arguments required for accesses to each register that can be visible in the CP14 interface.

**Table G4-47 Mapping of CP14 LDC and STC instruction arguments to registers**

| CRn | Instruction | Name | Width | Description |
|---|---|---|---|---|
| c5 | LDC | DBGDTRTXint | 32-bit | Debug Data Transfer Register, Transmit, Internal View |
| c5 | STC | DBGDTRRXint | 32-bit | Debug Data Transfer Register, Receive, Internal View |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.17 Organization of the CP15 registers in VMSAv8-32

Previous documentation has described the CP15 registers in order of their primary coprocessor register number. More precisely, the ordered set of values {CRn, opc1, CRm, opc2} determined the register order. As the number of System registers has increased this ordering has become less appropriate. Also, it applies only to 32-bit registers, since 64-bit registers are identified only by {CRm, opc1}, making it difficult to include 32-bit and 64-bit versions of a single register in a common ordering scheme.

This document now:

- Groups the CP15 registers by functional group. For more information about this grouping in VMSAv8-32, including a summary of each functional group, see *Functional grouping of VMSAv8-32 System registers* on page G4-4161.
- Describes all of the System registers for VMSAv8-32, including the CP15 registers, in Chapter G6 *AArch32 System Register Descriptions*.

This section gives additional information about the organization of the CP15 registers in VMSAv8-32, as follows:

**Register ordering by {CRn, opc1, CRm, opc2}**

See:

- *CP15 32-bit register summary by coprocessor register number, CRn* on page G4-4142.
- *Full list of VMSAv8-32 CP15 registers, by coprocessor register number* on page G4-4147.

**Note**

The ordered listing of CP15 registers by the {CRn, opc1, CRm, opc2} encoding of the 32-bit registers is most likely to be useful to those implementing AArch32 state, and to those validating such implementations. However, otherwise, the grouping of registers by function is more logical.

**Views of the registers, that depend on the current state of the PE**

See *AArch32 views of the CP15 registers* on page G4-4158.

**Note**

The different register views are particularly significant in implementations that include EL2.

In addition, the indexes in Appendix K13 *Registers Index* include all of the CP15 registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.17.1 CP15 32-bit register summary by coprocessor register number, CRn

Figure G4-21 summarizes the grouping of CP15 registers by primary coprocessor register number for a VMSAv8-32 implementation.

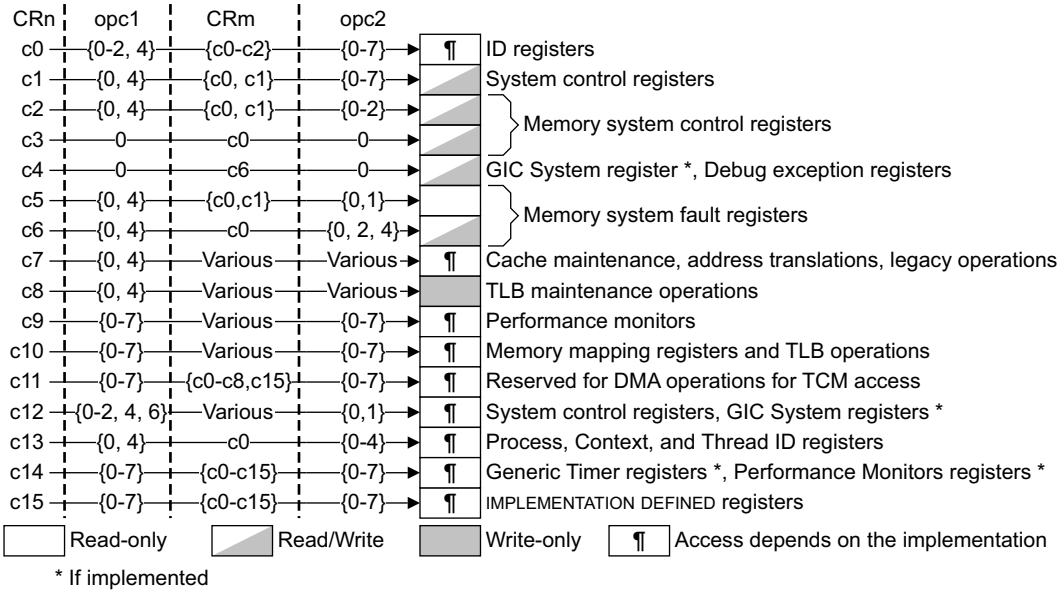



**Figure G4-21 CP15 32-bit register grouping by primary coprocessor register, CRn**

**Note**

- Figure G4-21 gives only an overview of the assigned encodings for 32-bit registers for each of the CP15 primary registers c0-c15. See the description of each primary register for the definition of the assigned and unassigned encodings for that register, including any dependencies on the implemented Exception levels.
- 64-bit registers in the CP15 encoding space use the same primary coprocessor register model, but in the 64-bit register read and write instructions, `MRRC` and `MCRR`, CRm identifies the primary coprocessor register.

The following sections give the register assignments for each of the CP15 primary registers, c0-c15:

- *VMSAv8-32 CP15 c0 register summary* on page G4-4143.
- *VMSAv8-32 CP15 c1 register summary* on page G4-4143.
- *VMSAv8-32 CP15 c2 and c3 register summary* on page G4-4143
- *VMSAv8-32 CP15 c4 register summary* on page G4-4143.
- *VMSAv8-32 CP15 c5 and c6 register summary* on page G4-4144.
- *VMSAv8-32 CP15 c7 register summary* on page G4-4144.
- *VMSAv8-32 CP15 c8 register summary* on page G4-4144.
- *VMSAv8-32 CP15 c9 register summary* on page G4-4144.
- *VMSAv8-32 CP15 c10 register summary* on page G4-4145.
- *VMSAv8-32 CP15 c11 register summary* on page G4-4145.
- *VMSAv8-32 CP15 c12 register summary* on page G4-4146.
- *VMSAv8-32 CP15 c13 register summary* on page G4-4146.
- *VMSAv8-32 CP15 c14 register summary* on page G4-4146.
- *VMSAv8-32 CP15 c15 register summary* on page G4-4147.

As *General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to CP15 System registers* on page G1-3868 describes, when the value of HSTR.T*n* is 1, Non-secure PL1 accesses to CP15 c*n* encodings are trapped to PL2, even if the encoding is UNDEFINED when the value of HSTR.T*n* is 0. If there are CP15 c*n* encodings that are accessible from Non-secure PL0 then those accesses are also trapped to PL2 when the value of HSTR.T*n* is 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### VMSAv8-32 CP15 c0 register summary

The CP15 c0 registers provide device and feature identification. That is, they provide the register functional group described in *Identification registers, functional group* on page G4-4162.

Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c0 registers. The behavior of CP15 c0 register encodings not shown in the table, and encodings that are part of an unimplemented Exception level, depends on the value of opc 1, and possibly on the value of CRm and opc2, as follows:

**opc1 == 0** All write accesses to the encodings are UNDEFINED.

For read accesses:

- The following encodings return an UNKNOWN value:
  - CRm == 3, opc2 == {0, 1, 2}.
  - CRm == {4, 6, 7}, opc2 == {0, 1}.
  - CRm == 5, opc2 == {0, 1, 4, 5}.
- All other encodings are RES0.

**opc1 > 0** All accesses to the encodings are UNDEFINED.

See also *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

**Note**

Some of these registers were previously described as being part of the CPUID identification scheme, see *The CPUID identification scheme* on page G4-4163.

### VMSAv8-32 CP15 c1 register summary

The CP15 c1 registers provide system control, including security and virtualization control. That is, they provide registers from the functional groups described in the following sections:

- *Other System registers, functional group* on page G4-4163.
- *Virtualization registers, functional group* on page G4-4164.
- *Security registers, functional group* on page G4-4167.

Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c1 registers. CP15 c1 register encodings not shown in the table, and encodings that are part of an unimplemented Exception level, are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

### VMSAv8-32 CP15 c2 and c3 register summary

The CP15 c2 and c3 registers provide memory system control. That is, they provide registers from the functional group described in the section *Virtual memory control registers, functional group* on page G4-4163.

Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c2 and c3 registers. CP15 c2 and c3 register encodings not shown in the table, and encodings that are part of an unimplemented Exception level, are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

### VMSAv8-32 CP15 c4 register summary

In an ARMv8 implementation, the CP15 c4 encoding space provides:

- In an implementation that includes the System register interface to an implementation of the ARM Generic Interrupt Control (GIC), two registers for this interface, the ICC_PMR and ICV_PMR. These are registers from the functional groups described in the section *Generic Interrupt Controller System registers, functional groups* on page G4-4175.
- Two registers that are used for handling Debug exceptions, the DLR and the DSPSR.

Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c4 registers. CP15 c4 register encodings not shown in the table are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### VMSAv8-32 CP15 c5 and c6 register summary

The CP15 c5 and c6 registers provide exception and fault handling. That is, they provide registers from the functional group described in the section *Exception and fault handling registers, functional group* on page G4-4168.

Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c5 and c6 registers. CP15 c5 and c6 register encodings not shown in the table, and encodings that are part of an unimplemented Exception level are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

### VMSAv8-32 CP15 c7 register summary

The CP15 c7 registers provide system operations for cache maintenance, address translation, and some legacy operations. That is, they provide registers from the functional groups described in the following sections:

- *Cache maintenance instructions, functional group* on page G4-4169.
- *Address translation instructions, functional group* on page G4-4172.
- *Legacy feature registers, functional group* on page G4-4179.

Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c7 registers. CP15 c7 register encodings not shown in the table, and encodings that are part of an unimplemented Exception level are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

### VMSAv8-32 CP15 c8 register summary

The CP15 c8 registers provide operations for TLB maintenance. That is, they provide registers from the functional groups described in *TLB maintenance instructions, functional group* on page G4-4171.

Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c8 registers. CP15 c8 register encodings not shown in the table, and encodings that are part of an unimplemented Exception level are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

### VMSAv8-32 CP15 c9 register summary

The CP15 c9 registers provide:

- Registers for the OPTIONAL Performance Monitors Extension. That is, they provide registers from the functional group described in *Performance Monitors Extension registers, functional group* on page G4-4173.
- Reserved encodings for IMPLEMENTATION DEFINED memory system functions, in particular:
  - Cache control, including lockdown.
  - TCM control, including lockdown.
  - Branch predictor control.

  **Note**

  The reserved encodings permit implementations that are compatible with previous versions of the ARM architecture, in particular with the ARMv6 requirements.
- Reserved encodings for additional IMPLEMENTATION DEFINED performance monitors.

Figure G4-22 shows the VMSAv8-32 allocation of CP15 c9 register encodings.

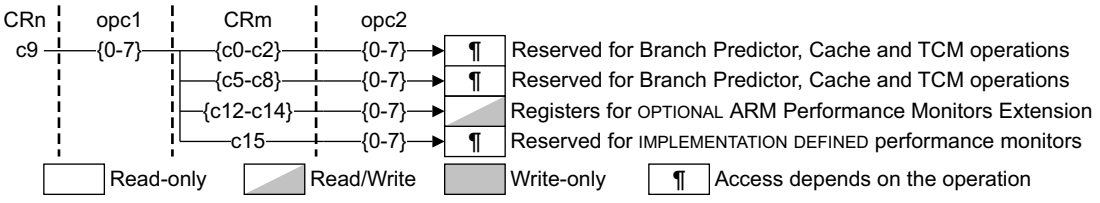



**Figure G4-22 VMSAv8-32 CP15 c9 32-bit register encodings**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

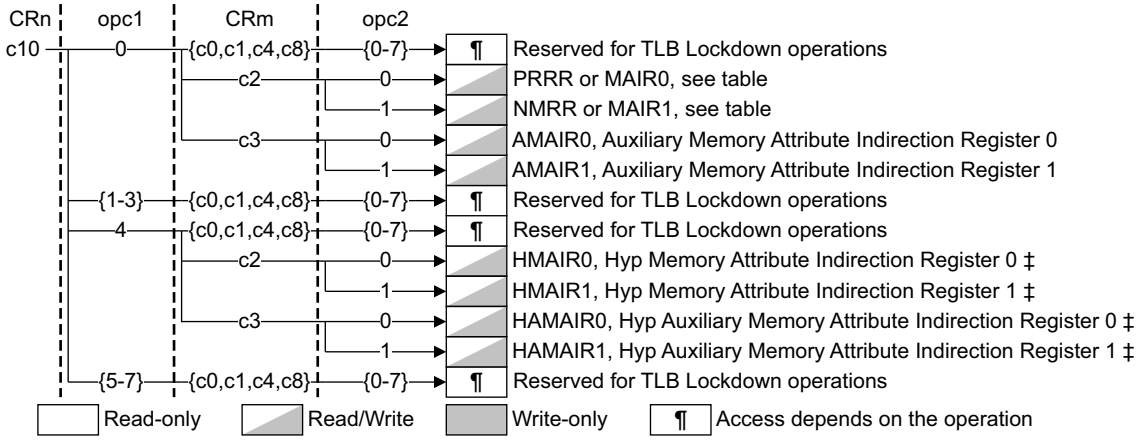
Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c9 registers. Unimplemented CP15 c9 register encodings, including encodings that are part of an unimplemented Exception level, are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

## VMSAv8-32 CP15 c10 register summary

The CP15 c10 registers provide:

- Virtual memory control registers. That is, they provide registers from the functional group described in *Virtual memory control registers, functional group* on page G4-4163.
- Reserved encodings for IMPLEMENTATION DEFINED TLB control functions, including lockdown.

Figure G4-23 shows the VMSAv8-32 allocation of CP15 c10 registers and reserved encodings.




‡ Implemented only as part of EL2

| When using Short-descriptor translation table format | When using Long-descriptor translation table format |
|---|---|
| PRRR, Primary Region Remap Register | MAIR0, Memory Attribute Indirection Register 0 |
| NMRR, Normal Memory Remap Register | MAIR1, Memory Attribute Indirection Register 1 |

**Figure G4-23 VMSAv8-32 CP15 c10 32-bit register encodings**

Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c10 registers. Unimplemented CP15 c10 register encodings, including encodings that are part of an unimplemented Exception level, are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

## VMSAv8-32 CP15 c11 register summary

The CP15 c11 registers provide some reserved encodings for IMPLEMENTATION DEFINED DMA operations to and from TCM. Figure G4-24 shows these reserved encodings:

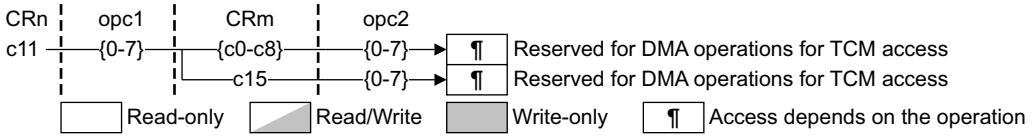



**Figure G4-24 VMSAv8-32 reserved CP15 c11 encodings**

CP15 c11 encodings not shown in Figure G4-24 are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### VMSAv8-32 CP15 c12 register summary

The CP15 c12 registers provide registers from the functional groups described in the following sections:

- *Exception and fault handling registers, functional group* on page G4-4168.
- *Virtualization registers, functional group* on page G4-4164.
- *Security registers, functional group* on page G4-4167.
- *Reset management registers, functional group* on page G4-4169.
- *Generic Interrupt Controller System registers, functional groups* on page G4-4175.

Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c12 registers. CP15 c12 register encodings not shown in the table, and encodings that are part of an unimplemented Exception level are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

**Note**

Some CP15 c12 registers are in more than one functional group.

### VMSAv8-32 CP15 c13 register summary

The CP15 c13 registers provide:

- An FCSE Process ID Register, that indicates that ARMv8 implementations do not include the FCSE.
- A Context ID Register.
- Software Thread ID Registers.

These registers are from the functional groups described in the following sections:

- *Virtual memory control registers, functional group* on page G4-4163.
- *Virtualization registers, functional group* on page G4-4164.
- *Thread and process ID registers, functional group* on page G4-4169.
- *Legacy feature registers, functional group* on page G4-4179.

**Note**

Some CP15 c12 registers are in more than one functional group.

Figure G4-25 shows these registers:

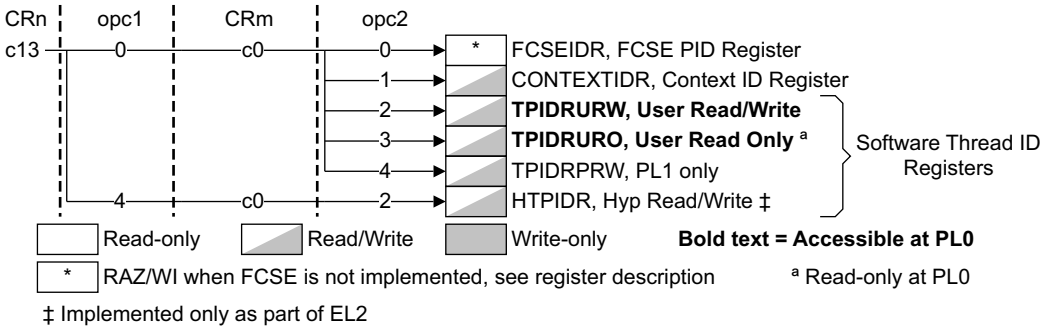



**Figure G4-25 CP15 c13 registers in VMSAv8-32**

Table G4-48 on page G4-4147 shows all of the architecturally required CP15 c13 registers. CP15 c13 register encodings not shown in the table, and encodings that are part of an unimplemented Exception level are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

### VMSAv8-32 CP15 c14 register summary

CP15 c14 is reserved for the System registers of the Generic Timer. For more information, see Chapter D6 *The Generic Timer in AArch64 state*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an implementation that includes the Generic Timer:

- The CP15 c14 registers provide Performance Monitors Extension and Generic Timer registers. That is, they provide registers from the functional group described in the following sections:
  - — *Performance Monitors Extension registers, functional group* on page G4-4173.
  - — *Generic Timer registers, functional group* on page G4-4174
- Table G4-48 shows all of the architecturally required CP15 c14 registers. CP15 c14 register encodings not shown in the table, and encodings that are part of an unimplemented Exception level are UNDEFINED, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

**Note**

Some CP15 c14 registers are also in the Virtualization group, see *Virtualization registers, functional group* on page G4-4164.

### VMSAv8-32 CP15 c15 register summary

AArch32 state reserves CP15 c15 for IMPLEMENTATION DEFINED purposes, and does not impose any restrictions on the use of the CP15 c15 encodings. The documentation of the ARM implementation must describe fully any registers implemented in CP15 c15. Normally, for processor implementations by ARM, this information is included in the *Technical Reference Manual* for the processor.

Typically, an implementation uses CP15 c15 to provide test features, and any required configuration options that are not covered by this manual.

## G4.17.2 Full list of VMSAv8-32 CP15 registers, by coprocessor register number

Table G4-48 shows the CP15 registers in VMSAv8-32, in the order of the {CRn, opc1, CRm, opc2} values used in `MCR` or `MRC` accesses to the 32-bit registers:

- For `MCR` or `MRC` accesses to the 32-bit registers, CRn identifies the CP15 primary register used for the access.
- For `MCRR` or `MRRC` accesses to the 64-bit registers, CRm identifies the CP15 primary register used for the access. Table G4-48 lists the 64-bit registers with the 32-bit registers accessed using the same CP15 primary register number.

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c0 | 0 | c0 | 0 | MIDR | 32-bit | Main ID Register |
| | | | 1 | CTR | 32-bit | Cache Type Register |
| | | | 2 | TCMTR | 32-bit | TCM Type Register |
| | | | 3 | TLBTR | 32-bit | TLB Type Register |
| | | | 4, 6[a], 7 | MIDR | 32-bit | Aliases of Main ID Register |
| | | | 5 | MPIDR | 32-bit | Multiprocessor Affinity Register |
| | | | 6[a] | REVIDR | 32-bit | Revision ID Register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order (continued)**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c0 | 0 | c1 | 0 | ID_PFR0 | 32-bit | Processor Feature Register 0 |
| | | | 1 | ID_PFR1 | 32-bit | Processor Feature Register 1 |
| | | | 2 | ID_DFR0 | 32-bit | Debug Feature Register 0 |
| | | | 3 | ID_AFR0 | 32-bit | Auxiliary Feature Register 0 |
| | | | 4 | ID_MMFR0 | 32-bit | Memory Model Feature Register 0 |
| | | | 5 | ID_MMFR1 | 32-bit | Memory Model Feature Register 1 |
| | | | 6 | ID_MMFR2 | 32-bit | Memory Model Feature Register 2 |
| | | | 7 | ID_MMFR3 | 32-bit | Memory Model Feature Register 3 |
| | | c2 | 0 | ID_ISAR0 | 32-bit | Instruction Set Attribute Register 0 |
| | | | 1 | ID_ISAR1 | 32-bit | Instruction Set Attribute Register 1 |
| | | | 2 | ID_ISAR2 | 32-bit | Instruction Set Attribute Register 2 |
| | | | 3 | ID_ISAR3 | 32-bit | Instruction Set Attribute Register 3 |
| | | | 4 | ID_ISAR4 | 32-bit | Instruction Set Attribute Register 4 |
| | | | 5 | ID_ISAR5 | 32-bit | Instruction Set Attribute Register 5 |
| | | | 6 | ID_MMFR4 | 32-bit | Memory Model Feature Register 4 |
| | 1 | c0 | 0 | CCSIDR | 32-bit | Cache Size ID Registers |
| | | | 1 | CLIDR | 32-bit | Cache Level ID Register |
| | | | 7 | AIDR | 32-bit | Auxiliary ID Register |
| c0 | 2 | c0 | 0 | CSSELR | 32-bit | Cache Size Selection Register |
| | 4 | c0 | 0 | VPIDR[b] | 32-bit | Virtualization Processor ID Register |
| | | | 5 | VMPIDR[b] | 32-bit | Virtualization Multiprocessor ID Register |
| c1 | 0 | c0 | 0 | SCTLR | 32-bit | System Control Register |
| | | | 1 | ACTLR | 32-bit | Auxiliary Control Register |
| | | | 2 | CPACR | 32-bit | Coprocessor Access Control Register |
| | | | 3 | ACTLR2 | 32-bit | Auxiliary Control Register 2 |
| | | c1 | 0 | SCR[c] | 32-bit | Secure Configuration Register |
| | | | 1 | SDER[c] | 32-bit | Secure Debug Enable Register |
| | | | 2 | NSACR[c] | 32-bit | Non-Secure Access Control Register |
| | | c3 | 1 | SDCR[d] | 32-bit | Secure Debug Configuration Register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order (continued)**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c1 | 4 | c0 | 0 | HSCTLR[b] | 32-bit | Hyp System Control Register |
| | | | 1 | HACTLR[b] | 32-bit | Hyp Auxiliary Control Register |
| | | | 3 | HACTLR2[b] | 32-bit | Hyp Auxiliary Control Register 2 |
| | | c1 | 0 | HCR[b] | 32-bit | Hyp Configuration Register |
| | | | 1 | HDCR[b] | 32-bit | Hyp Debug Configuration Register |
| | | | 2 | HCPTR[b] | 32-bit | Hyp Coprocessor Trap Register |
| | | | 3 | HSTR[b] | 32-bit | Hyp System Trap Register |
| | | | 4 | HCR2[b, d] | 32-bit | Hyp Configuration Register 2 |
| | | | 7 | HACR[b] | 32-bit | Hyp Auxiliary Configuration Register |
| c2 | 0 | c0 | 0 | TTBR0 | 32-bit | Translation Table Base Register 0 |
| - | 0 | c2 | - | TTBR0 | 64-bit | |
| c2 | 0 | c0 | 1 | TTBR1 | 32-bit | Translation Table Base Register 1 |
| - | 1 | c2 | - | TTBR1 | 64-bit | |
| c2 | 0 | c0 | 2 | TTBCR | 32-bit | Translation Table Base Control Register |
| | 4 | c0 | 2 | HTCR[b] | 32-bit | Hyp Translation Control Register |
| | | c1 | 2 | VTCR[b] | 32-bit | Virtualization Translation Control Register |
| - | 4 | c2 | - | HTTBR[b] | 64-bit | Hyp Translation Table Base Register |
| - | 6 | c2 | - | VTTBR[b] | 64-bit | Virtualization Translation Table Base Register |
| c3 | 0 | c0 | 0 | DACR | 32-bit | Domain Access Control Register |
| c4 | 0 | c6 | 0 | ICC_PMR[e]<br>ICV_PMR[e] | 32-bit | Interrupt Controller Interrupt Priority Mask Register<br>Interrupt Controller Virtual Priority Mask Register |
| | 3 | c5 | 0 | DSPSR[d] | 32-bit | Debug Saved Program Status Register[f] |
| | | | 1 | DLR[d] | 32-bit | Debug Link Register[f] |
| c5 | 0 | c0 | 0 | DFSR | 32-bit | Data Fault Status Register |
| | | | 1 | IFSR | 32-bit | Instruction Fault Status Register |
| | | c1 | 0 | ADFSR | 32-bit | Auxiliary Data Fault Status Register |
| | | | 1 | AIFSR | 32-bit | Auxiliary Instruction Fault Status Register |
| c5 | 4 | c1 | 0 | HADFSR[b] | 32-bit | Hyp Auxiliary Data Fault Syndrome Register |
| | | | 1 | HAIFSR | 32-bit | Hyp Auxiliary Instruction Fault Syndrome Register |
| | | c2 | 0 | HSR[b] | 32-bit | Hyp Syndrome Register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order (continued)**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c6 | 0 | c0 | 0 | DFAR | 32-bit | Data Fault Address Register |
| | | | 2 | IFAR | 32-bit | Instruction Fault Address Register |
| | 4 | c0 | 0 | HDFAR[b] | 32-bit | Hyp Data Fault Address Register |
| | | | 2 | HIFAR[b] | 32-bit | Hyp Instruction Fault Address Register |
| | | | 4 | HPFAR[b] | 32-bit | Hyp IPA Fault Address Register |
| c7 | 0 | c1 | 0 | ICIALLUIS | 32-bit | See *Cache maintenance instructions, functional group* on page G4-4169 |
| | | | 6 | BPIALLIS | 32-bit | |
| | | c4 | 0 | PAR | 32-bit | Physical Address Register |
| - | 0 | c7 | - | PAR | 64-bit | |
| c7 | 0 | c5 | 0 | ICIALLU | 32-bit | See *Cache maintenance instructions, functional group* on page G4-4169 |
| | | | 1 | ICIMVAU | 32-bit | |
| | | | 4 | CP15ISB | 32-bit | See *Memory barriers* on page E2-2375 |
| | | | 6 | BPIALL | 32-bit | See *Cache maintenance instructions, functional group* on page G4-4169 |
| | | | 7 | BPIMVA | 32-bit | |
| | | c6 | 1 | DCIMVAC | 32-bit | See *Cache maintenance instructions, functional group* on page G4-4169 |
| | | | 2 | DCISW | 32-bit | |
| c7 | 0 | c8 | 0 | ATS1CPR | 32-bit | See *Address translation instructions* on page G4-4110 |
| | | | 1 | ATS1CPW | 32-bit | |
| | | | 2 | ATS1CUR | 32-bit | |
| | | | 3 | ATS1CUW | 32-bit | |
| | | | 4 | ATS12NSOPR[c] | 32-bit | |
| | | | 5 | ATS12NSOPW[c] | 32-bit | |
| | | | 6 | ATS12NSOUR[c] | 32-bit | |
| | | | 7 | ATS12NSOUW[c] | 32-bit | |
| c7 | 0 | c10 | 1 | DCCMVAC | 32-bit | See *Cache maintenance instructions, functional group* on page G4-4169 |
| | | | 2 | DCCSW | 32-bit | |
| | | | 4 | CP15DSB | 32-bit | See *Memory barriers* on page E2-2375 |
| | | | 5 | CP15DMB | 32-bit | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order (continued)**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c7 | 0 | c11 | 1 | DCCMVAU | 32-bit | See *Cache maintenance instructions, functional group* on page G4-4169 |
| | | c14 | 1 | DCCIMVAC | 32-bit | See *Cache maintenance instructions, functional group* on page G4-4169 |
| | | | 2 | DCCISW | 32-bit | |
| | 4 | c8 | 0 | ATS1HR[b] | 32-bit | See *Address translation instructions* on page G4-4110 |
| | | | 1 | ATS1HW[b] | 32-bit | |
| c8 | 0 | c3 | 0 | TLBIALLIS | 32-bit | See *The scope of TLB maintenance instructions* on page G4-4069 |
| | | | 1 | TLBIMVAIS | 32-bit | |
| | | | 2 | TLBIASIDIS | 32-bit | |
| | | | 3 | TLBIMVAAIS | 32-bit | |
| | | | 5 | TLBIMVALIS[d] | 32-bit | |
| | | | 7 | TLBIMVAALIS[d] | 32-bit | |
| | | c5 | 0 | ITLBIALL | 32-bit | See *The scope of TLB maintenance instructions* on page G4-4069 |
| | | | 1 | ITLBIMVA | 32-bit | |
| | | | 2 | ITLBIASID | 32-bit | |
| | | c6 | 0 | DTLBIALL | 32-bit | See *The scope of TLB maintenance instructions* on page G4-4069 |
| | | | 1 | DTLBIMVA | 32-bit | |
| | | | 2 | DTLBIASID | 32-bit | |
| | | c7 | 0 | TLBIALL | 32-bit | See *The scope of TLB maintenance instructions* on page G4-4069 |
| | | | 1 | TLBIMVA | 32-bit | |
| | | | 2 | TLBIASID | 32-bit | |
| | | | 3 | TLBIMVAA | 32-bit | |
| | | | 5 | TLBIMVAL[d] | 32-bit | |
| | | | 7 | TLBIMVAAL[d] | 32-bit | |
| c8 | 4 | c0 | 1 | TLBIIPAS2IS[d] | 32-bit | See *The scope of TLB maintenance instructions* on page G4-4069 |
| | | | 5 | TLBIIPAS2LIS[d] | 32-bit | |
| c8 | 4 | c3 | 0 | TLBIALLHIS[b] | 32-bit | See *The scope of TLB maintenance instructions* on page G4-4069 |
| | | | 1 | TLBIMVAHIS[b] | 32-bit | |
| | | | 4 | TLBIALLNSNHIS[b] | 32-bit | |
| | | | 5 | TLBIMVALHIS[d] | 32-bit | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order (continued)**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c8 | 4 | c4 | 1 | TLBIIPAS2[d] | 32-bit | See *The scope of TLB maintenance instructions* on page G4-4069 |
| | | | 5 | TLBIIPAS2L[d] | 32-bit | |
| | | c7 | 0 | TLBIALLH[b] | 32-bit | See *The scope of TLB maintenance instructions* on page G4-4069 |
| | | | 1 | TLBIMVAH[b] | 32-bit | |
| | | | 4 | TLBIALLNSNH[b] | 32-bit | |
| | | | 5 | TLBIMVALH[d] | 32-bit | |
| c9 | 0 | c12 | 0 | PMCR | 32-bit | Performance Monitors Control Register |
| | | | 1 | PMCNTENSET | 32-bit | Performance Monitors Count Enable Set register |
| | | | 2 | PMCNTENCLR | 32-bit | Performance Monitors Count Enable Clear register |
| | | | 3 | PMOVSR | 32-bit | Performance Monitors Overflow Flag Status Register |
| | | | 4 | PMSWINC | 32-bit | Performance Monitors Software Increment register |
| | | | 5 | PMSELR | 32-bit | Performance Monitors Event Counter Selection Register |
| | | | 6 | PMCEID0 | 32-bit | Performance Monitors Common Event Identification register 0 |
| | | | 7 | PMCEID1 | 32-bit | Performance Monitors Common Event Identification register 1 |
| c9 | 0 | c13 | 0 | PMCCNTR | 32-bit | Performance Monitors Cycle Count Register |
| - | 0 | c9 | - | PMCCNTR_EL0 | 64-bit | Performance Monitors Cycle Count Register |
| c9 | 0 | c13 | 1 | PMXEVTYPER | 32-bit | Performance Monitors Event Type Select Register |
| | | | 2 | PMXEVCNTR | 32-bit | Performance Monitors Event Count Register |
| c9 | 0 | c14 | 0 | PMUSERENR | 32-bit | Performance Monitors User Enable Register |
| | | | 1 | PMINTENSET | 32-bit | Performance Monitors Interrupt Enable Set register |
| | | | 2 | PMINTENCLR | 32-bit | Performance Monitors Interrupt Enable Clear register |
| | | | 3 | PMOVSSET[b] | 32-bit | Performance Monitors Overflow Flag Status Set register |
| c10 | 0 | c2 | 0 | PRRR[g] | 32-bit | Primary Region Remap Register |
| | | | | MAIR0[g] | 32-bit | Memory Attribute Indirection Register 0 |
| | | | 1 | NMRR[g] | 32-bit | Normal Memory Remap Register |
| | | | | MAIR1[g] | 32-bit | Memory Attribute Indirection Register 1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order (continued)**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c10 | 0 | c3 | 0 | AMAIR0 | 32-bit | Auxiliary Memory Attribute Indirection Register 0 |
| | | | 1 | AMAIR1 | 32-bit | Auxiliary Memory Attribute Indirection Register 1 |
| | 4 | c2 | 0 | HMAIR0[b] | 32-bit | Hyp Memory Attribute Indirection Register 0 |
| | | | 1 | HMAIR1[b] | 32-bit | Hyp Memory Attribute Indirection Register 1 |
| | | c3 | 0 | HAMAIR0[b] | 32-bit | Hyp Auxiliary Memory Attribute Indirection Register 0 |
| | | | 1 | HAMAIR1[b] | 32-bit | Hyp Auxiliary Memory Attribute Indirection Register 1 |
| c11 | 0-7 | c0-c8 | 0-7 | - | 32-bit | See *VMSAv8-32 CP15 c11 register summary* on page G4-4145 |
| | | c15 | c15 | - | 32-bit | |
| - | 0 | c12 | - | ICC_SGI1R[e] | 64-bit | Interrupt Controller SGI group 1 Register |
| c12 | 0 | c0 | 0 | VBAR | 32-bit | Vector Base Address Register |
| | | | 1 | MVBAR[c] | 32-bit | Monitor Vector Base Address Register |
| | | | | RVBAR | 32-bit | Reset Vector Base Address Register |
| | | | 2 | RMR (at EL1)[h] | 32-bit | Reset Management Register, at EL1 |
| | | | | RMR (at EL3)[h] | 32-bit | Reset Management Register, at EL3 |
| | | c1 | 0 | ISR[c] | 32-bit | Interrupt Status Register |
| | | c8 | 0 | ICC_IAR0 [e]<br>ICV_IAR0 [e] | 32-bit | Interrupt Controller Interrupt Acknowledge Register 0<br>Interrupt Controller Virtual Acknowledge Register 0 |
| | | | 1 | ICC_EOIR0 [e]<br>ICV_EOIR0 [e] | 32-bit | Interrupt Controller End Of Interrupt Register 0<br>Interrupt Controller Virtual End Of Interrupt Register 0 |
| | | | 2 | ICC_HPPIR0 [e]<br>ICV_HPPIR0 [e] | 32-bit | Interrupt Controller Highest Priority Pending Interrupt Register 0<br>Interrupt Controller Virtual Highest Priority Pending Interrupt Register 0 |
| | | | 3 | ICC_BPR0 [e]<br>ICV_BPR0 [e] | 32-bit | Interrupt Controller Binary Point Register 0<br>Interrupt Controller Virtual Binary Point Register 0 |
| | | | 4 | ICC_AP0R0 [e]<br>ICV_AP0R0 [e] | 32-bit | Interrupt Controller Active Priorities Register (0,0)<br>Interrupt Controller Virtual Active Priorities Register (0,0) |
| | | | 5 | ICC_AP0R1 [e]<br>ICV_AP0R1 [e] | 32-bit | Interrupt Controller Active Priorities Register (0,1)<br>Interrupt Controller Virtual Active Priorities Register (0,1) |
| | | | 6 | ICC_AP0R1 [e]<br>ICV_AP0R1 [e] | 32-bit | Interrupt Controller Active Priorities Register (0,2)<br>Interrupt Controller Virtual Active Priorities Register (0,2) |
| | | | 7 | ICC_AP0R3 [e]<br>ICV_AP0R3 [e] | 32-bit | Interrupt Controller Active Priorities Register (0,3)<br>Interrupt Controller Virtual Active Priorities Register (0,3) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order (continued)**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c12 | 0 | c9 | 0 | ICC_AP1R0 [e]<br>ICV_AP1R0 [e] | 32-bit | Interrupt Controller Active Priorities Register (1,0)<br>Interrupt Controller Virtual Active Priorities Register (1,0) |
| | | | 1 | ICC_AP1R1 [e]<br>ICV_AP1R1 [e] | 32-bit | Interrupt Controller Active Priorities Register (1,1)<br>Interrupt Controller Virtual Active Priorities Register (1,1) |
| | | | 2 | ICC_AP1R2 [e]<br>ICV_AP1R2 [e] | 32-bit | Interrupt Controller Active Priorities Register (1,2)<br>Interrupt Controller Virtual Active Priorities Register (1,2) |
| | | | 3 | ICC_AP1R3 [e]<br>ICV_AP1R3 [e] | 32-bit | Interrupt Controller Active Priorities Register (1,3)<br>Interrupt Controller Virtual Active Priorities Register (1,3) |
| | | c11 | 1 | ICC_DIR [e]<br>ICV_DIR [e] | 32-bit | Interrupt Controller Deactivate Interrupt Register<br>Interrupt Controller Deactivate Virtual Interrupt Register |
| | | | 3 | ICC_RPR [e]<br>ICV_RPR [e] | 32-bit | Interrupt Controller Running Priority Register<br>Interrupt Controller Virtual Running Priority Register |
| c12 | 0 | c12 | 0 | ICC_IAR1 [e]<br>ICV_IAR1 [e] | 32-bit | Interrupt Controller Interrupt Acknowledge Register 1<br>Interrupt Controller Virtual Interrupt Acknowledge Register 1 |
| | | | 1 | ICC_EOIR1 [e]<br>ICV_EOIR1 [e] | 32-bit | Interrupt Controller End Of Interrupt Register 1<br>Interrupt Controller Virtual End Of Interrupt Register 1 |
| | | | 2 | ICC_HPPIR1 [e]<br>ICV_HPPIR1 [e] | 32-bit | Interrupt Controller Highest Priority Pending Interrupt Register 1<br>Interrupt Controller Virtual Highest Priority Pending Interrupt Register 1 |
| | | | 3 | ICC_BPR1 [e]<br>ICV_BPR1 [e] | 32-bit | Interrupt Controller Binary Point Register 1<br>Interrupt Controller Virtual Binary Point Register 1 |
| | | | 4 | ICC_CTLR [e]<br>ICV_CTLR [e] | 32-bit | Interrupt Controller Control Register<br>Interrupt Controller Virtual Control Register |
| | | | 5 | ICC_SRE [e] | 32-bit | Interrupt Controller System Register Enable Register |
| | | | 6 | ICC_IGRPEN0 [e]<br>ICV_IGRPEN0 [e] | 32-bit | Interrupt Controller Interrupt Group 0 Enable Register<br>Interrupt Controller Virtual Interrupt Group 0 Enable Register |
| | | | 7 | ICC_IGRPEN1 [e]<br>ICV_IGRPEN1 [e] | 32-bit | Interrupt Controller Interrupt Group 1 Enable Register<br>Interrupt Controller Virtual Interrupt Group 1 Enable Register |
| - | 1 | c12 | - | ICC_ASGI1R [e] | 64 bit | Interrupt Controller Alias SGI group 1 Register |
| - | 2 | c12 | - | ICC_SGI0R [e] | 64 bit | Interrupt Controller SGI group 0 Register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order (continued)**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c12 | 4 | c0 | 0 | HVBAR [b, c] | 32-bit | Hyp Vector Base Address Register |
| | | | 2 | HRMR [h] | 32-bit | Hyp Reset Management Register |
| | | c8 | 0 | ICH_AP0R0 [e] | 32-bit | Interrupt Controller Hyp Active Priorities Register (0,0) |
| | | | 1 | ICH_AP0R1 [e] | 32-bit | Interrupt Controller Hyp Active Priorities Register (0,1) |
| | | | 2 | ICH_AP0R2 [e] | 32-bit | Interrupt Controller Hyp Active Priorities Register (0,2) |
| | | | 3 | ICH_AP0R3 [e] | 32-bit | Interrupt Controller Hyp Active Priorities Register (0,3) |
| | | c9 | 0 | ICH_AP1R0 [e] | 32-bit | Interrupt Controller Hyp Active Priorities Register (1,0) |
| | | | 1 | ICH_AP1R1 [e] | 32-bit | Interrupt Controller Hyp Active Priorities Register (1,1) |
| | | | 2 | ICH_AP1R2 [e] | 32-bit | Interrupt Controller Hyp Active Priorities Register (1,2) |
| | | | 3 | ICH_AP1R3 [e] | 32-bit | Interrupt Controller Hyp Active Priorities Register (1,3) |
| | | | 5 | ICC_HSRE [e] | 32-bit | Interrupt Controller Hyp System Register Enable register |
| c12 | 4 | c11 | 0 | ICH_HCR [e] | 32-bit | Interrupt Controller Hyp Control Register |
| | | | 1 | ICH_VTR [e] | 32-bit | Interrupt Controller VGIC Type Register |
| | | | 2 | ICH_MISR [e] | 32-bit | Interrupt Controller Maintenance Interrupt State Register |
| | | | 3 | ICH_EISR [e] | 32-bit | Interrupt Controller End of Interrupt Status Register |
| | | | 5 | ICH_ELRSR [e] | 32-bit | Interrupt Controller Empty List Register Status Register |
| | | | 7 | ICH_VMCR [e] | 32-bit | Interrupt Controller Virtual Machine Control Register |
| c12 | 4 | c12 | 0-7 | ICH_LR<n>, for n==0 to 7 [e] | 32-bit | Interrupt Controller List Registers 0 to 7 |
| | | c13 | 0-7 | ICH_LR<n>, for n==8 to 15 [e] | 32-bit | Interrupt Controller List Registers 8 to 15 |
| | | c14 | 0-7 | ICH_LRC<n>, for n==0 to 7 [e] | 32-bit | Interrupt Controller List Registers 0 to 7, continuation |
| | | c15 | 0-7 | ICH_LRC<n>, for n==8 to 15 [e] | 32-bit | Interrupt Controller List Registers 8 to 15, continuation |
| | 6 | c12 | 4 | ICC_MCTLR [e] | 32-bit | Interrupt Controller Monitor Control Register |
| | | | 5 | ICC_MSRE [e] | 32-bit | Interrupt Controller Monitor System Register Enable register |
| | | | 7 | ICC_MGRPEN1 [e] | 32-bit | Interrupt Controller Monitor Interrupt Group 1 Enable register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order (continued)**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c13 | 0 | c0 | 0 | FCSEIDR | 32-bit | FCSE Process ID Register |
| | | | 1 | CONTEXTIDR | 32-bit | Context ID Register |
| | | | 2 | TPIDRURW | 32-bit | User Read/Write Thread ID Register |
| | | | 3 | TPIDRURO | 32-bit | User Read-Only Thread ID Register |
| | | | 4 | TPIDRPRW | 32-bit | EL1 only Thread ID Register |
| | 4 | c0 | 2 | HTPIDR[b] | 32-bit | Hyp Software Thread ID Register |
| - | 0 | c14 | - | CNTPCT[i] | 64-bit | Physical Count register |
| c14 | 0 | c0 | 0 | CNTFRQ[i] | 32-bit | Counter Frequency register |
| c14 | 0 | c1 | 0 | CNTKCTL[i] | 32-bit | Timer EL1 Control register |
| | | c2 | 0 | CNTP_TVAL[i] | 32-bit | EL1 Physical TimerValue register |
| | | | 1 | CNTP_CTL[i] | 32-bit | EL1 Physical Timer Control register |
| | | c3 | 0 | CNTV_TVAL[i] | 32-bit | Virtual TimerValue register |
| | | | 1 | CNTV_CTL[i] | 32-bit | Virtual Timer Control register |
| c14 | 0 | c8 | 0-7 | PMEVCNTR<n>, for n==0 to 7[d] | 32-bit | Performance Monitors Event Count Registers, 0-7 |
| | | c9 | 0-7 | PMEVCNTR<n>, for n==8 to 15[d] | 32-bit | Performance Monitors Event Count Registers, 8-15 |
| | | c10 | 0-7 | PMEVCNTR<n>, for n==16 to 23[d] | 32-bit | Performance Monitors Event Count Registers, 16-23 |
| | | c11 | 0-6 | PMEVCNTR<n>, for n==24 to 30[d] | 32-bit | Performance Monitors Event Count Registers, 24-30 |
| | | c12 | 0-7 | PMEVTYPER<n>, for n==0 to 7[d] | 32-bit | Performance Monitors Event Type Registers, 0-7 |
| c14 | 0 | c13 | 0-7 | PMEVTYPER<n>, for n==8 to 15[d] | 32-bit | Performance Monitors Event Type Registers, 8-15 |
| | | c14 | 0-7 | PMEVTYPER<n>, for n==16 to 23[d] | 32-bit | Performance Monitors Event Type Registers, 16-23 |
| | | c15 | 0-6 | PMEVTYPER<n>, for n==17 to 30[d] | 32-bit | Performance Monitors Event Type Registers, 24-30 |
| | | c15 | 7 | PMCCFILTR[d] | 32-bit | Performance Monitors Cycle Count Filter Register |
| - | 1 | c14 | - | CNTVCT[i] | 64-bit | Virtual Count register |
| | 2 | | | CNTP_CVAL[i] | 64-bit | EL1 Physical Timer CompareValue register |
| | 3 | | | CNTV_CVAL[i] | 64-bit | Virtual Timer CompareValue register |
| | 4 | | | CNTVOFF[j] | 64-bit | Virtual Offset register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-48 Summary of VMSAv8-32 CP15 register descriptions, in coprocessor register number order (continued)**

| CRn | opc1 | CRm | opc2 | Name | Width | Description |
|---|---|---|---|---|---|---|
| c14 | 4 | c1 | 0 | CNTHCTL | 32-bit | Timer EL2 Control register |
| | | c2 | 0 | CNTHP_TVAL | 32-bit | EL2 Physical TimerValue register |
| | | | 1 | CNTHP_CTL | 32-bit | EL2 Physical Timer Control register |
| - | 6 | c14 | - | CNTHP_CVAL | 64-bit | EL2 Physical Timer CompareValue register |
| c15 | 0-7 | c0-c15 | 0-7 | - | 32-bit | See *VMSAv8-32 CP15 c15 register summary* on page G4-4147 |

a. REVIDR is an optional register. If it is not implemented, the encoding with opc2 set to 6 is an alias of MIDR.

b. Implemented only as part of EL2, when EL2 is using AArch32. Otherwise, encoding is unallocated and CONSTRAINED UNPREDICTABLE, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

c. Implemented only as part of the EL3, when EL3 is using AArch32. Otherwise, as described in *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119, encoding is unallocated and:

- UNDEFINED, for the registers accessed using CRn set to c12.
- CONSTRAINED UNPREDICTABLE, for the register accessed using CRn values other than c12.

d. Introduced in ARMv8.

e. Introduced in ARMv8. Implemented only if the implementation includes the GIC System registers. For information see *Generic Interrupt Controller System registers, functional groups* on page G4-4175. As that subsection describes, each ICV_* register uses the same encoding as the corresponding ICC_* register

f. This register is only accessible in Debug state.

g. When an implementation is using the Long descriptor translation table format these encodings access the MAIR0 and MAIR1registers. Otherwise, they access the PRRR and NMRR.

h. Introduced in ARMv8. Only one of RMR (at EL1), HRMR, and RMR (at EL3) is implemented, corresponding to the highest implemented Exception level, and the register is implemented only if that Exception level is using AArch32.

i. Implemented as part of the Generic Timer.

j. Implemented as RW as part of the Generic Timer on an implementation that includes EL2 and when EL2 is using AArch32. For more information see *Status of the CNTVOFF register* on page G5-4205.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.17.3 AArch32 views of the CP15 registers

The following sections summarize the different software views of the CP15 registers, for VMSAv8-32:

- *PL0 views of the CP15 registers*.
- *PL1 views of the CP15 registers* on page G4-4160.
- *Non-secure PL2 view of the CP15 registers* on page G4-4160.

**Note**

In previous versions of the ARM architecture, the behavior of accesses to the CP15 registers were described in relation to the PE modes. In Secure state, the Exception level of the privileged PE modes depends on whether EL3 is using AArch32, or is using AArch64. This means it is simpler to describe the views of the registers in terms of *privilege levels*, PL0-PL2, rather than the *Exception levels* EL1-EL3. For more information This is because each AArch32 PE mode is associated with a particular privilege level, but in Secure state its Exception level can depend on the EL3 Execution state. *Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990.

#### PL0 views of the CP15 registers

Software executing at PL0, unprivileged, can access only a small subset of the CP15 registers, as Table G4-49 shows. This table excludes possible PL0 access to the following CP15 registers:

- The Performance Monitors Extension, see *Possible PL0 access to the Performance Monitors Extension CP15 registers* on page G4-4159.
- The Generic Timer registers, see *Possible PL0 access to the Generic Timer CP15 registers* on page G4-4159.

**Table G4-49 CP15 registers accessible from PL0**

| Name | Access | Description | Note |
|---|---|---|---|
| CP15ISB | WO | *Memory barriers* on page E2-2375 | ARM deprecates use of these operations |
| CP15DSB | WO | | |
| CP15DMB | WO | | |
| TPIDRURW | RW | *TPIDRURW, PL0 Read/Write Software Thread ID Register* on page G6-4621 | - |
| TPIDRURO | RO | *TPIDRURO, PL0 Read-Only Software Thread ID Register* on page G6-4619 | RW at PL1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Possible PL0 access to the Performance Monitors Extension CP15 registers

In a VMSAv8-32 implementation that includes the Performance Monitors Extension, when using CP15 to access the Performance Monitors registers:

- The PMUSERENR is RO from PL0.
- When:
  - The value of PMUSERENR.EN is 1, PMCNTENSET, PMCNTENCLR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCNTR, PMUSERENR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, and PMCCFILTR.are accessible by reads and writes from PL0.
  - The value of PMUSERENR.ER is 1, PMXEVCNTR and PMEVCNTR<n>.are accessible by reads and from PL0, and PMSELR is accessible by reads and writes from PL0.
  - The value of PMUSERENR.CR is 1, PMCCNTR is accessible by reads from PL0.
  - The value of PMUSERENR.SW is 1, PMSWINC is accessible by writes from PL0.

  In general, when the value of a PMUSERENR.{EN, ER, CR, SW} bit is 1, the enabled registers have the same access permissions from PL0 as they do from PL1.

For more information, see:

- *Traps to EL1 of EL0 accesses to Performance Monitors registers* on page D1-1582.
- Chapter D5 *The Performance Monitors Extension*, in particular *Access permissions* on page D5-1893.

### Possible PL0 access to the Generic Timer CP15 registers

In a VMSAv8-32 implementation, when using CP15 to access the Generic Timer registers:

- If CNTKCTL.PL0PCTEN is set to 1, then if the physical counter register CNTPCT is accessible from PL1 it is also accessible from PL0. For more information see *Accessing the physical counter* on page D6-1900.
- If CNTKCTL.PL0PVTEN is set to 1, the virtual counter register CNTVCT is accessible from PL0. For more information, see *Accessing the virtual counter* on page D6-1901.
- If at least one of CNTKCTL.{PL0PCTEN, PL0PVTEN} is set to 1, the CNTFRQ register is RO from PL0.
- If:
  - CNTKCTL.PL0PTEN is set to 1, the physical timer registers CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are accessible from PL0.
  - CNTKCTL.PL0VTEN is set to 1, the virtual timer registers CNTV_CTL, CNTV_CVAL, and CNTV_TVAL, are accessible from PL0.

  For more information, see *Accessing the timer registers* on page D6-1903.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## PL1 views of the CP15 registers

Software executing at PL1 can access all CP15 registers, with the following exceptions:

**Non-secure PL1 software**

EL3 restricts or prevents access to some registers by Non-secure PL1 software. In particular:

- The Restricted access CP15 registers are either not accessible to Non-secure PL1 software, or are read-only to Non-secure PL1 software, see *Restricted access System registers* on page G4-4124.
- Configuration settings determine whether Non-secure PL1 software can access the Configurable access CP15 registers, see *Configurable access System registers* on page G4-4125.

The individual register descriptions identify these access restrictions.

In an implementation that includes EL2, Non-secure PL1 software has no visibility of the PL2-mode registers summarized in *Hyp mode CP15 read/write registers* on page G4-4125. The individual register descriptions identify these registers as EL2-mode registers.

**Secure PL1 software**

In general, Secure PL1 software has access to all CP15 registers. However:

- When EL3 is using AArch32, the **CP15SDISABLE** signal disables write access to a number of Secure registers, see *The CP15SDISABLE input signal* on page G4-4129.
- The PL2-mode registers are accessible from Secure state only if EL3 is using AArch32. When this is the case, Secure PL1 software can access these registers by moving into Monitor mode, and setting SCR.NS to 1.

  *Hyp mode CP15 read/write registers* on page G4-4125 summarizes these registers.

The individual register descriptions identify:

- The registers affected by the **CP15SDISABLE** signal.
- The PL2-mode registers.

## Non-secure PL2 view of the CP15 registers

Non-secure software executing at PL2 can access:

- The registers that are accessible to Non-secure software executing at PL1, as defined in *PL1 views of the CP15 registers*. Access permissions for these registers are identical to those for Non-secure software executing at EL1.
- The PL2-mode registers summarized in *Hyp mode CP15 read/write registers* on page G4-4125, and described in *Virtualization registers, functional group* on page G4-4164.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.18 Functional grouping of VMSAv8-32 System registers

This section describes how the System registers in an VMSAv8-32 implementation divide into functional groups.

*General system control registers* on page G6-4213 describes each of these registers.

**Note**

- Table G4-48 on page G4-4147 lists all of the VMSAv8-32 CP15 registers, ordered by:
  1. The CP15 primary register used when accessing the register. This is the CRn value for an access to a 32-bit register, or the CRm value for an access to a 64-bit register.
  2. The opc1 value used when accessing the register.
  3. For 32-bit registers, the {CRm, opc2} values used when accessing the register.
- The functional groups defined in this section mainly consist of the VMSAv8-32 registers, but include some additional System registers.
- Some registers belong to more than one functional group.

For other related information see:

- *The conceptual coprocessor interface and system control* on page G1-3839 for general information about the System Control Coprocessor, CP15 and the register access instructions `MRC` and `MCR`
- *About the System registers for VMSAv8-32* on page G4-4116 for general information about the CP15 registers for VMSAv8-32, including:
  - Their organization, both by CP15 primary registers c0 to c15, and by function.
  - Their general behavior.
  - The effect of not implementing some Exception levels on the registers.
  - Different views of the registers, that depend on the state of the PE.
  - Conventions used in describing the registers.

The remainder of this chapter, and *General system control registers* on page G6-4213, assumes you are familiar with *About the System registers for VMSAv8-32* on page G4-4116, and uses conventions and other information from that section without any explanation.

Each of the following sections summarizes a functional group of VMSAv8-32 System registers:

- *Identification registers, functional group* on page G4-4162.
- *Other System registers, functional group* on page G4-4163.
- *Virtual memory control registers, functional group* on page G4-4163.
- *Virtualization registers, functional group* on page G4-4164.
- *Security registers, functional group* on page G4-4167.
- *Exception and fault handling registers, functional group* on page G4-4168.
- *Reset management registers, functional group* on page G4-4169.
- *Thread and process ID registers, functional group* on page G4-4169.
- *Cache maintenance instructions, functional group* on page G4-4169.
- *TLB maintenance instructions, functional group* on page G4-4171.
- *Address translation instructions, functional group* on page G4-4172.
- *Lockdown, DMA, and TCM features, functional group* on page G4-4173.
- *Performance Monitors Extension registers, functional group* on page G4-4173.
- *Generic Timer registers, functional group* on page G4-4174.
- *Generic Interrupt Controller System registers, functional groups* on page G4-4175.
- *Legacy feature registers, functional group* on page G4-4179.
- *IMPLEMENTATION DEFINED registers, functional group* on page G4-4179.
- *Advanced SIMD and floating-point registers, functional group* on page G4-4180.
- *Debug registers, functional group* on page G4-4180.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.18.1 Identification registers, functional group

Table G4-50 shows the VMSAv8-32 CP15 registers in the Identification registers functional group.

**Table G4-50 Identification registers, VMSAv8-32**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| AIDR | c0 | 1 | c0 | 7 | 32-bit | RO | Auxiliary ID Register |
| CCSIDR | c0 | 1 | c0 | 0 | 32-bit | RO | Cache Size ID Registers |
| CLIDR | c0 | 1 | c0 | 1 | 32-bit | RO | Cache Level ID Register |
| CSSELR | c0 | 2 | c0 | 0 | 32-bit | RW | Cache Size Selection Register |
| CTR | c0 | 0 | c0 | 1 | 32-bit | RO | Cache Type Register |
| ID_AFR0 | c0 | 0 | c1 | 3 | 32-bit | RO | Auxiliary Feature Register 0[a] |
| ID_DFR0 | c0 | 0 | c1 | 2 | 32-bit | RO | Debug Feature Register 0[a] |
| ID_ISAR0 | c0 | 0 | c2 | 0 | 32-bit | RO | Instruction Set Attribute Register 0[a] |
| ID_ISAR1 | c0 | 0 | c2 | 1 | 32-bit | RO | Instruction Set Attribute Register 1[a] |
| ID_ISAR2 | c0 | 0 | c2 | 2 | 32-bit | RO | Instruction Set Attribute Register 2[a] |
| ID_ISAR3 | c0 | 0 | c2 | 3 | 32-bit | RO | Instruction Set Attribute Register 3[a] |
| ID_ISAR4 | c0 | 0 | c2 | 4 | 32-bit | RO | Instruction Set Attribute Register 4[a] |
| ID_ISAR5 | c0 | 0 | c2 | 5 | 32-bit | RO | Instruction Set Attribute Register 5[a] |
| ID_MMFR0 | c0 | 0 | c1 | 4 | 32-bit | RO | Memory Model Feature Register 0[a] |
| ID_MMFR1 | c0 | 0 | c1 | 5 | 32-bit | RO | Memory Model Feature Register 1[a] |
| ID_MMFR2 | c0 | 0 | c1 | 6 | 32-bit | RO | Memory Model Feature Register 2[a] |
| ID_MMFR3 | c0 | 0 | c1 | 7 | 32-bit | RO | Memory Model Feature Register 3[a] |
| ID_MMFR4 | c0 | 0 | c2 | 6 | 32-bit | RO | Memory Model Feature Register 4[a] |
| ID_PFR0 | c0 | 0 | c1 | 0 | 32-bit | RO | Processor Feature Register 0[a] |
| ID_PFR1 | c0 | 0 | c1 | 1 | 32-bit | RO | Processor Feature Register 1[a] |
| MIDR | c0 | 0 | c0 | 0 | 32-bit | RO | Main ID Register |
| MPIDR | c0 | 0 | c0 | 5 | 32-bit | RO | Multiprocessor Affinity Register |
| REVIDR | c0 | 0 | c0 | 6 | 32-bit | RO | Revision ID Register |
| TCMTR | c0 | 0 | c0 | 2 | 32-bit | RO | TCM Type Register |
| TLBTR | c0 | 0 | c0 | 3 | 32-bit | RO | TLB Type Register |
| VMPIDR | c0 | 4 | c0 | 5 | 32-bit | RW | Virtualization Multiprocessor ID Register |
| VPIDR | c0 | 4 | c0 | 5 | 32-bit | RW | Virtualization Processor ID Register |

a. CPUID register, see *The CPUID identification scheme* on page G4-4163.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

The other registers in this group are the FPSID, MVFR0, MVFR1, and MVFR2.

The JIDR holds legacy identification information.

#### The CPUID identification scheme

The ID_* registers were originally called the CPUID identification scheme registers. A footnote to Table G4-50 on page G4-4162 identifies these registers. However, functionally, there is no value in separating these registers from the slightly larger Identification registers functional group.

### G4.18.2 Other System registers, functional group

Table G4-51 shows the VMSAv8-32 CP15 registers in the Other System registers functional group.

**Table G4-51 Other System registers, VMSAv8-32**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| ACTLR | c1 | 0 | c0 | 1 | 32-bit | RW | Auxiliary Control Register |
| ACTLR2 | c1 | 0 | c0 | 3 | 32-bit | RW | Auxiliary Control Register 2 |
| CPACR | c1 | 0 | c0 | 2 | 32-bit | RW | Coprocessor Access Control Register |
| HACTLR | c1 | 4 | c0 | 0 | 32-bit | RW | Hyp Auxiliary System Control Register |
| HACTLR2 | c1 | 4 | c0 | 3 | 32-bit | RW | Hyp Auxiliary System Control Register 2 |
| HSCTLR | c1 | 4 | c0 | 0 | 32-bit | RW | Hyp System Control Register |
| SCTLR | c1 | 0 | c0 | 0 | 32-bit | RW | System Control Register |

The following sections summarize the System registers added by the corresponding Exception levels:

- *Security registers, functional group* on page G4-4167.
- *Virtualization registers, functional group* on page G4-4164.

### G4.18.3 Virtual memory control registers, functional group

Table G4-52 shows the VMSAv8-32 CP15 registers in the Virtual memory control registers functional group.

**Table G4-52 Virtual memory control registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| AMAIR0 | c10 | 0 | c3 | 0 | 32-bit | RW | Auxiliary Memory Attribute Indirection Register 0 |
| AMAIR1 | c10 | 0 | c3 | 1 | 32-bit | RW | Auxiliary Memory Attribute Indirection Register 1 |
| CONTEXTIDR | c13 | 0 | c0 | 1 | 32-bit | RW | Context ID Register |
| DACR | c3 | 0 | c0 | 0 | 32-bit | RW | Domain Access Control Register |
| HAMAIR0 | c10 | 4 | c3 | 0 | 32-bit | RW | Hyp Auxiliary Memory Attribute Indirection Register 0 |
| HAMAIR1 | c10 | 4 | c3 | 1 | 32-bit | RW | Hyp Auxiliary Memory Attribute Indirection Register 1 |
| HMAIR0 | c10 | 4 | c2 | 0 | 32-bit | RW | Hyp Memory Attribute Indirection Register 0 |
| HMAIR1 | c10 | 4 | c2 | 1 | 32-bit | RW | Hyp Memory Attribute Indirection Register 1 |
| HTCR | c2 | 4 | c0 | 2 | 32-bit | RW | Hyp Translation Control Register |
| HTTBR | - | 4 | c2 | - | 64-bit | RW | Hyp Translation Table Base Register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-52 Virtual memory control registers (continued)**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| MAIR0 | c10 | 0 | c2 | 0 | 32-bit | RW | Memory Attribute Indirection Register 0 |
| MAIR1 | c10 | 0 | c2 | 1 | 32-bit | RW | Memory Attribute Indirection Register 1 |
| NMRR | c10 | 0 | c2 | 1 | 32-bit | RW | Normal Memory Remap Register |
| PRRR | c10 | 0 | c2 | 0 | 32-bit | RW | Primary Region Remap Register |
| SCTLR | c1 | 0 | c0 | 0 | 32-bit | RW | System Control Register |
| TTBCR | c2 | 0 | c0 | 2 | 32-bit | RW | Translation Table Base Control Register |
| TTBR0 | c2 | 0 | c0 | 0 | 32-bit | RW | Translation Table Base Register 0 |
| TTBR0 | - | 0 | c2 | - | 64-bit | RW | Translation Table Base Register 0 |
| TTBR1 | c2 | 0 | c0 | 1 | 32-bit | RW | Translation Table Base Register 1 |
| TTBR1 | - | 1 | c2 | - | 64-bit | RW | Translation Table Base Register 1 |
| VTCR | c2 | 4 | c1 | 2 | 32-bit | RW | Virtualization Translation Control Register |
| VTTBR | - | 6 | c2 | - | 64-bit | RW | Virtualization Translation Table Base Register |

The ACTLR and, if implemented, ACTLR2, might provide additional virtual memory control.

### G4.18.4 Virtualization registers, functional group

Table G4-53 shows the VMSAv8-32 CP15 registers in the Virtualization registers functional group, excluding the CP15 operations in this group.

**Table G4-53 Virtualization registers, excluding CP15 operations**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| CNTHCTL | c14 | 4 | c1 | 0 | 32-bit | RW | Counter-timer Hyp Control register |
| CNTHP_CTL | c14 | 4 | c2 | 1 | 32-bit | RW | Counter-timer Hyp Physical Timer Control register |
| CNTHP_CVAL | - | 6 | c14 | - | 64-bit | RW | Counter-timer Hyp Physical CompareValue register |
| CNTHP_TVAL | c14 | 4 | c2 | 0 | 32-bit | RW | Counter-timer Hyp Physical Timer TimerValue register |
| CNTVOFF | - | 4 | c14 | - | 64-bit | RW | Counter-timer Virtual Offset register |
| HACR | c1 | 4 | c1 | 7 | 32-bit | RW | Hyp Auxiliary Configuration Register |
| HACTLR | c1 | 4 | c0 | 1 | 32-bit | RW | Hyp Auxiliary Control Register |
| HACTLR2 | c1 | 4 | c0 | 3 | 32-bit | RW | Hyp Auxiliary Control Register 2 |
| HADFSR | c5 | 4 | c1 | 0 | 32-bit | RW | Hyp Auxiliary Data Fault Status Register |
| HAIFSR | c5 | 4 | c1 | 1 | 32-bit | RW | Hyp Auxiliary Instruction Fault Status Register |
| HAMAIR0 | c10 | 4 | c3 | 0 | 32-bit | RW | Hyp Auxiliary Memory Attribute Indirection Register 0 |
| HAMAIR1 | c10 | 4 | c3 | 1 | 32-bit | RW | Hyp Auxiliary Memory Attribute Indirection Register 1 |
| HCPTR | c1 | 4 | c1 | 2 | 32-bit | RW | Hyp Coprocessor Trap Register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-53 Virtualization registers, excluding CP15 operations (continued)**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| HCR | c1 | 4 | c1 | 0 | 32-bit | RW | Hyp Configuration Register |
| HCR2 | c1 | 4 | c1 | 4 | 32-bit | RW | Hyp Configuration Register 2 |
| HDCR | c1 | 4 | c1 | 1 | 32-bit | RW | Hyp Debug Configuration Register |
| HDFAR | c6 | 4 | c0 | 0 | 32-bit | RW | Hyp Data Fault Address Register |
| HIFAR | c6 | 4 | c0 | 2 | 32-bit | RW | Hyp Instruction Fault Address Register |
| HMAIR0 | c10 | 4 | c2 | 0 | 32-bit | RW | Hyp Memory Attribute Indirection Register 0 |
| HMAIR1 | c10 | 4 | c2 | 1 | 32-bit | RW | Hyp Memory Attribute Indirection Register 1 |
| HPFAR | c6 | 4 | c0 | 4 | 32-bit | RW | Hyp IPA Fault Address Register |
| HRMR | c12 | 4 | c0 | 2 | 32-bit | RW | Hyp Reset Management Register |
| HSCTLR | c1 | 4 | c0 | 0 | 32-bit | RW | Hyp System Control Register |
| HSR | c5 | 4 | c2 | 0 | 32-bit | RW | Hyp Syndrome Register |
| HSTR | c1 | 4 | c1 | 3 | 32-bit | RW | Hyp System Trap Register |
| HTCR | c2 | 4 | c0 | 2 | 32-bit | RW | Hyp Translation Control Register |
| HTPIDR | c13 | 4 | c0 | 2 | 32-bit | RW | Hyp Thread and Process ID Register |
| HTTBR | - | 4 | c2 | - | 64-bit | RW | Hyp Translation Table Base Register |
| HVBAR | c12 | 4 | c0 | 0 | 32-bit | RW | Hyp Vector Base Address Register |
| ICC_HSRE [a] | c12 | 4 | c9 | 5 | 32-bit | RW | Interrupt Controller Hyp System Register Enable register |
| ICH_AP0R0 [a] | c12 | 4 | c8 | 0 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (0,0) |
| ICH_AP0R1 [a] | c12 | 4 | c8 | 1 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (0,1) |
| ICH_AP0R2 [a] | c12 | 4 | c8 | 2 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (0,2) |
| ICH_AP0R3 [a] | c12 | 4 | c8 | 3 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (0,3) |
| ICH_AP1R0 [a] | c12 | 4 | c9 | 0 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (1,0) |
| ICH_AP1R1 [a] | c12 | 4 | c9 | 1 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (1,1) |
| ICH_AP1R2 [a] | c12 | 4 | c9 | 2 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (1,2) |
| ICH_AP1R3 [a] | c12 | 4 | c9 | 3 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (1,3) |
| ICH_EISR [a] | c12 | 4 | c11 | 3 | 32-bit | RO | Interrupt Controller End of Interrupt Status Register |
| ICH_ELRSR [a] | c12 | 4 | c11 | 5 | 32-bit | RO | Interrupt Controller Empty List Register Status Register |
| ICH_HCR [a] | c12 | 4 | c11 | 0 | 32-bit | RW | Interrupt Controller Hyp Control Register |
| ICH_LR<n>, n==0-7 [a] | c12 | 4 | c12 | 0-7 | 32-bit | RW | Interrupt Controller List Registers, 0-7 |
| ICH_LR<n>, n==8-15 [a] | c12 | 4 | c13 | 0-7 | 32-bit | RW | Interrupt Controller List Registers, 8-15 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-53 Virtualization registers, excluding CP15 operations (continued)**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| ICH_LRC<n>, n==0-7 [a] | c12 | 4 | c14 | 0-7 | 32-bit | RW | Interrupt Controller List Registers Continuation, 0-7 |
| ICH_LRC<n>, n==8-15 [a] | c12 | 4 | c15 | 0-7 | 32-bit | RW | Interrupt Controller List Registers Continuation, 8-15 |
| ICH_MISR [a] | c12 | 4 | c11 | 2 | 32-bit | RO | Interrupt Controller Maintenance Interrupt State Register |
| ICH_VMCR [a] | c12 | 4 | c11 | 7 | 32-bit | RW | Interrupt Controller Virtual Machine Control Register |
| ICH_VTR [a] | c12 | 4 | c11 | 1 | 32-bit | RO | Interrupt Controller VGIC Type Register |
| VMPIDR | c0 | 4 | c0 | 5 | 32-bit | RW | Virtualization Multiprocessor ID Register |
| VPIDR | c0 | 4 | c0 | 0 | 32-bit | RW | Virtualization Processor ID Register |
| VTCR | c2 | 4 | c1 | 2 | 32-bit | RW | Virtualization Translation Control Register |
| VTTBR | - | 6 | c2 | - | 64-bit | RW | Virtualization Translation Table Base Register |

a. For more information about these registers see *Generic Interrupt Controller System registers, functional groups* on page G4-4175. As that section describes, each ICV_* register uses the same encoding as the corresponding ICC_* register.

Table G4-54 shows the Hyp mode CP15 operations, that are part of this functional group. See also Table G4-53 on page G4-4164.

**Table G4-54 Hyp mode CP15 operations**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| ATS1HR | c7 | 4 | c8 | 0 | 32-bit | WO | Address Translate Stage 1 Hyp mode Read |
| ATS1HW | c7 | 4 | c8 | 1 | 32-bit | WO | Address Translate Stage 1 Hyp mode Write |
| TLBIALLH [a] | c8 | 4 | c7 | 0 | 32-bit | WO | Invalidate entire Hyp unified TLB |
| TLBIALLHIS [a] | c8 | 4 | c3 | 0 | 32-bit | WO | Invalidate entire Hyp unified TLB |
| TLBIALLNSNH [a] | c8 | 4 | c7 | 4 | 32-bit | WO | Invalidate entire Non-secure Non-Hyp unified TLB |
| TLBIALLNSNHIS [a] | c8 | 4 | c3 | 4 | 32-bit | WO | Invalidate entire Non-secure Non-Hyp unified TLB |
| TLBIIPAS2 [a] | c8 | 4 | c4 | 1 | 32-bit | WO | TLB Invalidate entry by IPA, Stage 2 |
| TLBIIPAS2IS [a] | c8 | 4 | c0 | 1 | 32-bit | WO | TLB Invalidate entry by IPA, Stage 2, Inner Shareable |
| TLBIIPAS2L [a] | c8 | 4 | c4 | 5 | 32-bit | WO | TLB Invalidate entry by IPA, Stage 2, Last level |
| TLBIIPAS2LIS [a] | c8 | 4 | c0 | 5 | 32-bit | WO | TLB Invalidate entry by IPA, Stage 2, Last level, Inner Shareable |
| TLBIMVAH [a] | c8 | 4 | c7 | 1 | 32-bit | WO | Invalidate Hyp unified TLB by VA |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-54 Hyp mode CP15 operations (continued)**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| TLBIMVAHIS[a] | c8 | 4 | c3 | 1 | 32-bit | WO | Invalidate Hyp unified TLB by VA |
| TLBIMVALH[a] | c8 | 4 | c7 | 5 | 32-bit | WO | TLB Invalidate entry by MVA, Last level, Hyp mode |
| TLBIMVALHIS[a] | c8 | 4 | c3 | 5 | 32-bit | WO | TLB Invalidate entry by MVA, Last level, Hyp mode, Inner Shareable |

a. These links are to a summary of the operation, and *The scope of TLB maintenance instructions* on page G4-4069 describes the operation.

All the encodings shown in Table G4-53 on page G4-4164 and Table G4-54 on page G4-4166 are unallocated and CONSTRAINED UNPREDICTABLE on an implementation that does not include EL2, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

### G4.18.5 Security registers, functional group

Table G4-55 shows the VMSAv8-32 CP15 registers in the Security registers functional group.

**Table G4-55 Security registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| ICC_MCTLR [a] | c12 | 6 | c12 | 4 | 32-bit | RW | Interrupt Controller Monitor Control Register |
| ICC_MGRPEN1 [a] | c12 | 6 | c12 | 7 | 32-bit | RW | Interrupt Controller Monitor Interrupt Group 1 Enable register |
| ICC_MSRE [a] | c12 | 6 | c12 | 5 | 32-bit | RW | Interrupt Controller Monitor System Register Enable register |
| MVBAR | c12 | 0 | c0 | 1 | 32-bit | RW | Monitor Vector Base Address Register |
| NSACR | c1 | 0 | c1 | 2 | 32-bit | RW | Non-Secure Access Control Register |
| RMR (at EL3) | c12 | 0 | c0 | 2 | 32-bit | RW | Reset Management Register |
| SCR | c1 | 0 | c1 | 0 | 32-bit | RW | Secure Configuration Register |
| SDER | c1 | 0 | c1 | 1 | 32-bit | RW | Secure Debug Enable Register |

a. For information about these registers see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).

All the encodings shown in Table G4-55 are unallocated and CONSTRAINED UNPREDICTABLE on an implementation that does not include EL3, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.18.6 Exception and fault handling registers, functional group

Table G4-56 shows the VMSAv8-32 CP15 registers in the Exception and fault handling registers functional group.

**Table G4-56 Exception and fault handling registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| ADFSR | c5 | 0 | c1 | 0 | 32-bit | RW | Auxiliary Data Fault Status Register |
| AIFSR | c5 | 0 | c1 | 1 | 32-bit | RW | Auxiliary Instruction Fault Status Register |
| DFAR | c6 | 0 | c0 | 0 | 32-bit | RW | Data Fault Address Register |
| DFSR | c5 | 0 | c0 | 0 | 32-bit | RW | Data Fault Status Register |
| HADFSR | c5 | 4 | c1 | 0 | 32-bit | RW | Hyp Auxiliary Data Fault Status Register |
| HAIFSR | c5 | 4 | c1 | 1 | 32-bit | RW | Hyp Auxiliary Instruction Fault Status Register |
| HDFAR | c6 | 4 | c0 | 0 | 32-bit | RW | Hyp Data Fault Address Register |
| HIFAR | c6 | 4 | c0 | 2 | 32-bit | RW | Hyp Instruction Fault Address Register |
| HPFAR | c6 | 4 | c0 | 4 | 32-bit | RW | Hyp IPA Fault Address Register |
| HSR | c5 | 4 | c2 | 0 | 32-bit | RW | Hyp Syndrome Register |
| HVBAR | c12 | 4 | c0 | 1 | 32-bit | RW | Hyp Vector Base Address Register |
| IFAR | c6 | 0 | c0 | 2 | 32-bit | RW | Instruction Fault Address Register |
| IFSR | c5 | 0 | c0 | 1 | 32-bit | RW | Instruction Fault Status Register |
| ISR | c12 | 0 | c1 | 0 | 32-bit | RO | Interrupt Status Register |
| MVBAR | c12 | 0 | c0 | 1 | 32-bit | RW | Monitor Vector Base Address Register |
| RVBAR | c12 | 0 | c0 | 1 | 32-bit | RW | Reset Vector Base Address Register |
| VBAR | c12 | 0 | c0 | 0 | 32-bit | RW | Vector Base Address Register |

The PE returns fault information using the fault status registers and the fault address registers. For details of how these registers are used see *Exception reporting in a VMSAv8-32 implementation* on page G4-4091.

**Note**

These registers also report information about debug exceptions. For more information see:

- *Data Abort exceptions, taken to a PL1 mode* on page G4-4093.
- *Prefetch Abort exceptions, taken to a PL1 mode* on page G4-4095.
- *Reporting exceptions taken to Hyp mode* on page G4-4101.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.18.7 Reset management registers, functional group

Table G4-57 shows the VMSAv8-32 CP15 registers in the Reset management registers functional group.

**Table G4-57 Reset management registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| HRMR | c12 | 4 | c0 | 2 | 32-bit | RW | Hyp Reset Management Register |
| RMR (at EL1) | c12 | 0 | c0 | 2 | 32-bit | RW | Reset Management Register, EL1 |
| RMR (at EL3) | c12 | 0 | c0 | 2 | 32-bit | RW | Reset Management Register, EL3 |

### G4.18.8 Thread and process ID registers, functional group

Table G4-58 shows the VMSAv8-32 Thread and process ID registers.

**Table G4-58 VMSAv8-32 Thread and process ID registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Type[a] | Description |
|---|---|---|---|---|---|---|---|
| HTPIDR[b] | c13 | 4 | c0 | 2 | 32-bit | RW | Hyp Software Thread ID Register |
| TPIDRPRW | c13 | 0 | c0 | 4 | 32-bit | RW | PL1 Software Thread ID Register |
| TPIDRURO | c13 | 0 | c0 | 3 | 32-bit | RW, PL0 | PL0 Read-Only Software Thread ID Register |
| TPIDRURW | c13 | 0 | c0 | 2 | 32-bit | RW, PL0 | PL0 Read/Write Software Thread ID Register |

a. PL0 in a Type description indicates that the encoding is accessible by software executing at PL0. See the register description for more information.

b. Implemented only as part of EL2. Otherwise, encoding is unallocated and CONSTRAINED UNPREDICTABLE, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.

### G4.18.9 Cache maintenance instructions, functional group

Table G4-59 shows the VMSAv8-32 Cache and branch predictor maintenance operations functional group.

**Table G4-59 Cache and branch predictor maintenance operations**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description | Limits[a] |
|---|---|---|---|---|---|---|---|---|
| BPIALL[b] | c7 | 0 | c5 | 6 | 32-bit | WO | Branch predictor invalidate all | - |
| BPIALLIS[b] | c7 | 0 | c1 | 6 | 32-bit | WO | Branch predictor invalidate all | IS |
| BPIMVA[b] | c7 | 0 | c5 | 7 | 32-bit | WO | Branch predictor invalidate by VA | - |
| DCCIMVAC[b] | c7 | 0 | c14 | 1 | 32-bit | WO | Data cache clean and invalidate by VA | PoC |
| DCCISW[b] | c7 | 0 | c14 | 2 | 32-bit | WO | Data cache clean and invalidate by set/way | - |
| DCCMVAC[b] | c7 | 0 | c10 | 1 | 32-bit | WO | Data cache clean by VA | PoC |
| DCCMVAU[b] | c7 | 0 | c11 | 1 | 32-bit | WO | Data cache clean by VA | PoU |
| DCCSW[b] | c7 | 0 | c10 | 2 | 32-bit | WO | Data cache clean by set/way | - |
| DCIMVAC[b] | c7 | 0 | c6 | 1 | 32-bit | WO | Data cache invalidate by VA | PoC |
| DCISW[b] | c7 | 0 | c6 | 2 | 32-bit | WO | Data cache invalidate by set/way | - |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table G4-59 Cache and branch predictor maintenance operations (continued)**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description | Limits[a] |
|---|---|---|---|---|---|---|---|---|
| ICIALLU[b] | c7 | 0 | c5 | 0 | 32-bit | WO | Instruction cache invalidate all | PoU |
| ICIALLUIS[b] | c7 | 0 | c1 | 0 | 32-bit | WO | Instruction cache invalidate all | PoU, IS |
| ICIMVAU[b] | c7 | 0 | c5 | 1 | 32-bit | WO | Instruction cache invalidate by VA | PoU |

a. PoU = to Point of Unification, PoC = to Point of Coherency, IS = Inner Shareable.

b. The links in this column are to a summary of the operation, see *AArch32 cache and branch predictor maintenance instructions* on page G3-3961.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### G4.18.10 TLB maintenance instructions, functional group

Table G4-60 shows the VMSAv8-32 TLB maintenance instructions functional group. *The scope of TLB maintenance instructions* on page G4-4069 describes the operations.

**Table G4-60 TLB maintenance instructions**

| Name[a] | CRn | opc1 | CRm | opc2 | Width | Type | Description | Limits[b] |
|---|---|---|---|---|---|---|---|---|
| DTLBIALL[c] | c8 | 0 | c6 | 0 | 32-bit | WO | Invalidate entire data TLB | - |
| DTLBIASID[c] | c8 | 0 | c6 | 2 | 32-bit | WO | Invalidate data TLB by ASID | - |
| DTLBIMVA[c] | c8 | 0 | c6 | 1 | 32-bit | WO | Invalidate data TLB entry by VA | - |
| ITLBIALL[c] | c8 | 0 | c5 | 0 | 32-bit | WO | Invalidate entire instruction TLB | - |
| ITLBIASID[c] | c8 | 0 | c5 | 2 | 32-bit | WO | Invalidate instruction TLB by ASID | - |
| ITLBIMVA[c] | c8 | 0 | c5 | 1 | 32-bit | WO | Invalidate instruction TLB by VA | - |
| TLBIALL[d] | c8 | 0 | c7 | 0 | 32-bit | WO | Invalidate entire unified TLB | - |
| TLBIALLH | c8 | 4 | c7 | 0 | 32-bit | WO | Invalidate entire Hyp unified TLB | - |
| TLBIALLHIS | c8 | 4 | c3 | 0 | 32-bit | WO | Invalidate entire Hyp unified TLB | IS |
| TLBIALLIS | c8 | 0 | c3 | 0 | 32-bit | WO | Invalidate entire unified TLB | IS |
| TLBIALLNSNH | c8 | 4 | c7 | 4 | 32-bit | WO | Invalidate entire Non-secure Non-Hyp unified TLB | - |
| TLBIALLNSNHIS | c8 | 4 | c3 | 4 | 32-bit | WO | Invalidate entire Non-secure Non-Hyp unified TLB | IS |
| TLBIASID | c8 | 0 | c7 | 2 | 32-bit | WO | Invalidate unified TLB by ASID | - |
| TLBIASIDIS | c8 | 0 | c3 | 2 | 32-bit | WO | Invalidate unified TLB by ASID | IS |
| TLBIIPAS2 | c8 | 4 | c4 | 1 | 32-bit | WO | TLB Invalidate entry by IPA, Stage 2 | - |
| TLBIIPAS2IS | c8 | 4 | c0 | 1 | 32-bit | WO | TLB Invalidate entry by IPA, Stage 2, Inner Shareable | IS |
| TLBIIPAS2L | c8 | 4 | c4 | 5 | 32-bit | WO | TLB Invalidate entry by IPA, Stage 2, Last level | - |
| TLBIIPAS2LIS | c8 | 4 | c0 | 5 | 32-bit | WO | TLB Invalidate entry by IPA, Stage 2, Last level, Inner Shareable | IS |
| TLBIMVAA | c8 | 0 | c7 | 3 | 32-bit | WO | Invalidate unified TLB by VA, all ASID | - |
| TLBIMVAAIS | c8 | 0 | c3 | 3 | 32-bit | WO | Invalidate unified TLB by VA, all ASID | IS |
| TLBIMVAAL | c8 | 0 | c7 | 7 | 32-bit | WO | TLB Invalidate entry by MVA, All ASID, Last level | - |
| TLBIMVAALIS | c8 | 0 | c3 | 7 | 32-bit | WO | TLB Invalidate entry by MVA, All ASID, Last level, Inner Shareable | IS |
| TLBIMVA | c8 | 0 | c7 | 1 | 32-bit | WO | Invalidate unified TLB by VA | - |
| TLBIMVAH | c8 | 4 | c7 | 1 | 32-bit | WO | Invalidate Hyp unified TLB by VA | - |
| TLBIMVAHIS | c8 | 4 | c3 | 1 | 32-bit | WO | Invalidate Hyp unified TLB by VA | IS |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-60 TLB maintenance instructions (continued)**

| Name[a] | CRn | opc1 | CRm | opc2 | Width | Type | Description | Limits[b] |
|---|---|---|---|---|---|---|---|---|
| TLBIMVAIS | c8 | 0 | c3 | 1 | 32-bit | WO | Invalidate unified TLB by VA | IS |
| TLBIMVAL | c8 | 0 | c7 | 5 | 32-bit | WO | TLB Invalidate entry by MVA, Last level | - |
| TLBIMVALH | c8 | 4 | c7 | 5 | 32-bit | WO | TLB Invalidate entry by MVA, Last level, Hyp mode | - |
| TLBIMVALHIS | c8 | 4 | c3 | 5 | 32-bit | WO | TLB Invalidate entry by MVA, Last level, Hyp mode, Inner Shareable | IS |
| TLBIMVALIS | c8 | 0 | c3 | 5 | 32-bit | WO | TLB Invalidate entry by MVA, Last level | IS |

a. These links are to a summary of the operation, and *The scope of TLB maintenance instructions* on page G4-4069 describes the operation.
b. IS = Inner Shareable.
c. Deprecated. ARM deprecates use of operations that operate only on an Instruction TLB, or only on a Data TLB.
d. The mnemonics for the operations with CRm==c7, opc2== {0, 1, 2} were previously UTLBIALL, UTLBIMVA and UTLBIMASID.

### G4.18.11 Address translation instructions, functional group

Table G4-61 shows the VMSAv8-32 Address translation instructions functional group.

**Table G4-61 Address translation instructions**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| ATS12NSOPR[a, c] | c7 | 0 | c8 | 4 | 32-bit | WO | Stages 1 and 2 Non-secure only EL1 read |
| ATS12NSOPW[a, c] | c7 | 0 | c8 | 5 | 32-bit | WO | Stages 1 and 2 Non-secure only EL1 write |
| ATS12NSOUR[a, c] | c7 | 0 | c8 | 6 | 32-bit | WO | Stages 1 and 2 Non-secure only unprivileged read |
| ATS12NSOUW[a, c] | c7 | 0 | c8 | 7 | 32-bit | WO | Stages 1 and 2 Non-secure only unprivileged write |
| ATS1CPR[c] | c7 | 0 | c8 | 0 | 32-bit | WO | Stage 1 Current state EL1 read |
| ATS1CPW[c] | c7 | 0 | c8 | 1 | 32-bit | WO | Stage 1 Current state EL1 write |
| ATS1CUR[c] | c7 | 0 | c8 | 2 | 32-bit | WO | Stage 1 Current state unprivileged read |
| ATS1CUW[c] | c7 | 0 | c8 | 3 | 32-bit | WO | Stage 1 Current state unprivileged write |
| ATS1HR[b, c] | c7 | 4 | c8 | 0 | 32-bit | WO | Stage 1 Hyp mode read |
| ATS1HW[b, c] | c7 | 4 | c8 | 1 | 32-bit | WO | Stage 1 Hyp mode write |
| PAR | c7 | 0 | c4 | 0 | 32-bit | RW | Physical Address Register |
| | - | 0 | c7 | - | 64-bit | RW | |

a. Implemented only as part of EL3. Otherwise, encoding is unallocated and CONSTRAINED UNPREDICTABLE, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.
b. Implemented only as part of EL2. Otherwise, encoding is unallocated and CONSTRAINED UNPREDICTABLE, see *Accesses to unallocated encodings in the CP14 and CP15 encoding space* on page G4-4119.
c. These links are to a summary of the operation.

*Address translation instructions* on page G4-4110 describes these operations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.18.12 Lockdown, DMA, and TCM features, functional group

Table G4-62 shows the VMSAv8-32 reserved encodings for the Lockdown, DMA, and TCM features registers functional group.

**Table G4-62 Lockdown, DMA, and TCM features, VMSAv8-32**

| Name | CRn | opc1 | CRm | Width | opc2 | Type | Description |
|---|---|---|---|---|---|---|---|
| IMPLEMENTATION DEFINED | c9 | 0-7 | c0-c2 | 32-bit | 0-7 | a | *VMSAv8-32 CP15 c9 register summary* on page G4-4144 |
| | | | c5-c8 | 32-bit | 0-7 | a | |
| | c10 | 0 | c0-c1 | 32-bit | 0-7 | a | *VMSAv8-32 CP15 c10 register summary* on page G4-4145 |
| | | | c4 | 32-bit | 0-7 | a | |
| | | | c8 | 32-bit | 0-7 | a | |
| | c11 | 0-7 | c0-c8 | 32-bit | 0-7 | a | *VMSAv8-32 CP15 c11 register summary* on page G4-4145 |
| | | | c15 | 32-bit | 0-7 | a | |

a. Access depends on the register or operation, and is IMPLEMENTATION DEFINED.

### G4.18.13 Performance Monitors Extension registers, functional group

Table G4-63 shows the VMSAv8-32 Performance Monitors Extension registers functional group. See also *IMPLEMENTATION DEFINED performance monitors* on page G4-4174.

**Table G4-63 Performance Monitors Extension registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Description |
|---|---|---|---|---|---|---|
| PMCCFILTR | c14 | 0 | c15 | 7 | 32-bit | Performance Monitors Cycle Count Filter Register |
| PMCCNTR | c9 | 0 | c13 | 0 | 32-bit | Performance Monitors Cycle Count Register |
| PMCEID0 | c9 | 0 | c12 | 6 | 32-bit | Performance Monitors Common Event Identification register 0 |
| PMCEID1 | c9 | 0 | c12 | 7 | 32-bit | Performance Monitors Common Event Identification register 1 |
| PMCNTENCLR | c9 | 0 | c12 | 2 | 32-bit | Performance Monitors Count Enable Clear register |
| PMCNTENSET | c9 | 0 | c12 | 1 | 32-bit | Performance Monitors Count Enable Set register |
| PMCR | c9 | 0 | c12 | 0 | 32-bit | Performance Monitors Control Register |
| PMEVCNTR<n>, for n==0 to 7 | c14 | 0 | c8 | 0-7 | 32-bit | Performance Monitors Event Count Registers, 0-7 |
| PMEVCNTR<n>, for n==16 to 23 | c14 | 0 | c10 | 0-7 | 32-bit | Performance Monitors Event Count Registers, 16-23 |
| PMEVCNTR<n>, for n==24 to 30 | c14 | 0 | c11 | 0-6 | 32-bit | Performance Monitors Event Count Registers, 24-30 |
| PMEVCNTR<n>, for n==8 to 15 | c14 | 0 | c9 | 0-7 | 32-bit | Performance Monitors Event Count Registers, 8-15 |
| PMEVTYPER<n>, for n==0 to 7 | c14 | 0 | c12 | 0-7 | 32-bit | Performance Monitors Event Type Registers, 0-7 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-63 Performance Monitors Extension registers (continued)**

| Name | CRn | opc1 | CRm | opc2 | Width | Description |
|---|---|---|---|---|---|---|
| PMEVTYPER<n>, for n==16 to 23 | c14 | 0 | c14 | 0-7 | 32-bit | Performance Monitors Event Type Registers, 16-23 |
| PMEVTYPER<n>, for n==17 to 30 | c14 | 0 | c15 | 0-6 | 32-bit | Performance Monitors Event Type Registers, 24-30 |
| PMEVTYPER<n>, for n==8 to 15 | c14 | 0 | c13 | 0-7 | 32-bit | Performance Monitors Event Type Registers, 8-15 |
| PMINTENCLR | c9 | 0 | c14 | 2 | 32-bit | Performance Monitors Interrupt Enable Clear register |
| PMINTENSET | c9 | 0 | c14 | 1 | 32-bit | Performance Monitors Interrupt Enable Set register |
| PMOVSR | c9 | 0 | c12 | 3 | 32-bit | Performance Monitors Overflow Flag Status Register |
| PMOVSSET | c9 | 0 | c14 | 3 | 32-bit | Performance Monitors Overflow Flag Status Set register |
| PMSELR | c9 | 0 | c12 | 5 | 32-bit | Performance Monitors Event Counter Selection Register |
| PMSWINC | c9 | 0 | c12 | 4 | 32-bit | Performance Monitors Software Increment register |
| PMUSERENR | c9 | 0 | c14 | 0 | 32-bit | Performance Monitors User Enable Register |
| PMXEVCNTR | c9 | 0 | c13 | 2 | 32-bit | Performance Monitors Event Count Register |
| PMXEVTYPER | c9 | 0 | c13 | 1 | 32-bit | Performance Monitors Event Type Select Register |

### IMPLEMENTATION DEFINED performance monitors

VMSAv8-32 reserves some additional CP15 c9 encodings for optional additional IMPLEMENTATION DEFINED performance monitors. Table G4-64 shows the allocation of CP15 c9 encodings:

**Table G4-64 Performance Monitors register encoding allocations in CP15 c9**

| CRn | opc1 | CRm | opc2 | Name | Width | Type |
|---|---|---|---|---|---|---|
| c9 | 0-7 | c12-c14 | 0-7 | Performance Monitors Extension registers, see Table G4-63 on page G4-4173 | 32-bit | RW or RO[a] |
| | | c15 | 0-7 | IMPLEMENTATION DEFINED | 32-bit | [b] |

a. The table referenced in the *Name* entry shows the type of each of the OPTIONAL Performance Monitors Extension registers.
b. Access depends on the register or operation, and is IMPLEMENTATION DEFINED.

## G4.18.14 Generic Timer registers, functional group

AArch32 state reserves CP15 primary coprocessor register c14 for access to the Generic Timer registers. Table G4-65 shows the VMSAv8-32 CP15 registers in the Generic Timer registers functional group.

**Table G4-65 Generic Timer registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Type[a] | Description |
|---|---|---|---|---|---|---|---|
| CNTFRQ | c14 | 0 | c0 | 0 | 32-bit | RW | Counter Frequency register |
| CNTHCTL | c14 | 4 | c1 | 0 | 32-bit | RW | Timer EL2 Control register |
| CNTHP_CTL | c14 | 4 | c2 | 1 | 32-bit | RW | EL2 Physical Timer Control register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-65 Generic Timer registers (continued)**

| Name | CRn | opc1 | CRm | opc2 | Width | Type[a] | Description |
|---|---|---|---|---|---|---|---|
| CNTHP_CVAL | - | 6 | c14 | - | 64-bit | RW | EL2 Physical Timer CompareValue register |
| CNTHP_TVAL | c14 | 4 | c2 | 0 | 32-bit | RW | EL2 Physical TimerValue register |
| CNTKCTL | c14 | 0 | c1 | 0 | 32-bit | RW | Timer EL1 Control register |
| CNTP_CTL | c14 | 0 | c2 | 1 | 32-bit | RW | EL1 Physical Timer Control register |
| CNTP_CVAL | - | 2 | c14 | - | 64-bit | RW | EL1 Physical Timer CompareValue register |
| CNTP_TVAL | c14 | 0 | c2 | 0 | 32-bit | RW | EL1 Physical TimerValue register |
| CNTPCT | - | 0 | c14 | - | 64-bit | RW | Physical Count register |
| CNTV_CTL | c14 | 0 | c3 | 1 | 32-bit | RW | Virtual Timer Control register |
| CNTV_CVAL | - | 3 | c14 | - | 64-bit | RW | Virtual Timer CompareValue register |
| CNTV_TVAL | c14 | 0 | c3 | 0 | 32-bit | RW | Virtual TimerValue register |
| CNTVCT | - | 1 | c14 | - | 64-bit | RO | Virtual Count register |
| CNTVOFF[b] | - | 4 | c14 | - | 64-bit | RW | Virtual Offset register |

a. See the register descriptions for more information. Accessibility can depend on configuration settings as well as on the current Exception level.

b. Implemented as RW as part of the Generic Timer on an implementation that includes EL2 and when EL2 is using AArch32. For more information see *Status of the CNTVOFF register* on page G5-4205.

### G4.18.15 Generic Interrupt Controller System registers, functional groups

From version 3.0 of the GIC architecture specification, the specification defines three groups of System registers, identified by the prefix of the register name:

**ICC_** GIC physical CPU interface System registers.

**ICH_** GIC virtual interface control System registers.

**ICV_** GIC Virtual CPU interface System registers.

**Note**

These registers are in addition to the GIC memory-mapped register groups GICC_, GICD_, GICH_, GICR_, GICV_, and GITS_.

In VMSAv8-32, the GIC System registers are all in the CP15 encoding space with CRn equal to c12. The ICV_* registers have the same {CRn, opc1, CRm, op2} encodings as the corresponding ICC_* registers. For these encodings, GIC register configuration fields determine which register is accessed.

For more information see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).

Table G4-66 on page G4-4176 shows the functional groups of GIC System registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-66 Generic Interrupt Controller System registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| ICC_AP0R0[a]<br>ICV_AP0R0[a] | c12 | 0 | c8 | 4 | 32-bit | RW | Interrupt Controller Active Priorities Register (0,0)<br>Interrupt Controller Virtual Active Priorities Register (0,0) |
| ICC_AP0R1[a]<br>ICV_AP0R1[a] | c12 | 0 | c8 | 5 | 32-bit | RW | Interrupt Controller Active Priorities Register (0,1)<br>Interrupt Controller Virtual Active Priorities Register (0,1) |
| ICC_AP0R2[a]<br>ICV_AP0R2[a] | c12 | 0 | c8 | 6 | 32-bit | RW | Interrupt Controller Active Priorities Register (0,2)<br>Interrupt Controller Virtual Active Priorities Register (0,2) |
| ICC_AP0R3[a]<br>ICV_AP0R3[a] | c12 | 0 | c8 | 7 | 32-bit | RW | Interrupt Controller Active Priorities Register (0,3)<br>Interrupt Controller Virtual Active Priorities Register (0,3) |
| ICC_AP1R0[a]<br>ICV_AP1R0[a] | c12 | 0 | c9 | 0 | 32-bit | RW | Interrupt Controller Active Priorities Register (1,0)<br>Interrupt Controller Virtual Active Priorities Register (1,0) |
| ICC_AP1R1[a]<br>ICV_AP1R1[a] | c12 | 0 | c9 | 1 | 32-bit | RW | Interrupt Controller Active Priorities Register (1,1)<br>Interrupt Controller Virtual Active Priorities Register (1,1) |
| ICC_AP1R2[a]<br>ICV_AP1R2[a] | c12 | 0 | c9 | 2 | 32-bit | RW | Interrupt Controller Active Priorities Register (1,2)<br>Interrupt Controller Virtual Active Priorities Register (1,2) |
| ICC_AP1R3[a]<br>ICV_AP1R3[a] | c12 | 0 | c9 | 3 | 32-bit | RW | Interrupt Controller Active Priorities Register (1,3)<br>Interrupt Controller Virtual Active Priorities Register (1,3) |
| ICC_ASGI1R | - | 1 | c12 | - | 64-bit | WO | Interrupt Controller Alias Software Generated Interrupt group 1 Register |
| ICC_BPR0[a]<br>ICV_BPR0[a] | c12 | 0 | c8 | 3 | 32-bit | RW | Interrupt Controller Binary Point Register 0<br>Interrupt Controller Virtual Binary Point Register 0 |
| ICC_BPR1[a]<br>ICV_BPR1[a] | c12 | 0 | c12 | 3 | 32-bit | RW | Interrupt Controller Binary Point Register 1<br>Interrupt Controller Virtual Binary Point Register 1 |
| ICC_CTLR[a]<br>ICV_CTLR[a] | c12 | 0 | c12 | 4 | 32-bit | RW | Interrupt Controller Control Register<br>Interrupt Controller Virtual Control Register |
| ICC_DIR[a]<br>ICV_DIR[a] | c12 | 0 | c11 | 1 | 32-bit | WO | Interrupt Controller Deactivate Interrupt Register<br>Interrupt Controller Deactivate Virtual Interrupt Register |
| ICC_EOIR0[a]<br>ICV_EOIR0[a] | c12 | 0 | c8 | 1 | 32-bit | WO | Interrupt Controller End Of Interrupt Register 0<br>Interrupt Controller Virtual End Of Interrupt Register 0 |
| ICC_EOIR1[a]<br>ICV_EOIR1[a] | c12 | 0 | c12 | 1 | 32-bit | WO | Interrupt Controller End Of Interrupt Register 1<br>Interrupt Controller Virtual End Of Interrupt Register 1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-66 Generic Interrupt Controller System registers (continued)**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| ICC_HPPIR0[a] | c12 | 0 | c8 | 2 | 32-bit | RO | Interrupt Controller Highest Priority Pending Interrupt Register 0 |
| ICV_HPPIR0[a] | | | | | | | Interrupt Controller Virtual Highest Priority Pending Interrupt Register 0 |
| ICC_HPPIR1[a] | c12 | 0 | c12 | 2 | 32-bit | RO | Interrupt Controller Highest Priority Pending Interrupt Register 1 |
| ICV_HPPIR1[a] | | | | | | | Interrupt Controller Virtual Highest Priority Pending Interrupt Register 1 |
| ICC_HSRE | c12 | 4 | c9 | 5 | 32-bit | RW | Interrupt Controller Hyp System Register Enable register |
| ICC_IAR0[a] | c12 | 0 | c8 | 0 | 32-bit | RO | Interrupt Controller Interrupt Acknowledge Register 0 |
| ICV_IAR0[a] | | | | | | | Interrupt Controller Virtual Interrupt Acknowledge Register 0 |
| ICC_IAR1[a] | c12 | 0 | c12 | 0 | 32-bit | RO | Interrupt Controller Interrupt Acknowledge Register 1 |
| ICV_IAR1[a] | | | | | | | Interrupt Controller Virtual Interrupt Acknowledge Register 1 |
| ICC_IGRPENR0[a] | c12 | 0 | c12 | 6 | 32-bit | RW | Interrupt Controller Interrupt Group 0 Enable register |
| ICV_IGRPENR0[a] | | | | | | | Interrupt Controller Virtual Interrupt Group 0 Enable register |
| ICC_IGRPENR1[a] | c12 | 0 | c12 | 7 | 32-bit | RW | Interrupt Controller Interrupt Group 1 Enable register |
| ICV_IGRPENR1[a] | | | | | | | Interrupt Controller Virtual Interrupt Group 1 Enable register |
| ICC_MCTLR | c12 | 6 | c12 | 4 | 32-bit | RW | Interrupt Controller Monitor Control Register |
| ICC_MGRPEN1 | c12 | 6 | c12 | 7 | 32-bit | RW | Interrupt Controller Monitor Interrupt Group 1 Enable register |
| ICC_MSRE | c12 | 6 | c12 | 5 | 32-bit | RW | Interrupt Controller Monitor System Register Enable register |
| ICC_PMR[a] | c4 | 0 | c6 | 0 | 32-bit | RW | Interrupt Controller Interrupt Priority Mask Register |
| ICV_PMR[a] | | | | | | | Interrupt Controller Virtual Interrupt Priority Mask Register |
| ICC_RPR[a] | c12 | 0 | c11 | 3 | 32-bit | RO | Interrupt Controller Running Priority Register |
| ICV_RPR[a] | | | | | | | Interrupt Controller Virtual Running Priority Register |
| ICC_SGI0R | - | 2 | c12 | - | 64-bit | WO | Interrupt Controller Software Generated Interrupt group 0 Register |
| ICC_SGI1R | - | 0 | c12 | - | 64-bit | WO | Interrupt Controller Software Generated Interrupt group 1 Register |
| ICC_SRE | c12 | 0 | c12 | 5 | 32-bit | RW | Interrupt Controller System Register Enable register |
| ICH_AP0R0 | c12 | 4 | c8 | 0 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (0,0) |
| ICH_AP0R1 | c12 | 4 | c8 | 1 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (0,1) |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-66 Generic Interrupt Controller System registers (continued)**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| ICH_AP0R2 | c12 | 4 | c8 | 2 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (0,2) |
| ICH_AP0R3 | c12 | 4 | c8 | 3 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (0,3) |
| ICH_AP1R0 | c12 | 4 | c9 | 0 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (1,0) |
| ICH_AP1R1 | c12 | 4 | c9 | 1 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (1,1) |
| ICH_AP1R2 | c12 | 4 | c9 | 2 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (1,2) |
| ICH_AP1R3 | c12 | 4 | c9 | 3 | 32-bit | RW | Interrupt Controller Hyp Active Priorities Register (1,3) |
| ICH_EISR | c12 | 4 | c11 | 3 | 32-bit | RO | Interrupt Controller End of Interrupt Status Register |
| ICH_ELRSR | c12 | 4 | c11 | 5 | 32-bit | RO | Interrupt Controller Empty List Register Status Register |
| ICH_HCR | c12 | 4 | c11 | 0 | 32-bit | RW | Interrupt Controller Hyp Control Register |
| ICH_LR<n>, n==0-7 | c12 | 4 | c12 | 0-7 | 32-bit | RW | Interrupt Controller List Registers, 0-7 |
| ICH_LR<n>, n==8-15 | c12 | 4 | c13 | 0-7 | 32-bit | RW | Interrupt Controller List Registers, 8-15 |
| ICH_LRC<n>, n==0-7 | c12 | 4 | c14 | 0-7 | 32-bit | RW | Interrupt Controller List Registers Continuation, 0-7 |
| ICH_LRC<n>, n==8-15 | c12 | 4 | c15 | 0-7 | 32-bit | RW | Interrupt Controller List Registers Continuation, 8-15 |
| ICH_MISR | c12 | 4 | c11 | 2 | 32-bit | RO | Interrupt Controller Maintenance Interrupt State Register |
| ICH_VMCR | c12 | 4 | c11 | 7 | 32-bit | RW | Interrupt Controller Virtual Machine Control Register |
| ICH_VTR | c12 | 4 | c11 | 1 | 32-bit | RO | Interrupt Controller VGIC Type Register |

a. As described in this section, these ICC_* and ICV_* registers use the same encodings.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.18.16 Legacy feature registers, functional group

Table G4-67 shows the VMSAv8-32 CP15 Legacy features registers.

**Table G4-67 CP15 Legacy features registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Type[a] | Description |
|---|---|---|---|---|---|---|---|
| CP15DMB | c7 | 0 | c10 | 5 | 32-bit | WO, PL0 | *Memory barriers* on page E2-2375 |
| CP15DSB | c7 | 0 | c10 | 4 | 32-bit | WO, PL0 | |
| CP15ISB | c7 | 0 | c5 | 4 | 32-bit | WO, PL0 | |
| FCSEIDR | c13 | 0 | c0 | 0 | 32-bit | [b] | FCSE Process ID Register |

a. PL0 in a Type description indicates that the encoding is accessible by software executing at PL0. See the register description for more information.

b. In ARMv8, the PE does not implement the FCSEIDR, and therefore the register is RO. See the register description for more information.

Table G4-68 shows the VMSAv8-32 CP14 Legacy features registers.

**Table G4-68 CP14 Legacy features registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| JIDR | c0 | 7 | c0 | 0 | 32-bit | RO | Jazelle ID Register |
| JMCR | c2 | 7 | c0 | 0 | 32-bit | RW | Jazelle Main Configuration Register |
| JOSCR | c1 | 7 | c0 | 0 | 32-bit | RW | Jazelle OS Control Register |

### G4.18.17 IMPLEMENTATION DEFINED registers, functional group

VMSAv8-32 defines some encodings for registers with content that is entirely IMPLEMENTATION DEFINED. Table G4-69 shows these registers.

**Table G4-69 Architectural encodings for registers with IMPLEMENTATION DEFINED content**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| ACTLR | c1 | 0 | c0 | 1 | 32-bit | RW | Auxiliary Control Register |
| ACTLR2 | c1 | 0 | c0 | 3 | 32-bit | RW | Auxiliary Control Register 2 |
| ADFSR | c5 | 0 | c1 | 0 | 32-bit | RW | Auxiliary Data Fault Status Register |
| AIDR | c0 | 1 | c0 | 7 | 32-bit | RO | Auxiliary ID Register |
| AIFSR | c5 | 0 | c1 | 1 | 32-bit | RW | Auxiliary Instruction Fault Status Register |
| AMAIR0 | c10 | 0 | c3 | 0 | 32-bit | RW | Auxiliary Memory Attribute Indirection Register 0 |
| AMAIR1 | c10 | 0 | c3 | 1 | 32-bit | RW | Auxiliary Memory Attribute Indirection Register 1 |
| HACR | c1 | 4 | c1 | 7 | 32-bit | RW | Hyp Auxiliary Configuration Register |
| HACTLR | c1 | 4 | c0 | 0 | 32-bit | RW | Hyp Auxiliary System Control Register |
| HACTLR2 | c1 | 4 | c0 | 3 | 32-bit | RW | Hyp Auxiliary System Control Register 2 |
| HADFSR | c5 | 4 | c1 | 0 | 32-bit | RW | Hyp Auxiliary Data Fault Status Register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-69 Architectural encodings for registers with IMPLEMENTATION DEFINED content (continued)**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| HAIFSR | c5 | 4 | c1 | 1 | 32-bit | RW | Hyp Auxiliary Instruction Fault Status Register |
| HAMAIR0 | c10 | 4 | c3 | 0 | 32-bit | RW | Hyp Auxiliary Memory Attribute Indirection Register 0 |
| HAMAIR1 | c10 | 4 | c3 | 1 | 32-bit | RW | Hyp Auxiliary Memory Attribute Indirection Register 1 |
| REVIDR | c0 | 0 | c0 | 6 | 32-bit | RO | Revision ID Register |
| TCMTR | c0 | 0 | c0 | 2 | 32-bit | RO | TCM Type Register |

See also *IMPLEMENTATION DEFINED performance monitors* on page G4-4174.

### G4.18.18 Advanced SIMD and floating-point registers, functional group

Table G4-70 shows the VMSAv8-32 Floating-point System registers. These registers are accesses using `VMRS` and `VMSR` instructions, see the register descriptions for more information.

**Table G4-70 Floating-point registers**

| Name | Width | Type | Description |
|---|---|---|---|
| FPEXC | 32-bit | RW | Floating-Point Exception Control register |
| FPSCR | 32-bit | RW | Floating-Point Status and Control Register |
| FPSID | 32-bit | RW[a] | Floating-Point System ID register |
| MVFR0 | 32-bit | RO | Media and VFP Feature Register 0 |
| MVFR1 | 32-bit | RO | Media and VFP Feature Register 1 |
| MVFR2 | 32-bit | RO | Media and VFP Feature Register 2 |

a. When the FPSID is accessible, `VMSR` accesses to the FPSID are ignored.

### G4.18.19 Debug registers, functional group

In AArch32 state, most Debug registers that are accessible through the System registers interface use CP14 encodings, and are accessed with an opc1 value of 0. Table G4-71 shows these registers.

**Table G4-71 System register CP14 encodings of Debug registers**

| Name | CRn | opc2 | CRm | Width | Type | Description |
|---|---|---|---|---|---|---|
| DBGAUTHSTATUS | c7 | 6 | c14 | 32-bit | RO | Authentication Status |
| DBGBCR<n> | c0 | 5 | c0-c15 | 32-bit | RW | Breakpoint Control |
| DBGBVR<n> | c0 | 4 | c0-c15 | 32-bit | RW | Breakpoint Value |
| DBGBXVR<n> | c1 | 1 | c0-c15 | 32-bit | RW | Breakpoint Extended Value |
| DBGCLAIMCLR | c7 | 6 | c9 | 32-bit | RW | CLAIM Tag Clear |
| DBGCLAIMSET | c7 | 6 | c8 | 32-bit | RW | CLAIM Tag Set |
| DBGDCCINT | c0 | 0 | c2 | 32-bit | RW | Debug Communications Channel Interrupt Enable Register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-71 System register CP14 encodings of Debug registers (continued)**

| Name | CRn | opc2 | CRm | Width | Type | Description |
|---|---|---|---|---|---|---|
| DBGDEVID | c7 | 7 | c2 | 32-bit | RO | Device ID 0 |
| DBGDEVID1 | c7 | 7 | c1 | 32-bit | RO | Device ID 1 |
| DBGDEVID2 | c7 | 7 | c0 | 32-bit | RO | Reserved, UNK |
| DBGDIDR | c0 | 0 | c0 | 32-bit | RO | Debug ID |
| DBGDRAR | - | - | c1 | 64-bit | RO | Debug ROM Address |
| | c1 | 0 | c0 | 32-bit | | |
| DBGDSAR | - | - | c2 | 64-bit | RO | Debug Self Address Offset |
| | c2 | 0 | c0 | 32-bit | | |
| DBGDSCRext | c0 | 2 | c2 | 32-bit | RW | Debug Status and Control external |
| DBGDSCRint | c0 | 0 | c1 | 32-bit | RO | Debug Status and Control internal |
| DBGDTRRXext | c0 | 2 | c0 | 32-bit | RW | Host to Target Data Transfer external |
| DBGDTRRXint | c0 | 0 | c5 | 32-bit | RO | Host to Target Data Transfer internal |
| DBGDTRTXext | c0 | 2 | c3 | 32-bit | RW | Target to Host Data Transfer external |
| DBGDTRTXint | c0 | 0 | c5 | 32-bit | WO | Target to Host Data Transfer internal |
| DBGOSDLR | c1 | 4 | c3 | 32-bit | RW | OS Double Lock |
| DBGOSECCR | c0 | 2 | c6 | 32-bit | RW | OS Lock Exception Catch Control Register |
| DBGOSLAR | c1 | 4 | c0 | 32-bit | WO | OS Lock Access |
| DBGOSLSR | c1 | 4 | c1 | 32-bit | RO | OS Lock Status |
| DBGPRCR | c1 | 4 | c4 | 32-bit | RW | Device Powerdown and Reset Control |
| DBGVCR | c0 | 0 | c7 | 32-bit | RW | Vector Catch |
| DBGWCR<n> | c0 | 7 | c0-c15 | 32-bit | RW | Watchpoint Control |
| DBGWFAR | c0 | 0 | c6 | 32-bit | RW | Watchpoint Fault Address |
| DBGWVR<n> | c0 | 6 | c0-c15 | 32-bit | RW | Watchpoint Value |
| - | c4 | 0-3 | c0-c15 | 32-bit | IMP DEF | IMPLEMENTATION DEFINED |
| - | c7 | 2-3 | c0-c15 | 32-bit | IMP DEF | Integration registers |
| | | 4 | c0 | 32-bit | IMP DEF | |

In AArch32 state, some Debug registers that are accessible through the System registers interface use CP15 encodings. Table G4-72 on page G4-4182 shows these registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table G4-72 System register CP15 encodings of Debug registers**

| Name | CRn | opc1 | CRm | opc2 | Width | Type | Description |
|---|---|---|---|---|---|---|---|
| DLR | c4 | 3 | c5 | 1 | 32-bit | RW | Debug Link Register |
| DSPSR | c4 | 3 | c5 | 0 | 32-bit | RW | Debug Saved Program Status Register |
| HDCR | c1 | 4 | c1 | 1 | 32-bit | RW | Hyp Debug Control Register |
| SDCR | c1 | 0 | c3 | 1 | 32-bit | RW | Secure Debug Configuration Register |
| SDER | c1 | 0 | c1 | 1 | 32-bit | RW | Secure Debug Enable Register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G4.19 Pseudocode description of VMSAv8-32 memory system operations

This section contains pseudocode describing VMSAv8-32 memory operations. The following subsections describe the pseudocode functions:

- *Alignment fault*.
- *Address translation*.
- *Domain checking* on page G4-4185.
- *TLB operations* on page G4-4185.
- *Translation table walk* on page G4-4186.
- *Reporting syndrome information* on page G4-4197.
- *Memory access decode when TEX remap is enabled* on page G4-4199.

See also the pseudocode for general memory system operations in *Pseudocode description of general memory system instructions* on page G3-3979.

### G4.19.1 Alignment fault

The `AlignmentFault()` pseudocode function describes the generation of an Alignment fault Data Abort exception:

```
// AArch32.AlignmentFault()
// ========================

FaultRecord AArch32.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch32.CreateFaultRecord(Fault_Alignment, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

See also *Abort exceptions* on page G3-3985.

### G4.19.2 Address translation

The `TranslateAddress()` and `FullTranslate()` pseudocode functions describe a VMSAv8-32 address translation.

```
// AArch32.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        return AArch64.TranslateAddress(ZeroExtend(vaddress, 64), acctype, iswrite, wasaligned,
                                        size);

    result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
```

The `FullTranslate()` function calls either:

- The function described in *Address translation when the stage 1 address translation is disabled* on page G4-4184.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- One of the functions described in *Translation table walk* on page G4-4186.

```
// AArch32.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch32.FullTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch32.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !IsFault(S1) && HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        s2fs1walk = FALSE;
        result = AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                              size);
    else
        result = S1;

    return result;
```

*Stage 2 translation table walk* on page G4-4194 describes the `CheckS2Permission()` and `CombineS1S2Desc()` pseudocode functions.

## Address translation when the stage 1 address translation is disabled

The `TranslateAddressS1Off()` pseudocode function describes the address translation performed when the stage 1 address translation is disabled.

```
// AArch32.TranslateAddressS1Off()
// ===============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch32.TranslateAddressS1Off(bits(32) vaddress, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
        default_cacheable = (dc == '1');
    else
        default_cacheable = FALSE;

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.type = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;      // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
    elsif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.type = MemType_Device;
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
    else
        // Instruction cacheability controlled by SCTLR/HSCTLR.I
        if PSTATE.EL == EL2 then
            cacheable = HSCTLR.I == '1';
        else
            cacheable = SCTLR.I == '1';
        result.addrdesc.memattrs.type = MemType_Normal;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            result.addrdesc.memattrs.inner.hints = MemHint_No;
        result.addrdesc.memattrs.shareable = TRUE;
        result.addrdesc.memattrs.outershareable = TRUE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.physicaladdress = ZeroExtend(vaddress);
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch32.NoFault();

    return result;
```

### G4.19.3 Domain checking

The `CheckDomain()` pseudocode function describes domain checking:

```
// AArch32.CheckDomain()
// =====================

(boolean, FaultRecord) AArch32.CheckDomain(bits(4) domain, bits(32) vaddress, integer level,
                                           AccType acctype, boolean iswrite)

    index = 2 * UInt(domain);
    attrfield = DACR<index+1:index>;

    if attrfield == '10' then          // Reserved, maps to an allocated value
        // Reserved value maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();

    if attrfield == '00' then
        fault = AArch32.DomainFault(domain, level, acctype, iswrite);
    else
        fault = AArch32.NoFault();

    permissioncheck = (attrfield == '01');

    return (permissioncheck, fault);
```

### G4.19.4 TLB operations

The `TLBRecord` type represents the contents of a TLB entry:

```
type TLBRecord is (
    Permissions       perms,
    bit               nG,             // '0' = Global, '1' = not Global
    bits(4)           domain,         // AArch32 only
    boolean           contiguous,     // Contiguous bit from page table
    integer           level,          // In AArch32 Short-descriptort format, indicates Section/Page
    integer           blocksize,      // Describes size of memory translated in KBytes
    AddressDescriptor addrdesc
)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.19.5 Translation table walk

Because of the complexity of a translation table walk, the following sections describe the different cases:

- *Translation table walk using the Short-descriptor translation table format for stage 1*.
- *Translation table walk using the Long-descriptor translation table format for stage 1* on page G4-4189.
- *Stage 2 translation table walk* on page G4-4194.

#### Translation table walk using the Short-descriptor translation table format for stage 1

The `TranslationTableWalkSD()` pseudocode function describes the translation table walk when the stage 1 translation tables use the Short-descriptor format. It calls the function described in *Stage 2 translation table walk* on page G4-4194 if necessary:

```
// AArch32.TranslationTableWalkSD()
// ================================
// Returns a result of a translation table walk using the Short-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkSD(bits(32) vaddress, AccType acctype, boolean iswrite,
                                         integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    // This is only called when address translation is enabled
    TLBRecord         result;
    AddressDescriptor l1descaddr;
    AddressDescriptor l2descaddr;
    bits(40)      outputaddress;

    // Variables for Abort functions
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    // Default setting of the domain
    domain = bits(4) UNKNOWN;

    // Determine correct Translation Table Base Register to use.
    bits(64) ttbr;
    n = UInt(TTBCR.N);
    if n == 0 || IsZero(vaddress<31:(32-n)>) then
        ttbr = TTBR0;
        disabled = (TTBCR.PD0 == '1');
    else
        ttbr = TTBR1;
        disabled = (TTBCR.PD1 == '1');
        n = 0;  // TTBR1 translation always works like N=0 TTBR0 translation

    // Check this Translation Table Base Register is not disabled.
    if disabled then
        level = 1;
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    // Obtain descriptor from initial lookup.
    l1descaddr.paddress.physicaladdress = ZeroExtend(ttbr<31:14-n>:vaddress<31-n:20>:'00');
    l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
    IRGN = ttbr<0>:ttbr<6>;             // TTBR.IRGN
    RGN = ttbr<4:3>;                    // TTBR.RGN
    SH = ttbr<1>:ttbr<5>;               // TTBR.S:TTBR.NOS
    l1descaddr.memattrs = WalkAttrDecode(SH, RGN, IRGN);

    if !HaveEL(EL2) || IsSecure()  then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // if only 1 stage of translation
    l1descaddr2 = l1descaddr;
else
    l1descaddr2 = AArch32.SecondStageWalk(l1descaddr, vaddress, acctype, iswrite, 4);
    // Check for a fault on the stage 2 walk
    if IsFault(l1descaddr2) then
        result.addrdesc.fault = l1descaddr2.fault;
        return result;

// Update virtual address for abort functions
l1descaddr2.vaddress = ZeroExtend(vaddress);

l1desc = _Mem[l1descaddr2, 4, AccType_PTW];
if SCTLR.EE == '1' then l1desc = BigEndianReverse(l1desc);

// Process descriptor from initial lookup.
case l1desc<1:0> of
    when '00'                                                 // Fault, Reserved
        level = 1;
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    when '01'                                                 // Large page or Small page
        domain = l1desc<8:5>;
        level = 2;
        pxn = l1desc<2>;
        NS = l1desc<3>;

        // Obtain descriptor from level 2 lookup.
        l2descaddr.paddress.physicaladdress = ZeroExtend(l1desc<31:10>:vaddress<19:12>:'00');
        l2descaddr.paddress.NS = if IsSecure() then '0' else '1';
        l2descaddr.memattrs = l1descaddr.memattrs;

        if !HaveEL(EL2) || IsSecure()  then
         // if only 1 stage of translation
            l2descaddr2 = l2descaddr;
        else
            l2descaddr2 = AArch32.SecondStageWalk(l2descaddr, vaddress, acctype, iswrite, 4);
            // Check for a fault on the stage 2 walk
            if IsFault(l2descaddr2) then
                result.addrdesc.fault = l2descaddr2.fault;
                return result;

        // Update virtual address for abort functions
        l2descaddr2.vaddress = ZeroExtend(vaddress);

        l2desc = _Mem[l2descaddr2, 4, AccType_PTW];
        if SCTLR.EE == '1' then l2desc = BigEndianReverse(l2desc);

        // Process descriptor from level 2 lookup.
        if l2desc<1:0> == '00' then
            result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        nG = l2desc<11>;
        S = l2desc<10>;
        ap = l2desc<9,5:4>;

        if SCTLR.AFE == '1' && l2desc<4> == '0' then
            // Hardware access to the Access Flag is not supported in ARMv8
            result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                            iswrite, secondstage, s2fs1walk);
            return result;

        if l2desc<1> == '0' then                              // Large page
            xn = l2desc<15>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                tex = l2desc<14:12>;
                c = l2desc<3>;
                b = l2desc<2>;
                blocksize = 64;
                outputaddress = ZeroExtend(l2desc<31:16>:vaddress<15:0>);
            else                                                    // Small page
                tex = l2desc<8:6>;
                c = l2desc<3>;
                b = l2desc<2>;
                xn = l2desc<0>;
                blocksize = 4;
                outputaddress = ZeroExtend(l2desc<31:12>:vaddress<11:0>);

        when '1x'                                                   // Section or Supersection
            NS = l1desc<19>;
            nG = l1desc<17>;
            S = l1desc<16>;
            ap = l1desc<15,11:10>;
            tex = l1desc<14:12>;
            xn = l1desc<4>;
            c = l1desc<3>;
            b = l1desc<2>;
            pxn = l1desc<0>;
            level = 1;

            if SCTLR.AFE == '1' && l1desc<10> == '0' then
                // Hardware management of the Access Flag is not supported in ARMv8
                result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                                iswrite, secondstage, s2fs1walk);
                return result;

            if l1desc<18> == '0' then                               // Section
                domain = l1desc<8:5>;
                blocksize = 1024;
                outputaddress = ZeroExtend(l1desc<31:20>:vaddress<19:0>);
            else                                                    // Supersection
                domain = '0000';
                blocksize = 16384;
                outputaddress = l1desc<8:5>:l1desc<23:20>:l1desc<31:24>:vaddress<23:0>;

    // Decode the TEX, C, B and S bits to produce the TLBRecord's memory attributes
    if SCTLR.TRE == '0' then
        if RemapRegsHaveResetValues() then
            result.addrdesc.memattrs = AArch32.DefaultTEXDecode(tex, c, b, S, acctype);
        else
            result.addrdesc.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        result.addrdesc.memattrs = AArch32.RemappedTEXDecode(tex, c, b, S, acctype);

    // Set the rest of the TLBRecord, try to add it to the TLB, and return it.
    result.perms.ap = ap;
    result.perms.xn = xn;
    result.perms.pxn = pxn;
    result.nG = nG;
    result.domain = domain;
    result.level = level;
    result.blocksize = blocksize;
    result.addrdesc.paddress.physicaladdress = ZeroExtend(outputaddress);
    result.addrdesc.paddress.NS = if IsSecure() then NS else '1';
    result.addrdesc.fault = AArch32.NoFault();

    return result;
```

The `ShortConvertAttrsHints()` pseudocode function converts the Normal memory cacheability attribute, from the TTBR or the translation table TEX field, into the separate cacheability attribute and cache allocation hint defined in a Long-descriptor translation table descriptor:

```
// ShortConvertAttrsHints()
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// ========================
// Converts the short attribute fields for Normal memory as used in the TTBR and
// TEX fields to orthogonal attributes and hints

MemAttrHints ShortConvertAttrsHints(bits(2) RGN, AccType acctype)

    MemAttrHints result;

    if S1CacheDisabled(acctype) then     // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case RGN of
            when '00'                   // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '01'                   // Write-back, Read and Write allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;
            when '10'                   // Write-through, Read allocate
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RA;
            when '11'                   // Write-back, Read allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RA;

    result.transient = FALSE;

    return result;
```

### Translation table walk using the Long-descriptor translation table format for stage 1

The `TranslationTableWalkLD()` pseudocode function describes the translation table walk when the stage 1 translation tables use the Long-descriptor format. It calls the function described in *Stage 2 translation table walk* on page G4-4194 if necessary:

```
// AArch32.TranslationTableWalkLD()
// ================================
// Returns a result of a translation table walk using the Long-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkLD(bits(40) ipaddress, bits(32) vaddress,
                                         AccType acctype, boolean iswrite, boolean secondstage,
                                         boolean s2fs1walk, integer size)
    if !secondstage then
        assert ELUsingAArch32(S1TranslationRegime());
    else
        assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && PSTATE.EL != EL2;

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(40) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2
    domain = bits(4) UNKNOWN;

    descaddr.memattrs.type = MemType_Normal;

    // Fixed parameters for the page table walk:
    //  grainsize = Log2(Size of Table)         - Size of Table is 4KB in AArch32
    //  stride = Log2(Address per Level)        - Bits of address consumed at each level
    constant integer grainsize = 12;                    // Log2(4KB page size)
    constant integer stride = grainsize - 3;            // Log2(page size / 8 bytes)

    // Derived parameters for the page table walk:
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
//  inputsize = Log2(Size of Input Address) - Input Address size in bits
//  level = Level to start walk from
// This means that the number of levels after start level = 3-level

if !secondstage then
    // First stage translation
    inputaddr = ZeroExtend(vaddress);
    if PSTATE.EL == EL2 then
        inputsize = 32 - UInt(HTCR.T0SZ);
        basefound = inputsize == 32 || IsZero(inputaddr<31:inputsize>);
        disabled = FALSE;
        baseregister = HTTBR;
        descaddr.memattrs = WalkAttrDecode(HTCR.SH0, HTCR.ORGN0, HTCR.IRGN0);
        reversedescriptors = HSCTLR.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;

    else
        basefound = FALSE;
        disabled = FALSE;
        t0size = UInt(TTBCR.T0SZ);
        if t0size == 0 || IsZero(inputaddr<31:(32-t0size)>) then
            inputsize = 32 - t0size;
            basefound = TRUE;
            disabled = TTBCR.EPD0 == '1';
            baseregister = TTBR0;
            descaddr.memattrs = WalkAttrDecode(TTBCR.SH0, TTBCR.ORGN0, TTBCR.IRGN0);
        t1size = UInt(TTBCR.T1SZ);
        if (t1size == 0 && !basefound) || (t1size > 0 && IsOnes(inputaddr<31:(32-t1size)>)) then
            inputsize = 32 - t1size;
            basefound = TRUE;
            disabled = TTBCR.EPD1 == '1';
            baseregister = TTBR1;
            descaddr.memattrs = WalkAttrDecode(TTBCR.SH1, TTBCR.ORGN1, TTBCR.IRGN1);
        reversedescriptors = SCTLR.EE == '1';
        lookupsecure = IsSecure();
        singlepriv = FALSE;

    // The starting level is the number of strides needed to consume the input address
    level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));

else
    // Second stage translation
    inputaddr = ipaddress;
    inputsize = 32 - SInt(VTCR.T0SZ);
    // VTCR.S must match VTCR.T0SZ[3]
    if VTCR.S != VTCR.T0SZ<3> then
        (-, inputsize) = ConstrainUnpredictableInteger(32-7, 32+8);
    basefound = inputsize == 40 || IsZero(inputaddr<39:inputsize>);
    disabled = FALSE;
    baseregister = VTTBR;
    descaddr.memattrs = WalkAttrDecode(VTCR.IRGN0, VTCR.ORGN0, VTCR.SH0);
    reversedescriptors = HSCTLR.EE == '1';
    lookupsecure = FALSE;
    singlepriv = TRUE;

    startlevel = UInt(VTCR.SL0);
    level = 2 - startlevel;
    if level <= 0 then basefound = FALSE;

    // Number of entries in the starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    startsizecheck = inputsize - ((3 - level)*stride + grainsize); // Log2(Num of entries)

    // Check for starting level table with fewer than 2 entries or longer than 16 pages.
    // Lower bound check is:  startsizecheck < Log2(2 entries)
    //   That is, VTCR.SL0 == '00' and SInt(VTCR.T0SZ) > 1, Size of Input Address < 2^31 bytes
    // Upper bound check is:  startsizecheck > Log2(pagesize/8*16)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    //   That is, VTCR.SL0 == '01' and SInt(VTCR.T0SZ) < -2, Size of Input Address > 2^34 bytes
    if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

if !basefound || disabled then
    level = 1;           // AArch64 reports this as a level 0 fault
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

if !IsZero(baseregister<47:40>) then
    level = 0;
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

// Bottom bound of the Base address is:
//     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
//     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
baseaddress = baseregister<39:baselowerbound>:Zeros(baselowerbound);

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

addrselecttop = inputsize - 1;

repeat
    addrselectbottom = (3-level)*stride + grainsize;

    bits(40) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
    descaddr.paddress.physicaladdress = ZeroExtend(baseaddress OR index);
    descaddr.paddress.NS = ns_table;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if !HaveEL(EL2) || secondstage || IsSecure() || PSTATE.EL == EL2 then
        descaddr2 = descaddr;
    else
        descaddr2 = AArch32.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
        // Check for a fault on the stage 2 walk
        if IsFault(descaddr2) then
            result.addrdesc.fault = descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    descaddr2.vaddress = ZeroExtend(vaddress);

    desc = _Mem[descaddr2, 8, AccType_PTW];
    if reversedescriptors then desc = BigEndianReverse(desc);

    if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
         // Fault (00), Reserved (10), or Block (01) at level 3
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Valid Block, Page, or Table entry
    if desc<1:0> == '01' || level == 3 then                 // Block (01) or Page (11)
        blocktranslate = TRUE;
    else                                                    // Table (11)
        if !IsZero(desc<47:40>) then
            result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        baseaddress = desc<39:grainsize>:Zeros(grainsize);

        if !secondstage then
            // Unpack the upper and lower table attributes
            ns_table    = ns_table    OR desc<63>;
            ap_table<1> = ap_table<1> OR desc<62>;       // read-only
            xn_table    = xn_table    OR desc<60>;
            // pxn_table and ap_table[0] apply only in EL1&0 translation regimes
            if !singlepriv then
                ap_table<0> = ap_table<0> OR desc<61>;   // privileged
                pxn_table   = pxn_table   OR desc<59>;

        level = level + 1;
        addrselecttop = addrselectbottom - 1;
        blocktranslate = FALSE;
until blocktranslate;

// Check the output address is inside the supported range
if !IsZero(desc<47:40>) then
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype,
                                                     iswrite, secondstage, s2fs1walk);
    return result;

// Check the access flag
if desc<10> == '0' then
    result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);
    return result;

// Unpack the descriptor into address and upper and lower block attributes
outputaddress = desc<39:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
xn = desc<54>;
pxn = desc<53>;
contiguousbit = desc<52>;
nG = desc<11>;
sh = desc<9:8>;
ap = desc<7:6>:'1';
memattr = desc<5:2>;                                    // AttrIndx and NS bit in stage 1

result.domain = bits(4) UNKNOWN;                        // Domains not used
result.level = level;
result.blocksize = 2^((3-level)*stride + grainsize);

// Stage 1 translation regimes also inherit attributes from the tables
if !secondstage then
    result.perms.xn      = xn OR xn_table;
    result.perms.ap<2>   = ap<2> OR ap_table<1>;        // Force read-only

    // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
    if !singlepriv then
        result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
        result.perms.pxn   = pxn OR pxn_table;
        // Pages from Non-secure tables are marked non-global in Secure EL1&0
        if IsSecure() then
            result.nG = nG OR ns_table;
        else
            result.nG = nG;
    else
        result.perms.ap<1> = '1';
        result.perms.pxn   = '0';
        result.nG          = '0';
    result.perms.ap<0>   = '1';
    result.addrdesc.memattrs = AArch32.S1AttrDecode(sh, memattr<2:0>, acctype);
    result.addrdesc.paddress.NS = memattr<3> OR ns_table;
else
    result.perms.ap<2:1> = ap<2:1>;
    result.perms.ap<0>   = '1';
    result.perms.xn      = xn;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result.perms.pxn     = '0';
        result.nG            = '0';
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
        result.addrdesc.paddress.NS = '1';

    result.addrdesc.paddress.physicaladdress = ZeroExtend(outputaddress);
    result.addrdesc.fault = AArch32.NoFault();
    result.contiguous = contiguousbit == '1';

    return result;
```

This function calls the `ConvertAttrsHints()` pseudocode function that is defined in *Translation table walk using the Short-descriptor translation table format for stage 1* on page G4-4186.

The `S1AttrDecode()` pseudocode function uses the MAIR0 and MAIR1 registers to decode the Attr[2:0] value from a stage 1 translation table descriptor:

```
// AArch32.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch32.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;

    if PSTATE.EL == EL2 then
        mair = HMAIR1:HMAIR0;
    else
        mair = MAIR1:MAIR0;
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();

    if attrfield<7:4> == '0000' then            // Device
        memattrs.type = MemType_Device;
        case attrfield<3:0> of
            when '0000'  memattrs.device = DeviceType_nGnRnE;
            when '0100'  memattrs.device = DeviceType_nGnRE;
            when '1000'  memattrs.device = DeviceType_nGRE;
            when '1100'  memattrs.device = DeviceType_GRE;
            otherwise    Unreachable();         // Reserved, handled above

    elsif attrfield<3:0> != '0000'  then        // Normal
        memattrs.type = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';

    else
        Unreachable();                          // Reserved, handled above

    return MemAttrDefaults(memattrs);
```

The `S2AttrDecode()` pseudocode function decodes the Attr[3:0] value from a stage 2 translation table descriptor:

```
// S2AttrDecode()
// ==============
// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)

    MemoryAttributes memattrs;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if attr<3:2> == '00' then                    // Device
        mematttrs.type = MemType_Device;
        case attr<1:0> of
            when '00'  mematttrs.device = DeviceType_nGnRnE;
            when '01'  mematttrs.device = DeviceType_nGnRE;
            when '10'  mematttrs.device = DeviceType_nGRE;
            when '11'  mematttrs.device = DeviceType_GRE;

    elsif attr<1:0> != '00' then                // Normal
        mematttrs.type = MemType_Normal;
        mematttrs.outer = S2ConvertAttrsHints(attr<3:2>, acctype);
        mematttrs.inner = S2ConvertAttrsHints(attr<1:0>, acctype);
        mematttrs.shareable = SH<1> == '1';
        mematttrs.outershareable = SH == '10';

    else
        mematttrs = MemoryAttributes UNKNOWN;    // Reserved

    return MemAttrDefaults(mematttrs);
```

### Stage 2 translation table walk

In the Non-secure EL1&0 translation regime, a descriptor address returned by stage 1 lookup is in the IPA address space, and must be mapped to a PA by a stage 2 translation. When EL2 is using AArch32, function `AArch32.SecondStageWalk()` performs this translation, by calling the `AArch32.SecondStageTranslate()` function. When called from `AArch32.SecondStageWalk()`, the.`AArch32.SecondStageTranslate()` function performs a second stage translation, from IPA to PA, of the supplied address, including checking that the access has read permission at the second stage. If the access does not have second stage read permission it generates a second stage Permission fault on the first stage translation table walk. The second stage translation might hit in a TLB, or might involve a translation table walk, which will use the algorithm described in this section. Stage 2 translations tables always use the Long-descriptor translation table format.

```
// AArch32.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch32.SecondStageWalk(AddressDescriptor S1, bits(32) vaddress, AccType acctype,
                                          boolean iswrite, integer size)
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                        size);
```

The `SecondStageTranslate()` pseudocode function performs the stage 2 translation table walk.

```
// AArch32.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.SecondStageTranslate(AddressDescriptor S1, bits(32) vaddress,
                                               AccType acctype, boolean iswrite, boolean wasaligned,
                                               boolean s2fs1walk, integer size)
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert IsZero(S1.paddress.physicaladdress<47:40>);

    if !ELUsingAArch32(EL2) then
        return AArch64.SecondStageTranslate(S1, ZeroExtend(vaddress, 64), acctype, iswrite,
                                            wasaligned, s2fs1walk, size);

    s2_enabled = HCR.VM == '1' || HCR.DC == '1';
    secondstage = TRUE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if s2_enabled then                          // Second stage enabled
        ipaddress = S1.paddress.physicaladdress<39:0>;
        S2 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                            s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if (!wasaligned && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype != AccType_IFETCH) then
            S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch32.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                          acctype, iswrite, s2fs1walk);

        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype == AccType_IFETCH) then
            domain = bits(4) UNKNOWN;
            S2.addrdesc = AArch32.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    domain, acctype, iswrite,
                                                    secondstage, s2fs1walk);

        // Check for protected table walk
        if (s2fs1walk && !IsFault(S2.addrdesc) && HCR.PTW == '1' &&
            S2.addrdesc.memattrs.type == MemType_Device) then
            domain = bits(4) UNKNOWN;
            S2.addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, S2.level, acctype,
                                                        iswrite, secondstage, s2fs1walk);
        result = CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

    return result;
```

The `CheckPermission()` pseudocode function checks the access permissions for the stage 1 translation.

```
// AArch32.CheckPermission()
// =========================
// Function used for permission checking from AArch32 stage 1 translations

FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level,
                                    bits(4) domain, bit NS, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());

    if PSTATE.EL != EL2 then
        wxn = SCTLR.WXN == '1';
        if TTBCR.EAE == '1' || SCTLR.AFE == '1' || perms.ap<0> == '1' then
            priv_r = TRUE;
            priv_w = perms.ap<2> == '0';
            user_r = perms.ap<1> == '1';
            user_w = perms.ap<2:1> == '01';
        else
            priv_r = perms.ap<2:1> != '00';
            priv_w = perms.ap<2:1> == '01';
            user_r = perms.ap<1> == '1';
            user_w = FALSE;
        uwxn = SCTLR.UWXN == '1';
        user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
        priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
                   (priv_w && wxn) || (user_w && uwxn));
        ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // Access from EL2
        wxn = HSCTLR.WXN == '1';
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' then
        secure_instr_fetch = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;
        if secure_instr_fetch == '1' then xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(40) UNKNOWN;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch32.NoFault();
```

The CheckS2Permission() pseudocode function checks the access permissions for the stage 2 translation.

```
// AArch32.CheckS2Permission()
// ===========================
// Function used for permission checking from AArch32 stage 2 translations

FaultRecord AArch32.CheckS2Permission(Permissions perms, bits(32) vaddress, bits(40) ipaddress,
                                      integer level, AccType acctype, boolean iswrite,
                                      boolean s2fs1walk)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && PSTATE.EL != EL2;

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    xn = !r || perms.xn == '1';

    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
    elsif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch32.NoFault();
```

The CombineS1S2Desc() pseudocode function combines the stage 1 and stage 2 access descriptors:

```
// CombineS1S2Desc()
// =================
// Combines the address descriptors from stage 1 and stage 2
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AddressDescriptor CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    AddressDescriptor result;

    result.paddress = s2desc.paddress;

    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    elsif s2desc.memattrs.type == MemType_Device || s1desc.memattrs.type == MemType_Device then
        result.memattrs.type = MemType_Device;
        if s1desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s2desc.memattrs.device;
        elsif s2desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s1desc.memattrs.device;
        else                    // Both Device
            result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
                                                       s2desc.memattrs.device);
    else                        // Both Normal
        result.memattrs.type = MemType_Normal;
        result.memattrs.device = DeviceType UNKNOWN;
        result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner);
        result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer);
        result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
        result.memattrs.outershareable = (s1desc.memattrs.outershareable ||
                                          s2desc.memattrs.outershareable);

    result.memattrs = MemAttrDefaults(result.memattrs);

    return result;
```

### G4.19.6 Reporting syndrome information

The `ReportHypEntry()`, `ReportDataAbort()`, and `ReportPrefetchAbort()` pseudocode functions write syndrome value information to the appropriate registers for the current mode.

```
// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception type = exception.type;

    (ec,il) = AArch32.ExceptionClass(type);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if type IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif type == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// AArch32.ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

    // If "route_to_monitor" is TRUE then use the Secure Banked system registers because EL3 is
    // implemented and using AArch32, and the Abort might be taken from Non-secure state.
    // Otherwise use the system registers for the current security state, which might not be banked.
    eae = (if route_to_monitor then TTBCR_S.EAE else TTBCR.EAE);
    d_side = TRUE;
    if eae == '1' then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType_IC then
        if (eae == '0' &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;



// AArch32.ReportPrefetchAbort()
// =============================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

    // If "route_to_monitor" is TRUE then use the Secure Banked system registers because EL3 is
    // implemented and using AArch32, and the Abort might be taken from Non-secure state.
    // Otherwise use the system registers for the current security state, which might not be banked.
    eae = (if route_to_monitor then TTBCR_S.EAE else TTBCR.EAE);
    d_side = FALSE;
    if eae == '1' then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G4.19.7 Memory access decode when TEX remap is enabled

When using the Short-descriptor translation table format, the function `RemappedTEXDecode()` decodes the `texcb` and `S` attributes derived from the translation tables when TEX remap is enabled. *Short-descriptor format memory region attributes, with TEX remap* on page G4-4049 shows the interpretation of the arguments.

```
// AArch32.RemappedTEXDecode()
// ===========================

MemoryAttributes AArch32.RemappedTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes mematTrs;

    region = UInt(TEX<0>:C:B);         // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        mematTrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        base = 2 * region;
        attrfield = PRRR<base+1:base>;

        if attrfield == '11' then      // Reserved, maps to allocated value
            (-, attrfield) = ConstrainUnpredictableBits();

        case attrfield of
            when '00'                  // Device-nGnRnE
                mematTrs.type = MemType_Device;
                mematTrs.device = DeviceType_nGnRnE;
            when '01'                  // Device-nGnRE
                mematTrs.type = MemType_Device;
                mematTrs.device = DeviceType_nGnRE;
            when '10'
                mematTrs.type = MemType_Normal;
                mematTrs.inner = ShortConvertAttrsHints(NMRR<base+1:base>, acctype);
                mematTrs.outer = ShortConvertAttrsHints(NMRR<base+17:base+16>, acctype);
                s_bit = if S == '0' then PRRR.NS0 else PRRR.NS1;
                mematTrs.shareable = (s_bit == '1');
                mematTrs.outershareable = (s_bit == '1' && PRRR<region+24> == '0');
            when '11'
                Unreachable();

    // transient bits are not supported in this format
    mematTrs.inner.transient = FALSE;
    mematTrs.outer.transient = FALSE;

    return MemAttrDefaults(mematTrs);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter G5
# The Generic Timer in AArch32 state

This chapter describes the implementation of the ARM Generic Timer as an extension to an ARMv8 implementation. It includes an overview of the AArch32 System register interface to an ARM Generic Timer.

It contains the following section:


- *About the Generic Timer in AArch32 state* on page G5-4202.


Chapter D6 *The Generic Timer in AArch64 state* describes the AArch64 view of the Generic Timer, including an additional timer that can be implemented in AArch64 state, and Chapter I1 *System Level Implementation of the Generic Timer* describes the system level implementation of the Generic Timer.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G5.1 About the Generic Timer in AArch32 state

Figure G5-1 shows an example system-on-chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The *ARM Generic Interrupt Controller Architecture Specification* defines a possible architecture for the interrupt controllers.
- Generic Timer functionality is distributed across multiple components.

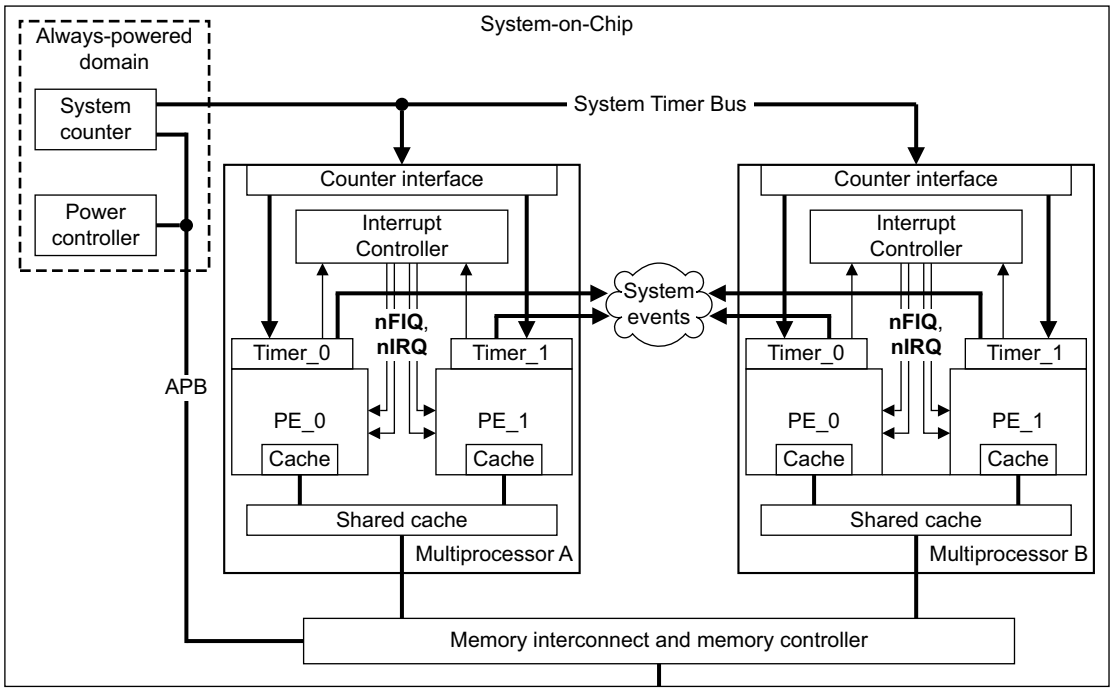



**Figure G5-1 Generic Timer example**

The Generic Timer:

- Provides a system counter, that measures the passing of time in real-time.

  **Note**

  The Generic Timer can also provide other components at a system level, but Figure G5-1 does not show any such components.

- Supports *virtual counters* that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.
- Timers, that can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

This chapter describes an instance of the Generic Timer component that Figure G5-1 shows as Timer_0 or Timer_1 within the Multiprocessor A or Multiprocessor B block. This component can be accessed from AArch64 state or AArch32 state, and this chapter describes access from AArch32 state. Chapter D6 *The Generic Timer in AArch64 state* describes access to this component from AArch64 state.

**Note**

The reset requirements of Generic Timer registers are more strict when they are accessed from AArch32 state than when they are accessed from AArch64 state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A Generic Timer implementation must also include a memory-mapped system component. This component:

- Must provide the System counter shown in Figure G5-1 on page G5-4202.
- Optionally, can provide timer components for use at a system level.

Chapter I1 *System Level Implementation of the Generic Timer* describes this memory-mapped component.

### G5.1.1 System counter

The Generic Timer provides a system counter with the following specification:

| | |
|---|---|
| **Width** | At least 56 bits wide. |
| | The value returned by any 64-bit read of the counter is zero-extended to 64 bits. |
| **Frequency** | Increments at a fixed frequency, typically in the range 1-50MHz. |
| | Can support one or more alternative operating modes in which it increments by larger amounts at a lower frequency, typically for power-saving. |
| **Roll-over** | Roll-over time of not less than 40 years. |
| **Accuracy** | ARM does not specify a required accuracy, but recommends that the counter does not gain or lose more than ten seconds in a 24-hour period. |
| | Use of lower-frequency modes must not affect the implemented accuracy. |
| **Start-up** | Starts operating from zero. |

The system counter must provide a uniform view of system time. More precisely, it must be impossible for the following sequence of events to show system time going backwards:

1. Device A reads the time from the system counter.
2. Device A communicates with another agent in the system, Device B.
3. After recognizing the communication from Device A, Device B reads the time from the system counter.

The system counter must be implemented in an always-on power domain.

To support lower-power operating modes, the counter can increment by larger amounts at a lower frequency. For example, a 10MHz system counter might either increment:

- By 1 at 10MHz.
- By 500 at 20kHz, when the system lowers the clock frequency, to reduce power consumption.

In this case, the counter must support transitions between high-frequency, high-precision operation, and lower-frequency, lower-precision operation, without any impact on the required accuracy of the counter.

The CNTFRQ register is intended to hold a copy of the current clock frequency to allow fast reference to this frequency by software running on the PE. For more information see *Initializing and reading the system counter frequency*.

The mechanism by which the count from the system counter is distributed to system components is IMPLEMENTATION DEFINED, but each PE with a System register interface to the system counter must have a counter input that can capture each increment of the counter.

**Note**

So that the system counter can be clocked independently from the PE hardware, the count value might be distributed using a Gray code sequence. *Gray-count scheme for timer distribution scheme* on page I1-5110 gives more information about this possibility.

#### Initializing and reading the system counter frequency

The CNTFRQ register must be programmed to the clock frequency of the system counter. Typically, this is done only during the system boot process, by using the System register interface to write the system counter frequency to the CNTFRQ register. Only software executing at the highest implemented Exception level can write to CNTFRQ.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The CNTFRQ register is UNKNOWN at reset, and therefore the counter frequency must be set as part of the system boot process.

Software can read the CNTFRQ register, to determine the current system counter frequency, in the following states and modes:

- Hyp mode.
- Secure and Non-secure PL1 modes.
- When CNTKCTL.PL0PCTEN is set to 1, Secure and Non-secure PL0 modes.

#### Memory-mapped controls of the system counter

Some system counter controls are accessible only through the memory-mapped interface to the system counter. These controls are:

- Enabling and disabling the counter.
- Setting the counter value.
- Changing the operating mode, to change the update frequency and increment value.
- Enabling Halt-on-debug, that a debugger can then use to suspend counting.

For descriptions of these controls, see Chapter I1 *System Level Implementation of the Generic Timer*.

### G5.1.2 The physical counter

The PE includes a physical counter that contains the count value of the system counter. The CNTPCT register holds the current physical counter value.

#### Accessing the physical counter

Software with sufficient privilege can read CNTPCT using a 64-bit System register read.

CNTPCT:

- Is always accessible from Secure PL1 modes and from Non-secure Hyp mode.
- Is accessible from Non-secure PL1 modes when the value of CNTHCTL.PL1PCTEN is 1. When the value of CNTHCTL.PL1PCTEN is 0, any attempt to access CNTPCT from Non-secure PL1 modes is trapped to Hyp mode.
- Is accessible from Secure User mode when the value of CNTKCTL.PL0PCTEN is 1. When the value of CNTKCTL.PL0PCTEN is 0, any attempt to access CNTPCT generates an UNDEFINED exception.
- Is accessible from Non-secure User mode when the value of CNTHCTL.PL1PCTEN is 1 and the value of CNTKCTL.PL0PCTEN is 1. Otherwise:
  - When the value of CNTKCTL.PL0PCTEN is 0, any attempt to access CNTPCT from Non-secure User mode generates an UNDEFINED exception.
  - When the value of CNTKCTL.PL0PCTEN is 1 and the value of CNTHCTL.PL1PCTEN is 0, any attempt to access CNTPCT from Non-secure User mode is trapped to Hyp mode.

Reads of CNTPCT can occur speculatively and out of order relative to other instructions executed on the same PE.

For example, if a read from memory is used to obtain a signal from another agent that indicates that CNTPCT must be read, an ISB must be used to ensure that the read of CNTPCT occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                ; polling for some communication to indicate a requirement to read the timer
    LDR R1, [R2]
    CMP R1, #1
    BNE loop
    ISB             ; without this, the CNTPCT could be read before the memory location in [R2]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                ; has had the value 1 written to it
    MRS R1, CNTPCT
```

### G5.1.3 The virtual counter

An implementation of the Generic Timer always includes a virtual counter, that indicates virtual time.

The virtual counter contains the value of the physical counter minus a 64-bit virtual offset. When executing in a Non-secure PL1 or PL0 mode, the virtual offset value relates to the current virtual machine.

The CNTVOFF register contains the virtual offset. CNTVOFF is only accessible:

- From Hyp mode.
- From Monitor mode only when SCR.NS is set to 1.

For more information see *Status of the CNTVOFF register*.

The CNTVCT register holds the current virtual counter value.

#### Accessing the virtual counter

Software with sufficient privilege can read CNTVCT using a 64-bit System register read.

CNTVCT is always accessible from Secure PL1 modes and from Non-secure PL1 and PL2 modes.

In addition, when CNTKCTL.PL0VCTEN is set to 1, CNTVCT is accessible from PL0.

When CNTKCTL.PL0VCTEN is set to 0, any attempt to access CNTVCT from PL0 is UNDEFINED.

Reads of CNTVCT can occur speculatively and out of order relative to other instructions executed on the same PE.

For example, if a read from memory is used to obtain a signal from another agent that indicates that CNTVCT must be read, an ISB must be used to ensure that the read of CNTVCT occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                ; polling for some communication to indicate a requirement to read the timer
    LDR R1, [R2]
    CMP R1, #1
    BNE loop
    ISB             ; without this, the CNTVCT could be read before the memory location in [R2]
                    ; has had the value 1 written to it
    MRS R1, CNTVCT
```

### G5.1.4 Status of the CNTVOFF register

All implementations of the Generic Timer include the virtual counter. Therefore, conceptually, all implementations include the CNTVOFF register that defines the *virtual offset* between the physical count and the virtual count. CNTVOFF is only accessible at EL2 or above. If EL2 is not implemented, the virtual counter uses a fixed virtual offset of zero.

### G5.1.5 Event streams

Any implementation of the Generic Timer can use the system counter to generate one or more *event streams*, to generate periodic wake-up events as part of the mechanism described in *Wait for Event mechanism and Send event on page D1-1611*.

**Note**

An event stream might be used:

- To impose a time-out on a Wait For Event polling loop.
- To safeguard against any programming error that means an expected event is not generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

An event stream is configured by:

- Selecting which bit, from the bottom 16 bits of a counter, triggers the event. This determines the frequency of the events in the stream.
- Selecting whether the event is generated on each 0 to 1 transition, or each 1 to 0 transition, of the selected counter bit.

The CNTKCTL.{EVNTEN, EVNTDIR, EVNTI} fields define an event stream that is generated from the virtual counter.

In all implementations the CNTHCTL.{EVNTEN, EVNTDIR, EVNTI} fields define an event stream that is generated from the physical counter.

The operation of an event stream is as follows:

- The pseudocode variables `PreviousCNTVCT` and `PreviousCNTPCT` are initialized as:

```
// Variables used for generation of the timer event stream.
bits(64) PreviousCNTVCT = bits(64) UNKNOWN;
bits(64) PreviousCNTPCT = bits(64) UNKNOWN;
```

- The pseudocode functions `TestEventCNTV()` and `TestEventCNTP()` are called on each cycle of the PE clock.
- The `TestEventCNTx()` pseudocode template defines the functions `TestEventCNTV()` and `TestEventCNTP()`:

```
// TestEventCNTx()
// ===============

// Template for the TestEventCNTV() and TestEventCNTP() functions
// Describes operation when all Exception Levels are using AArch32:
//   CNTxCT         is  CNTVCT          or  CNTPCT          64-bit count value
//   CNTx_CTL       is  CNTV_CTL        or  CNTP_CTL        Control register
//   PreviousCNTxCT is  PreviousCNTVCT  or  PreviousCNTPCT

TestEventCNTx()
    if CNTx_CTL.EVNTEN == '1' then
        n = UInt(CNTx_CTL.EVNTI);
        SampleBit   = CNTxCT<n>;
        PreviousBit = PreviousCNTxCT<n>;

        if CNTx_CTL.EVNTDIR == '0' then
            if PreviousBit == '0' && SampleBit == '1' then EventRegisterSet();
        else
            if PreviousBit == '1' && SampleBit == '0' then EventRegisterSet();

    PreviousCNTxCT = CNTxCT;

    return;
```

### G5.1.6 Timers

In an implementation that includes EL3, in any implementation of the Generic Timer, the following timers are accessible from AArch32 state, provided the appropriate Exception level can use AArch32:

- A Non-secure EL1 physical timer. A Non-secure EL1 control determines whether this register is accessible from Non-secure EL0.
- A Secure PL1 physical timer. This timer:
  - Is accessible from Secure EL1 using AArch32 when EL3 is using AArch64.
  - Is accessible from Secure EL3 when EL3 is using AArch32.

  A Secure PL1 control determines whether this register is accessible from Secure EL0.
- A Non-secure EL2 physical timer.
- A virtual timer.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The output of each implemented timer:

- Provides an output signal to the system.
- If the PE interfaces to a *Generic Interrupt Controller* (GIC), signals a *Private Peripheral Interrupt* (PPI) to that GIC. In a multiprocessor implementation, each PE must use the same interrupt number for each timer.

Each timer is implemented as three registers:

- A 64-bit CompareValue register, that provides a 64-bit unsigned upcounter.
- A 32-bit TimerValue register, that provides a 32-bit signed downcounter.
- A 32-bit Control register.

In all implementations, the AArch32 System registers for the EL1 (or PL1) physical timer are Banked, to provide the Secure and Non-secure implementations of the timer. Table G5-1 shows the Timer registers.

**Table G5-1 Timer registers summary for the Generic Timer**

| Timer register | Secure PL1/Non-secure EL1 physical timer | PL2 physical timer | Virtual timer |
|---|---|---|---|
| CompareValue register | CNTP_CVAL[a] | CNTHP_CVAL | CNTV_CVAL |
| TimerValue register | CNTP_TVAL[a] | CNTHP_TVAL | CNTV_TVAL |
| Control register | CNTP_CTL[a] | CNTHP_CTL | CNTV_CTL |

a. In AArch32 state, these registers are Banked to provide the Non-secure EL1 physical timer and the Secure PL1 physical timer.

The following sections describe:

- *Accessing the timer registers*
- *Operation of the CompareValue views of the timers* on page G5-4208
- *Operation of the TimerValue views of the timers* on page G5-4209.

## Accessing the timer registers

For each timer, all timer registers have the same access permissions, as follows:

**Secure PL1 and Non-secure EL1 physical timer**

Accessible from PL1 modes, except that Non-secure software executing at EL2 controls access from Non-secure EL1 modes.

When access from PL1 modes is permitted, CNTKCTL.PL0PTEN determines whether the registers are accessible from EL0. If an access is not permitted because CNTKCTL.PL0PTEN is set to 0, an attempted access from EL0 is UNDEFINED.

In all implementations:

- Except for accesses from Monitor mode, accesses are to the registers for the current Security state.
- For accesses from Monitor mode, the value of SCR_EL3.NS determines whether accesses are to the Secure or the Non-secure registers.
- The Non-secure registers are accessible from Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- CNTHCTL.PL1PCEN determines whether the Non-secure registers are accessible from Non-secure EL1 modes. If this bit is set to 1, to enable access from Non-secure EL1 modes, CNTKCTL.PL0PTEN determines whether the registers are accessible from Non-secure EL0.

  If an access is not permitted because CNTHCTL.PL1PCEN is set to 0, an attempted access from a Non-secure EL1 or EL0 mode generates a Hyp Trap exception. However, if CNTKCTL.PL0PTEN is set to 0, this control takes priority, and an attempted access from EL0 is UNDEFINED.

**EL2 physical timer** Accessible from Hyp mode, and from Secure Monitor mode when SCR_EL3.NS is set to 1.

**Virtual timer** Accessible from Secure and Non-secure PL1 modes, and from Hyp mode.

CNTKCTL.PL0VTEN determines whether the registers are accessible from EL0 modes. If an access is not permitted because CNTKCTL.PL0VTEN is set to 0, an attempted access from EL0 is UNDEFINED.

### Operation of the CompareValue views of the timers

The CompareValue view of a timer operates as a 64-bit upcounter. The timer triggers when the appropriate counter reaches the value programmed into a CompareValue register. When the timer triggers, it generates an interrupt if the interrupt is enabled in the corresponding timer control register, CNTP_CTL, CNTHP_CTL, or CNTV_CTL. For CNTP_CTL, it is the interrupt asserted by the Non-secure instance of the register that is the same as the interrupt asserted by CNTP_CTL_EL0.

The operation of this view of a timer is:

```
EventTriggered = (((Counter[63:0] - Offset[63:0])[63:0] - CompareValue[63:0]) >= 0)
```

Where:

`EventTriggered` Is TRUE if the event for this timer must be triggered, and FALSE otherwise.

`Counter` The physical counter value, that can be read from the CNTPCT register.

**Note**

The virtual counter value, that can be read from the CNTVCT register, is the value:

```
(Counter - Offset)
```

`Offset` For a physical timer it is zero, and for the virtual timer it is the virtual offset, held in the CNTVOFF register.

`CompareValue` The value of the appropriate CompareValue register, CNTP_CVAL, CNTHP_CVAL, or CNTV_CVAL.

In this view of a timer, `Counter`, `Offset`, and `CompareValue` are all 64-bit unsigned values.

**Note**

This means that a timer with a `CompareValue` of, or close to, `0xFFFF_FFFF_FFFF_FFFF` might never trigger. However, there is no practical requirement to use values close to the counter wrap value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Operation of the TimerValue views of the timers

The TimerValue view of a timer operates as a signed 32-bit downcounter. A TimerValue register is programmed with a count value. This value decrements on each increment of the appropriate counter, and the timer triggers when the value reaches zero. When the timer triggers, it generates an interrupt if the interrupt is enabled in the corresponding timer control register, CNTP_CTL, CNTHP_CTL, or CNTV_CTL.

This view of a timer depends on the following behavior of accesses to TimerValue registers:

**Reads** `TimerValue = (CompareValue - (Counter - Offset))[31:0]`

**Writes** `CompareValue = ((Counter - Offset)[63:0] + SignExtend(TimerValue))[63:0]`

Where the arguments have the definitions used in *Operation of the CompareValue views of the timers* on page G5-4208, and in addition:

`TimerValue` The value of a TimerValue register, CNTP_TVAL, CNTHP_TVAL, or CNTV_TVAL.

In this view of a timer, all values are signed, in standard two's complement form.

After an event has triggered, a read of a TimerValue register indicates the time since the event triggered.

**Note**

- *Operation of the CompareValue views of the timers* on page G5-4208 gives a strict definition of EventTriggered. However, provided that the TimerValue is not expected to wrap as a 32-bit signed value when decremented from `0x80000000`, the TimerValue view can be used as giving an effect equivalent to:

  `EventTriggered = (TimerValue ≤ 0)`

- Programming TimerValue to a negative number with magnitude greater than (Counter−Offset) can lead to an arithmetic overflow that causes the CompareValue to be an extremely large positive value. This potentially delays the resultant interrupt for an extremely long period of time.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter G6
# AArch32 System Register Descriptions

This chapter describes each of the AArch32 System registers.

It contains the following sections:


- *About the AArch32 System registers* on page G6-4212.
- *General system control registers* on page G6-4213.
- *Debug registers* on page G6-4648.
- *Performance Monitors registers* on page G6-4737.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## G6.1 About the AArch32 System registers

For general information about the AArch32 System registers, see:

- *About the System registers for VMSAv8-32* on page G4-4116.
- *Organization of the CP14 registers in VMSAv8-32* on page G4-4138.
- *Organization of the CP15 registers in VMSAv8-32* on page G4-4141.
- *Functional grouping of VMSAv8-32 System registers* on page G4-4161.

The remainder of this chapter describes the AArch32 System registers, in the following sections:

- *General system control registers* on page G6-4213.
- *Debug registers* on page G6-4648.
- *Performance Monitors registers* on page G6-4737.
- *Generic Timer registers* on page G6-4780.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2 General system control registers

This section lists the System registers in AArch32 that are not part of one of the other listed groups.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.1 ACTLR, Auxiliary Control Register

The ACTLR characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

ACTLR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

ACTLR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

ACTLR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TAC==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HCR_EL2.TACR==1, Non-secure EL1 accesses to this register are trapped to EL2 using AArch64.

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register ACTLR is architecturally mapped to AArch64 System register ACTLR_EL1[31:0].

Some bits might define global configuration settings, and be common to the Secure and Non-secure instances of the register.

RW fields in this register reset to architecturally UNKNOWN values.

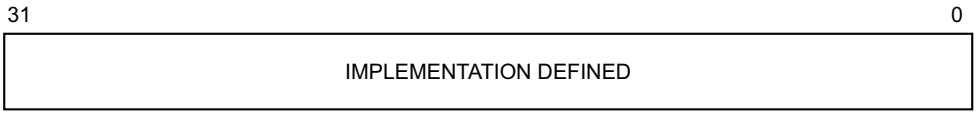
**Attributes**

ACTLR is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The ACTLR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

### Accessing the ACTLR:

To access the ACTLR:

```
MRC p15,0,<Rt>,c1,c0,1 ; Read ACTLR into Rt
MCR p15,0,<Rt>,c1,c0,1 ; Write Rt to ACTLR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0001 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.2 ACTLR2, Auxiliary Control Register 2

The ACTLR2 characteristics are:

**Purpose**

Provides additional space to the ACTLR register to hold IMPLEMENTATION DEFINED trap functionality for execution at EL1 and EL0.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

ACTLR2(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

ACTLR2(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

ACTLR2 is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TAC==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HCR_EL2.TACR==1, Non-secure EL1 accesses to this register are trapped to EL2 using AArch64.

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register ACTLR2 is architecturally mapped to AArch64 System register ACTLR_EL1[63:32].

It is IMPLEMENTATION DEFINED whether this register is implemented, or whether it causes UNDEFINED exceptions when accessed.

The implementation of this register can be detected by examining bits [7:4] of the ID_MMFR4/ID_MMFR4_EL1 register.

RW fields in this register reset to architecturally UNKNOWN values.

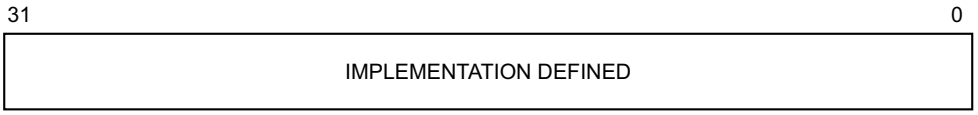

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attributes**

ACTLR2 is a 32-bit register.

## Field descriptions

The ACTLR2 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

## Accessing the ACTLR2:

To access the ACTLR2:

```
MRC p15,0,<Rt>,c1,c0,3 ; Read ACTLR2 into Rt
MCR p15,0,<Rt>,c1,c0,3 ; Write Rt to ACTLR2
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0001 | 0000 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.3 ADFSR, Auxiliary Data Fault Status Register

The ADFSR characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for Data Abort exceptions taken to EL1 modes.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

ADFSR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

ADFSR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

ADFSR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T5==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T5==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register ADFSR is architecturally mapped to AArch64 System register AFSR0_EL1.

If EL3 is implemented and is using AArch32, this register also provides fault status information for Data Abort exceptions taken to EL3 modes.

RW fields in this register reset to architecturally UNKNOWN values.

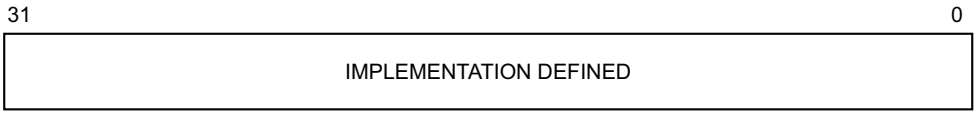

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attributes**

ADFSR is a 32-bit register.

## Field descriptions

The ADFSR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

## Accessing the ADFSR:

To access the ADFSR:

```
MRC p15,0,<Rt>,c5,c1,0 ; Read ADFSR into Rt
MCR p15,0,<Rt>,c5,c1,0 ; Write Rt to ADFSR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0101 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.4 AIDR, Auxiliary ID Register

The AIDR characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED identification information.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

The value of this register must be used in conjunction with the value of MIDR.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID1==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID1==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register AIDR is architecturally mapped to AArch64 System register AIDR_EL1.

**Attributes**

AIDR is a 32-bit register.

#### Field descriptions

The AIDR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the AIDR:

To access the AIDR:

```
MRC p15,1,<Rt>,c0,c0,7 ; Read AIDR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 001 | 0000 | 0000 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.5 AIFSR, Auxiliary Instruction Fault Status Register

The AIFSR characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for Prefetch Abort exceptions taken to EL1 modes.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

AIFSR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

AIFSR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

AIFSR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T5==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T5==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register AIFSR is architecturally mapped to AArch64 System register AFSR1_EL1.

If EL3 is implemented and is using AArch32, this register also provides fault status information for Data Abort exceptions taken to EL3 modes.

RW fields in this register reset to architecturally UNKNOWN values.

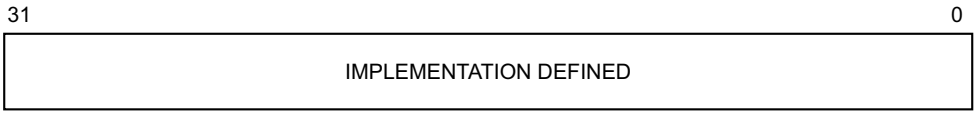

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attributes**

AIFSR is a 32-bit register.

## Field descriptions

The AIFSR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

## Accessing the AIFSR:

To access the AIFSR:

```
MRC p15,0,<Rt>,c5,c1,1 ; Read AIFSR into Rt
MCR p15,0,<Rt>,c5,c1,1 ; Write Rt to AIFSR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0101 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.6 AMAIR0, Auxiliary Memory Attribute Indirection Register 0

The AMAIR0 characteristics are:

**Purpose**

When using the Long-descriptor format translation tables for stage 1 translations, provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR0.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

AMAIR0(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

AMAIR0(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

AMAIR0 is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

This register is RES0 in the following cases:

- When an implementation does not provide any IMPLEMENTATION DEFINED memory attributes.
- When the Long-descriptor translation table format is not used.

If EL3 is implemented and is using AArch32:

- AMAIR0(S) gives the value for memory accesses from Secure state.
- AMAIR0(NS) gives the value for memory accesses from Non-secure states other than Hyp mode.

Any IMPLEMENTATION DEFINED memory attributes are additional qualifiers for the memory locations and must not change the architected behavior specified by MAIR0 and MAIR1.

In a typical implementation, AMAIR0 and AMAIR1 split into eight one-byte fields, corresponding to the MAIRn.Attr<n> fields, but the architecture does not require them to do so.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T10==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register AMAIR0 is architecturally mapped to AArch64 System register AMAIR_EL1[31:0].

When EL3 is using AArch32, write access to AMAIR0(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

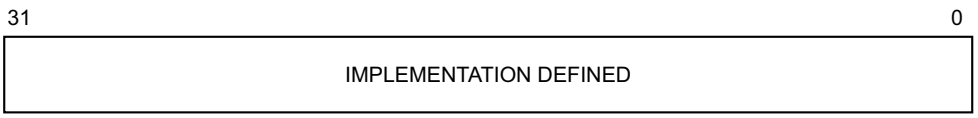
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AMAIR0 is a 32-bit register.

## Field descriptions

The AMAIR0 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

## Accessing the AMAIR0:

To access the AMAIR0:

```
MRC p15,0,<Rt>,c10,c3,0 ; Read AMAIR0 into Rt
MCR p15,0,<Rt>,c10,c3,0 ; Write Rt to AMAIR0
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1010 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.7 AMAIR1, Auxiliary Memory Attribute Indirection Register 1

The AMAIR1 characteristics are:

**Purpose**

When using the Long-descriptor format translation tables for stage 1 translations, provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR1.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

AMAIR1(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

AMAIR1(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

AMAIR1 is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

This register is RES0 in the following cases:

- When an implementation does not provide any IMPLEMENTATION DEFINED memory attributes.
- When the Long-descriptor translation table format is not used.

If EL3 is implemented and is using AArch32:

- AMAIR1(S) gives the value for memory accesses from Secure state.
- AMAIR1(NS) gives the value for memory accesses from Non-secure states other than Hyp mode.

Any IMPLEMENTATION DEFINED memory attributes are additional qualifiers for the memory locations and must not change the architected behavior specified by MAIR0 and MAIR1.

In a typical implementation, AMAIR0 and AMAIR1 split into eight one-byte fields, corresponding to the MAIRn.Attr<n> fields, but the architecture does not require them to do so.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T10==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register AMAIR1 is architecturally mapped to AArch64 System register AMAIR_EL1[63:32].

When EL3 is using AArch32, write access to AMAIR1(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

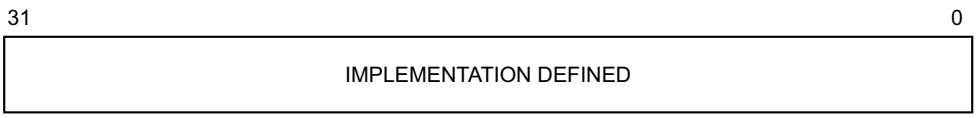
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

AMAIR1 is a 32-bit register.

## Field descriptions

The AMAIR1 bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

## Accessing the AMAIR1:

To access the AMAIR1:

```
MRC p15,0,<Rt>,c10,c3,1 ; Read AMAIR1 into Rt
MCR p15,0,<Rt>,c10,c3,1 ; Write Rt to AMAIR1
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1010 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.8 APSR, Application Program Status Register

The APSR characteristics are:

**Purpose**

Hold program status and control information.

**Usage constraints**

The APSR can be read using the MRS instruction and written using the MSR (immediate) or MSR (register) instructions. For more details on the instruction syntax, see *PSTATE and banked register access instructions* on page F1-2420.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

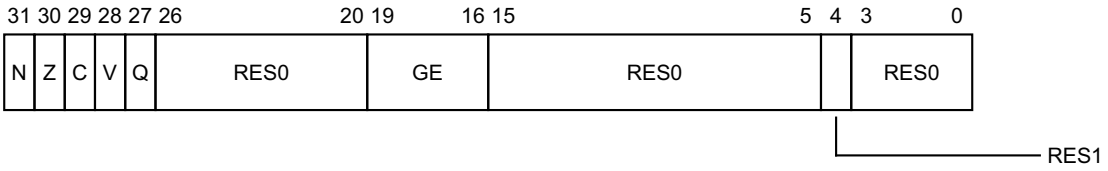
There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

APSR is a 32-bit register.

## Field descriptions

The APSR bit assignments are:




**N, bit [31]**

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

**Z, bit [30]**

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

**C, bit [29]**

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

**V, bit [28]**

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

**Bits [26:20]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**Bits [15:5]**

Reserved, RES0.

**Bit [4]**

Reserved, RES1.

**Bits [3:0]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.9 ATS12NSOPR, Address Translate Stages 1 and 2 Non-secure Only PL1 Read

The ATS12NSOPR characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for PL1 and the Non-secure state, with permissions as if reading from the given virtual address.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If EL3 is implemented and is using AArch64, execution of ATS12NSOPR in Secure EL1 using AArch32 is trapped as an exception to EL3.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

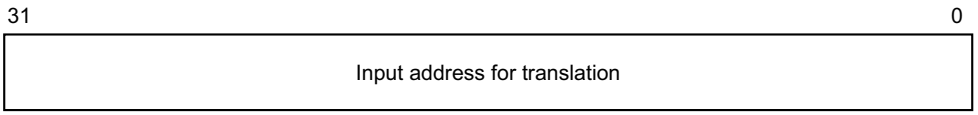
There are no configuration notes.

**Attributes**

ATS12NSOPR is a 32-bit system operation.

#### Field descriptions

The ATS12NSOPR input value bit assignments are:




**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the ATS12NSOPR operation:

To perform the ATS12NSOPR operation:

```
MCR p15,0,<Rt>,c7,c8,4 ; ATS12NSOPR operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1000 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.10 ATS12NSOPW, Address Translate Stages 1 and 2 Non-secure Only PL1 Write

The ATS12NSOPW characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for PL1 and the Non-secure state, with permissions as if writing to the given virtual address.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If EL3 is implemented and is using AArch64, execution of ATS12NSOPW in Secure EL1 using AArch32 is trapped as an exception to EL3.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

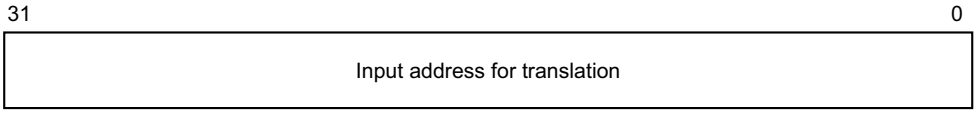
There are no configuration notes.

**Attributes**

ATS12NSOPW is a 32-bit system operation.

#### Field descriptions

The ATS12NSOPW input value bit assignments are:




**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the ATS12NSOPW operation:

To perform the ATS12NSOPW operation:

```
MCR p15,0,<Rt>,c7,c8,5 ; ATS12NSOPW operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1000 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.11 ATS12NSOUR, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read

The ATS12NSOUR characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for PL0 and the Non-secure state, with permissions as if reading from the given virtual address.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If EL3 is implemented and is using AArch64, execution of ATS12NSOUR in Secure EL1 using AArch32 is trapped as an exception to EL3.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

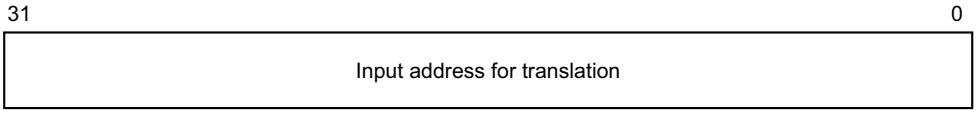
There are no configuration notes.

**Attributes**

ATS12NSOUR is a 32-bit system operation.

#### Field descriptions

The ATS12NSOUR input value bit assignments are:




**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the ATS12NSOUR operation:

To perform the ATS12NSOUR operation:

```
MCR p15,0,<Rt>,c7,c8,6 ; ATS12NSOUR operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1000 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.12 ATS12NSOUW, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write

The ATS12NSOUW characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for PL0 and the Non-secure state, with permissions as if writing to the given virtual address.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If EL3 is implemented and is using AArch64, execution of ATS12NSOUW in Secure EL1 using AArch32 is trapped as an exception to EL3.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

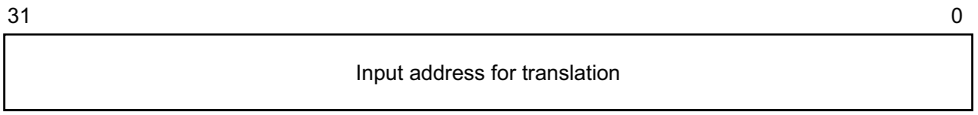
There are no configuration notes.

**Attributes**

ATS12NSOUW is a 32-bit system operation.

## Field descriptions

The ATS12NSOUW input value bit assignments are:




**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the ATS12NSOUW operation:

To perform the ATS12NSOUW operation:

```
MCR p15,0,<Rt>,c7,c8,7 ; ATS12NSOUW operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1000 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.13 ATS1CPR, Address Translate Stage 1 Current state PL1 Read

The ATS1CPR characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL1 and the current Security state, with permissions as if reading from the given virtual address.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

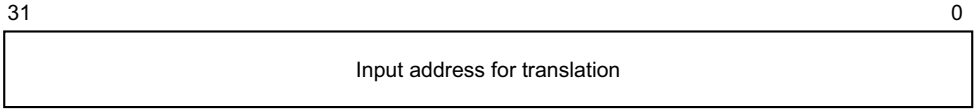
There are no configuration notes.

**Attributes**

ATS1CPR is a 32-bit system operation.

#### Field descriptions

The ATS1CPR input value bit assignments are:




**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This instruction takes a VA as input. In an implementation that includes EL2, when executed in Non-secure state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the ATS1CPR operation:

To perform the ATS1CPR operation:

```
MCR p15,0,<Rt>,c7,c8,0 ; ATS1CPR operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.14 ATS1CPW, Address Translate Stage 1 Current state PL1 Write

The ATS1CPW characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL1 and the current Security state, with permissions as if writing to the given virtual address.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

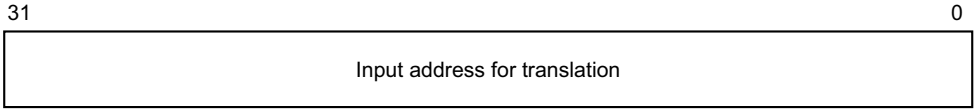
There are no configuration notes.

**Attributes**

ATS1CPW is a 32-bit system operation.

#### Field descriptions

The ATS1CPW input value bit assignments are:




**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This instruction takes a VA as input. In an implementation that includes EL2, when executed in Non-secure state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the ATS1CPW operation:

To perform the ATS1CPW operation:

```
MCR p15,0,<Rt>,c7,c8,1 ; ATS1CPW operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.15 ATS1CUR, Address Translate Stage 1 Current state Unprivileged Read

The ATS1CUR characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL0 and the current Security state, with permissions as if reading from the given virtual address.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

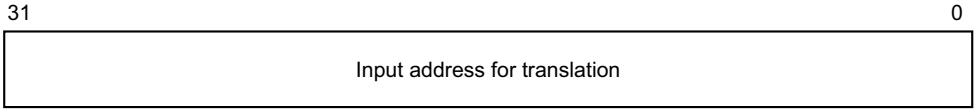
There are no configuration notes.

**Attributes**

ATS1CUR is a 32-bit system operation.

#### Field descriptions

The ATS1CUR input value bit assignments are:




**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This instruction takes a VA as input. In an implementation that includes EL2, when executed in Non-secure state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the ATS1CUR operation:

To perform the ATS1CUR operation:

```
MCR p15,0,<Rt>,c7,c8,2 ; ATS1CUR operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.16 ATS1CUW, Address Translate Stage 1 Current state Unprivileged Write

The ATS1CUW characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL0 and the current Security state, with permissions as if writing to the given virtual address.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

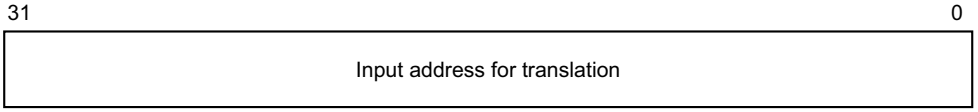
There are no configuration notes.

**Attributes**

ATS1CUW is a 32-bit system operation.

## Field descriptions

The ATS1CUW input value bit assignments are:




**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This instruction takes a VA as input. In an implementation that includes EL2, when executed in Non-secure state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the ATS1CUW operation:

To perform the ATS1CUW operation:

```
MCR p15,0,<Rt>,c7,c8,3 ; ATS1CUW operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1000 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.17 ATS1HR, Address Translate Stage 1 Hyp mode Read

The ATS1HR characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL2 and the Non-secure state, with permissions as if reading from the given virtual address.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0, Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|---|
| - | - | - | WO | WO | WO | WO-UNPREDICTABLE |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If this operation is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE. For more information see *Hyp mode VA to PA address translation instructions* on page K1-5499.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

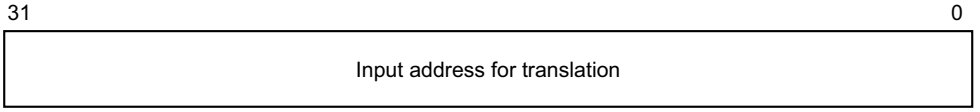
There are no configuration notes.

**Attributes**

ATS1HR is a 32-bit system operation.

#### Field descriptions

The ATS1HR input value bit assignments are:




**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This instruction takes a VA as input. The resulting address is the PA that is the output address of the translation.

### Performing the ATS1HR operation:

To perform the ATS1HR operation:

```
MCR p15,4,<Rt>,c7,c8,0 ; ATS1HR operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0111 | 1000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.18 ATS1HW, Address Translate Stage 1 Hyp mode Write

The ATS1HW characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL2 and the Non-secure state, with permissions as if writing to the given virtual address.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0, Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|---|
| - | - | - | WO | WO | WO | WO-UNPREDICTABLE |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If this operation is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE. For more information see *Hyp mode VA to PA address translation instructions* on page K1-5499.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

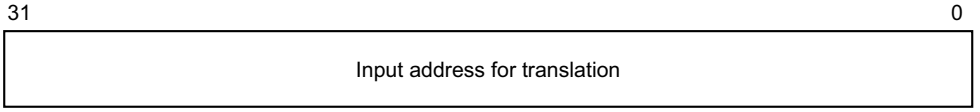
There are no configuration notes.

**Attributes**

ATS1HW is a 32-bit system operation.

#### Field descriptions

The ATS1HW input value bit assignments are:




**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This instruction takes a VA as input. The resulting address is the PA that is the output address of the translation.

### Performing the ATS1HW operation:

To perform the ATS1HW operation:

```
MCR p15,4,<Rt>,c7,c8,1 ; ATS1HW operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0111 | 1000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.19 BPIALL, Branch Predictor Invalidate All

The BPIALL characteristics are:

**Purpose**

Invalidate all entries from branch predictors.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

BPIALL is a 32-bit system operation.

#### Field descriptions

The BPIALL operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the BPIALL operation:

To perform the BPIALL operation:

```
MCR p15,0,<Rt>,c7,c5,6 ; BPIALL operation, ignoring the value in Rt
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 0101 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.20 BPIALLIS, Branch Predictor Invalidate All, Inner Shareable

The BPIALLIS characteristics are:

**Purpose**

Invalidate all entries from branch predictors Inner Shareable.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

BPIALLIS is a 32-bit system operation.

#### Field descriptions

The BPIALLIS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the BPIALLIS operation:

To perform the BPIALLIS operation:

```
MCR p15,0,<Rt>,c7,c1,6 ; BPIALLIS operation, ignoring the value in Rt
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 0001 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.21 BPIMVA, Branch Predictor Invalidate by VA

The BPIMVA characteristics are:

**Purpose**

Invalidate virtual address from branch predictors.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

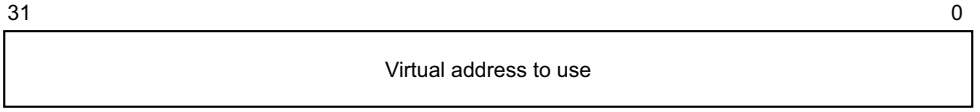
There are no configuration notes.

**Attributes**

BPIMVA is a 32-bit system operation.

### Field descriptions

The BPIMVA input value bit assignments are:




**Bits [31:0]**

Virtual address to use.

### Performing the BPIMVA operation:

To perform the BPIMVA operation:

```
MCR p15,0,<Rt>,c7,c5,7 ; BPIMVA operation
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 0101 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.22 CCSIDR, Current Cache Size ID Register

The CCSIDR characteristics are:

**Purpose**

Provides information about the architecture of the currently selected cache.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

If CSSELR.Level indicates a cache that is not implemented, then on a read of the CCSIDR the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR read is treated as NOP.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID2==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID2==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register CCSIDR is architecturally mapped to AArch64 System register CCSIDR_EL1.

The implementation includes one CCSIDR for each cache that it can access. CSSELR and the Security state select which Cache Size ID Register is accessible.

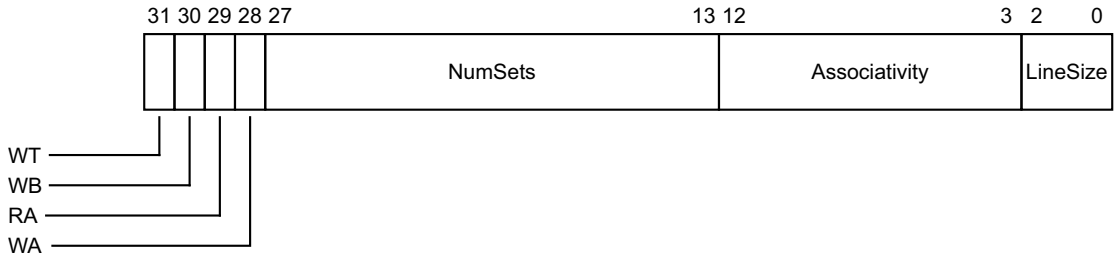
**Attributes**

CCSIDR is a 32-bit register.

#### Field descriptions

The CCSIDR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**WT, bit [31]**

Indicates whether the selected cache level supports write-through. Permitted values are:

0 Write-through not supported.

1 Write-through supported.

**WB, bit [30]**

Indicates whether the selected cache level supports write-back. Permitted values are:

0 Write-back not supported.

1 Write-back supported.

**RA, bit [29]**

Indicates whether the selected cache level supports read-allocation. Permitted values are:

0 Read-allocation not supported.

1 Read-allocation supported.

**WA, bit [28]**

Indicates whether the selected cache level supports write-allocation. Permitted values are:

0 Write-allocation not supported.

1 Write-allocation supported.

**NumSets, bits [27:13]**

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

**Associativity, bits [12:3]**

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

**LineSize, bits [2:0]**

($Log_2$(Number of bytes in cache line)) - 4. For example:

For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.

For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

**Note**

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

## Accessing the CCSIDR:

To access the CCSIDR:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRC p15,1,<Rt>,c0,c0,0 ; Read CCSIDR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 001 | 0000 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.23 CLIDR, Cache Level ID Register

The CLIDR characteristics are:

**Purpose**

Identifies the type of cache, or caches, implemented at each level, up to a maximum of seven levels. Also identifies the Level of Coherence (LoC) and Level of Unification (LoU) for the cache hierarchy.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID2==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID2==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

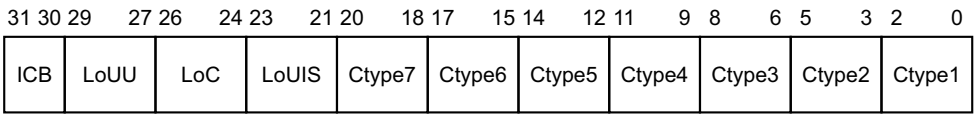
AArch32 System register CLIDR is architecturally mapped to AArch64 System register CLIDR_EL1.

**Attributes**

CLIDR is a 32-bit register.

### Field descriptions

The CLIDR bit assignments are:




**ICB, bits [31:30]**

Inner cache boundary. This field indicates the boundary between the inner and the outer domain.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The possible values are:

`00` Not disclosed in this mechanism.

`01` L1 cache is the highest inner level.

`10` L2 cache is the highest inner level.

`11` L3 cache is the highest inner level.

**LoUU, bits [29:27]**

Level of Unification Uniprocessor for the cache hierarchy.

**LoC, bits [26:24]**

Level of Coherence for the cache hierarchy.

**LoUIS, bits [23:21]**

Level of Unification Inner Shareable for the cache hierarchy.

**Ctype<n>, bits [3(n-1)+2:3(n-1)], for n = 1 to 7**

Cache Type fields. Indicate the type of cache implemented at each level, from Level 1 up to a maximum of seven levels of cache hierarchy. Possible values of each field are:

`000` No cache.

`001` Instruction cache only.

`010` Data cache only.

`011` Separate instruction and data caches.

`100` Unified cache.

All other values are reserved.

If software reads the Cache Type fields from Ctype1 upwards, once it has seen a value of `000`, no caches exist at further-out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of `000`, the values of Ctype4 to Ctype7 must be ignored.

### Accessing the CLIDR:

To access the CLIDR:

```
MRC p15,1,<Rt>,c0,c0,1 ; Read CLIDR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 001 | 0000 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.24 CONTEXTIDR, Context ID Register

The CONTEXTIDR characteristics are:

**Purpose**

Identifies the current Process Identifier and, when using the Short-descriptor translation table format, the Address Space Identifier.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

CONTEXTIDR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

CONTEXTIDR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

CONTEXTIDR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T13==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T13==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register CONTEXTIDR is architecturally mapped to AArch64 System register CONTEXTIDR_EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The register format depends on whether address translation is using the Long-descriptor or the Short-descriptor translation table format.

RW fields in this register reset to architecturally UNKNOWN values.

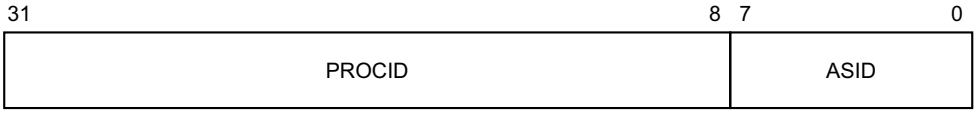
**Attributes**

CONTEXTIDR is a 32-bit register.

## Field descriptions

The CONTEXTIDR bit assignments are:

***When TTBCR.EAE==0:***



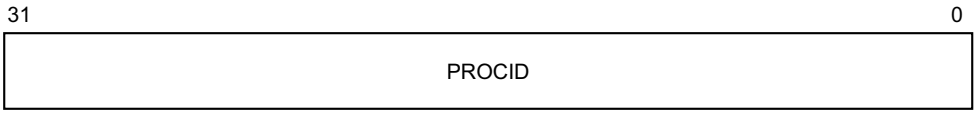

**PROCID, bits [31:8]**

Process Identifier. This field must be programmed with a unique value that identifies the current process.

**ASID, bits [7:0]**

Address Space Identifier. This field is programmed with the value of the current ASID.

***When TTBCR.EAE==1:***




**PROCID, bits [31:0]**

Process Identifier. This field must be programmed with a unique value that identifies the current process.

## Accessing the CONTEXTIDR:

To access the CONTEXTIDR:

```
MRC p15,0,<Rt>,c13,c0,1 ; Read CONTEXTIDR into Rt
MCR p15,0,<Rt>,c13,c0,1 ; Write Rt to CONTEXTIDR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1101 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.25 CP15DMB, CP15 Data Memory Barrier operation

The CP15DMB characteristics are:

**Purpose**

Performs a Data Memory Barrier.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| WO | WO | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| WO | WO | WO |

ARM deprecates any use of this operation, and strongly recommends that software use the DMB instruction instead.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL0 and EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL0 and EL1 execution of this instruction is trapped to EL2.

If HSCTLR.CP15BEN==0, PL2 execution of this instruction is UNDEFINED.

If SCTLR.CP15BEN==0, PL0 and PL1 execution of this instruction is UNDEFINED.

If SCTLR_EL1.CP15BEN==0, PL0 execution of this instruction is UNDEFINED.

**Configurations**

There are no configuration notes.

**Attributes**

CP15DMB is a 32-bit system operation.

#### Field descriptions

The CP15DMB operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the CP15DMB operation:

To perform the CP15DMB operation:

```
MCR p15,0,<Rt>,c7,c10,5 ; CP15DMB operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1010 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.26 CP15DSB, CP15 Data Synchronization Barrier operation

The CP15DSB characteristics are:

**Purpose**

Performs a Data Synchronization Barrier.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| WO | WO | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| WO | WO | WO |

ARM deprecates any use of this operation, and strongly recommends that software use the DSB instruction instead.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL0 and EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL0 and EL1 execution of this instruction is trapped to EL2.

If HSCTLR.CP15BEN==0, PL2 execution of this instruction is UNDEFINED.

If SCTLR.CP15BEN==0, PL0 and PL1 execution of this instruction is UNDEFINED.

If SCTLR_EL1.CP15BEN==0, PL0 execution of this instruction is UNDEFINED.

**Configurations**

There are no configuration notes.

**Attributes**

CP15DSB is a 32-bit system operation.

#### Field descriptions

The CP15DSB operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the CP15DSB operation:

To perform the CP15DSB operation:

```
MCR p15,0,<Rt>,c7,c10,4 ; CP15DSB operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1010 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.27 CP15ISB, CP15 Instruction Synchronization Barrier operation

The CP15ISB characteristics are:

**Purpose**

Performs an Instruction Synchronization Barrier.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| WO | WO | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| WO | WO | WO |

ARM deprecates any use of this operation, and strongly recommends that software use the ISB instruction instead.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL0 and EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL0 and EL1 execution of this instruction is trapped to EL2.

If HSCTLR.CP15BEN==0, PL2 execution of this instruction is UNDEFINED.

If SCTLR.CP15BEN==0, PL0 and PL1 execution of this instruction is UNDEFINED.

If SCTLR_EL1.CP15BEN==0, PL0 execution of this instruction is UNDEFINED.

**Configurations**

There are no configuration notes.

**Attributes**

CP15ISB is a 32-bit system operation.

#### Field descriptions

The CP15ISB operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the CP15ISB operation:

To perform the CP15ISB operation:

```
MCR p15,0,<Rt>,c7,c5,4 ; CP15ISB operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 0101 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.28 CPACR, Architectural Feature Access Control Register

The CPACR characteristics are:

**Purpose**

Controls access to Trace, floating-point, and Advanced SIMD functionality from EL0, EL1, and EL3.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

In an implementation that includes EL2, the CPACR has no effect on instructions executed at EL2.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CPTR_EL2.TCPAC==1, Non-secure EL1 accesses to this register are trapped to EL2.

If CPTR_EL3.TCPAC==1, EL1 and EL2 accesses to this register are trapped to EL3.

If HCPTR.TCPAC==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register CPACR is architecturally mapped to AArch64 System register CPACR_EL1.

Bits in the NSACR control Non-secure access to the CPACR fields. See the field descriptions for more information.

**Note**

In the register field descriptions, controls are described as applying at specified Privilege levels. This is because, in Secure state, a PL1 control:

- Applies to execution in a Secure EL3 mode when EL3 is using AArch32.
- Applies to execution in a Secure EL1 mode when EL3 is using AArch64.

See *Execution privilege, Exception levels, and AArch32 Privilege levels* on page G4-3990.

Some or all RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

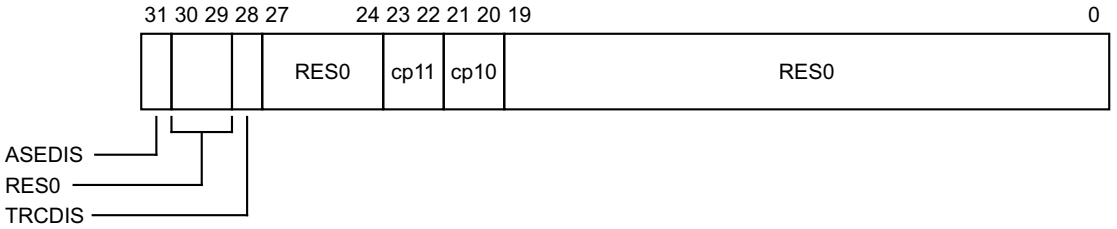

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attributes**

CPACR is a 32-bit register.

## Field descriptions

The CPACR bit assignments are:




**ASEDIS, bit [31]**

Disables PL0 and PL1 execution of Advanced SIMD instructions.

`0` This control permits execution of Advanced SIMD instructions at PL0 and PL1.

`1` All instruction encodings that are Advanced SIMD instruction encodings, but are not also floating-point instruction encodings, are UNDEFINED at PL0 and PL1.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSASEDIS is 1, this field behaves as RAO/WI in Non-secure state, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.

For the list of instructions affected by this field, see *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-2332.

See the description of CPACR.cp10 for a list of other controls that can disable or trap execution of Advanced SIMD instructions in AArch32 state.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**Bits [30:29]**

Reserved, RES0.

**TRCDIS, bit [28]**

Traps PL0 and PL1 System register accesses to all implemented trace registers to Undefined mode.

`0` This control has no effect on PL0 and PL1 System register accesses to trace registers.

`1` PL0 and PL1 System register accesses to all implemented trace registers are trapped to Undefined mode.

If the implementation does not include a trace macrocell, or does not include a System register interface to the trace macrocell registers, this field is RES0. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSTRCDIS is 1, this field behaves as RAO/WI in Non-secure state, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the implementation includes an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED.
- The architecture does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**Bits [27:24]**

Reserved, RES0.

**cp11, bits [23:22]**

The value of this field is ignored. If this field is programmed with a different value to the cp10 field then this field is UNKNOWN on a direct read of the CPACR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

In Non-secure state, if EL3 is implemented and is using AArch32, when the value of NSACR.cp10 is 0, this field behaves as RAZ/WI, regardless of its actual value.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**cp10, bits [21:20]**

Defines the access rights for the floating-point and Advanced SIMD functionality. Possible values of the field are:

`00` PL0 and PL1 accesses to floating-point and Advanced SIMD registers or instructions are UNDEFINED.

`01` PL0 accesses to floating-point and Advanced SIMD registers or instructions are UNDEFINED.

`10` Reserved. The effect of programming this field to this value is CONSTRAINED UNPREDICTABLE. See *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

`11` This control permits full access to the floating-point and Advanced SIMD functionality from PL0 and PL1.

The floating-point and Advanced SIMD features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

**Note**

The CPACR has no effect on floating-point and Advanced SIMD accesses from PL2. These can be disabled by the HCPTR.TCP10 field.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

In Non-secure state, if EL3 is implemented and is using AArch32, when the value of NSACR.cp10 is 0, this field behaves as RAZ/WI, regardless of its actual value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Execution of floating-point and Advanced SIMD instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.
- If executing in Non-secure state:
  - HCPTR.TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
  - CPACR.ASEDIS.
  - If executing in Non-secure state, HCPTR.TASE and NSACR.NSTRCDIS.

See the descriptions of the controls for more information.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**Bits [19:0]**

Reserved, RES0.

### Accessing the CPACR:

To access the CPACR:

```
MRC p15,0,<Rt>,c1,c0,2 ; Read CPACR into Rt
MCR p15,0,<Rt>,c1,c0,2 ; Write Rt to CPACR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0001 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.29 CPSR, Current Program Status Register

The CPSR characteristics are:

**Purpose**

Holds PE status and control information.

**Usage constraints**

The CPSR can be read using the MRS instruction and written using the MSR (immediate) or MSR (register) instructions. For more details on the instruction syntax, see *PSTATE and banked register access instructions* on page F1-2420.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

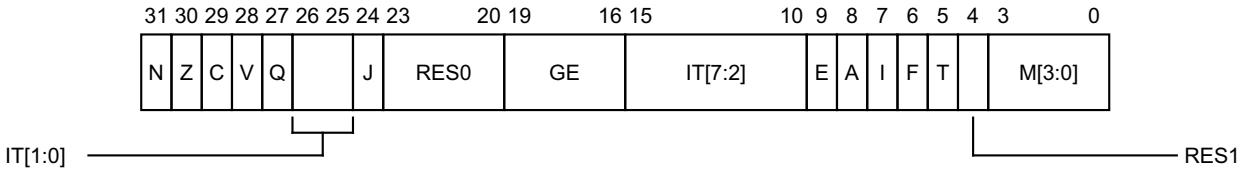
There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

CPSR is a 32-bit register.

### Field descriptions

The CPSR bit assignments are:




**N, bit [31]**

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

**Z, bit [30]**

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

**C, bit [29]**

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

**V, bit [28]**

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:20]**

Reserved, RES0.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**T, bit [5]**

T32 Instruction set state bit. Indicates the AArch32 instruction set state. Possible values of this bit are:

`0` A32 state.

`1` T32 state.

**Bit [4]**

Reserved, RES1.

**M[3:0], bits [3:0]**

Current PE mode. Possible values are:

| M[3:0] | Mode |
|---|---|
| `0b0000` | User |
| `0b0001` | FIQ |
| `0b0010` | IRQ |
| `0b0011` | Supervisor |
| `0b0110` | Monitor |
| `0b0111` | Abort |
| `0b1010` | Hyp |
| `0b1011` | Undefined |
| `0b1111` | System |

Other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.30 CSSELR, Cache Size Selection Register

The CSSELR characteristics are:

**Purpose**

Selects the current Cache Size ID Register, CCSIDR, by specifying the required cache level and the cache type (either instruction or data cache).

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

CSSELR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

CSSELR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

CSSELR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

If CSSELR.Level is programmed to a cache level that is not implemented, then a read of CSSELR is CONSTRAINED UNPREDICTABLE, and returns UNKNOWN values for CSSELR.{Level, InD}.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID2==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID2==1, Non-secure EL1 accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register CSSELR is architecturally mapped to AArch64 System register CSSELR_EL1.

RW fields in this register reset to architecturally UNKNOWN values.

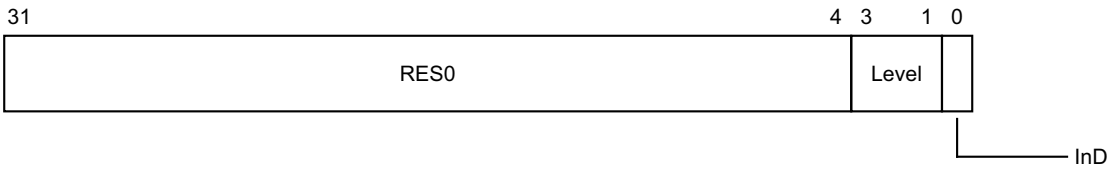
**Attributes**

CSSELR is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The CSSELR bit assignments are:




**Bits [31:4]**

Reserved, RES0.

**Level, bits [3:1]**

Cache level of required cache. Permitted values are:

`000` Level 1 cache

`001` Level 2 cache

`010` Level 3 cache

`011` Level 4 cache

`100` Level 5 cache

`101` Level 6 cache

`110` Level 7 cache

All other values are reserved. See the Usage constraints for the behavior when the Level is programmed to a reserved value, or a cache level that is not implemented.

**InD, bit [0]**

Instruction not Data bit. Permitted values are:

`0` Data or unified cache.

`1` Instruction cache.

### Accessing the CSSELR:

To access the CSSELR:

```
MRC p15,2,<Rt>,c0,c0,0 ; Read CSSELR into Rt
MCR p15,2,<Rt>,c0,c0,0 ; Write Rt to CSSELR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 010 | 0000 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.31 CTR, Cache Type Register

The CTR characteristics are:

**Purpose**

Provides information about the architecture of the caches.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID2==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID2==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

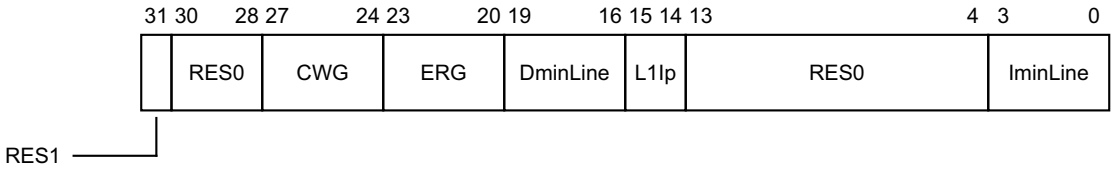
AArch32 System register CTR is architecturally mapped to AArch64 System register CTR_EL0.

**Attributes**

CTR is a 32-bit register.

## Field descriptions

The CTR bit assignments are:




**Bit [31]**

Reserved, RES1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [30:28]**

Reserved, RES0.

**CWG, bits [27:24]**

Cache Writeback Granule. $Log_2$ of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

A value of `0b0000` indicates that this register does not provide Cache Writeback Granule information and either:

- The architectural maximum of 512 words (2KB) must be assumed.
- The Cache Writeback Granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than `0b1001` are reserved.

**ERG, bits [23:20]**

Exclusives Reservation Granule. $Log_2$ of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

A value of `0b0000` indicates that this register does not provide Exclusives Reservation Granule information and the architectural maximum of 512 words (2KB) must be assumed.

Values greater than `0b1001` are reserved.

**DminLine, bits [19:16]**

$Log_2$ of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

**L1Ip, bits [15:14]**

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache. Possible values of this field are:

`01` ASID-tagged Virtual Index, Virtual Tag (AIVIVT)

`10` Virtual Index, Physical Tag (VIPT)

`11` Physical Index, Physical Tag (PIPT)

Other values are reserved.

**Bits [13:4]**

Reserved, RES0.

**IminLine, bits [3:0]**

$Log_2$ of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

## Accessing the CTR:

To access the CTR:

```
MRC p15,0,<Rt>,c0,c0,1 ; Read CTR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.32 DACR, Domain Access Control Register

The DACR characteristics are:

**Purpose**

Defines the access permission for each of the sixteen memory domains.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

DACR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

DACR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

DACR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T3==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T3==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register DACR is architecturally mapped to AArch64 System register DACR32_EL2.

When EL3 is using AArch32, write access to DACR(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

This register has no function when TTBCR.EAE is set to 1, to select the Long-descriptor translation table format.

RW fields in this register reset to architecturally UNKNOWN values.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attributes**

DACR is a 32-bit register.

## Field descriptions

The DACR bit assignments are:

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D15 | | | | D14 | | | | D13 | | | | D12 | | | | D11 | | | | D10 | | | | D9 | | | | D8 | | | |
| D7 | | | | D6 | | | | D5 | | | | D4 | | | | D3 | | | | D2 | | | | D1 | | | | D0 | | | |

**D<n>, bits [2n+1:2n], for n = 0 to 15**

Domain n access permission, where n = 0 to 15. Permitted values are:

00 No access. Any access to the domain generates a Domain fault.

01 Client. Accesses are checked against the permission bits in the translation tables.

11 Manager. Accesses are not checked against the permission bits in the translation tables.

The value 10 is reserved.

## Accessing the DACR:

To access the DACR:

```
MRC p15,0,<Rt>,c3,c0,0 ; Read DACR into Rt
MCR p15,0,<Rt>,c3,c0,0 ; Write Rt to DACR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0011 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.33 DCCIMVAC, Data Cache line Clean and Invalidate by VA to PoC

The DCCIMVAC characteristics are:

**Purpose**

Clean and Invalidate data or unified cache line by virtual address to PoC.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TPC==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TPC==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

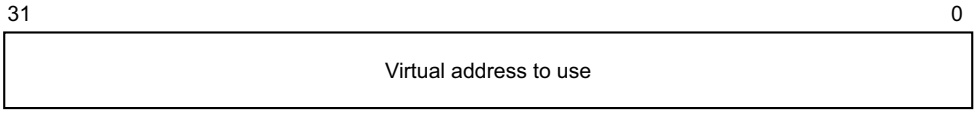
AArch32 System operation DCCIMVAC performs the same function as AArch64 System operation DC CIVAC.

**Attributes**

DCCIMVAC is a 32-bit system operation.

#### Field descriptions

The DCCIMVAC input value bit assignments are:




**Bits [31:0]**

Virtual address to use.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the DCCIMVAC operation:

To perform the DCCIMVAC operation:

```
MCR p15,0,<Rt>,c7,c14,1 ; DCCIMVAC operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1110 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.34 DCCISW, Data Cache line Clean and Invalidate by Set/Way

The DCCISW characteristics are:

**Purpose**

Clean and Invalidate data or unified cache line by set/way.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TSW==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TSW==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

AArch32 System operation DCCISW performs the same function as AArch64 System operation DC CISW.

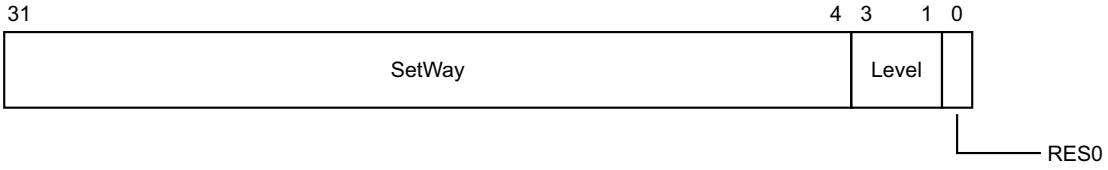
**Attributes**

DCCISW is a 32-bit system operation.

#### Field descriptions

The DCCISW input value bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**SetWay, bits [31:4]**

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = $Log_2$(ASSOCIATIVITY), L = $Log_2$(LINELEN), B = (L + S), S = $Log_2$(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**

Reserved, RES0.

### Performing the DCCISW operation:

To perform the DCCISW operation:

```
MCR p15,0,<Rt>,c7,c14,2 ; DCCISW operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1110 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.35 DCCMVAC, Data Cache line Clean by VA to PoC

The DCCMVAC characteristics are:

**Purpose**

Clean data or unified cache line by virtual address to PoC.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TPC==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TPC==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

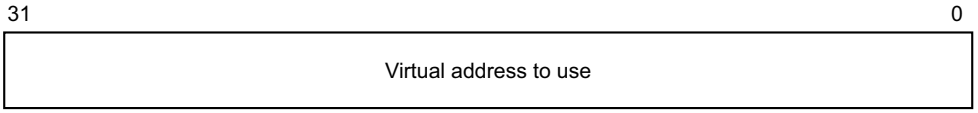
AArch32 System operation DCCMVAC performs the same function as AArch64 System operation DC CVAC.

**Attributes**

DCCMVAC is a 32-bit system operation.

#### Field descriptions

The DCCMVAC input value bit assignments are:




**Bits [31:0]**

Virtual address to use.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the DCCMVAC operation:

To perform the DCCMVAC operation:

```
MCR p15,0,<Rt>,c7,c10,1 ; DCCMVAC operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.36 DCCMVAU, Data Cache line Clean by VA to PoU

The DCCMVAU characteristics are:

**Purpose**

Clean data or unified cache line by virtual address to PoU.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TPU==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TPU==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

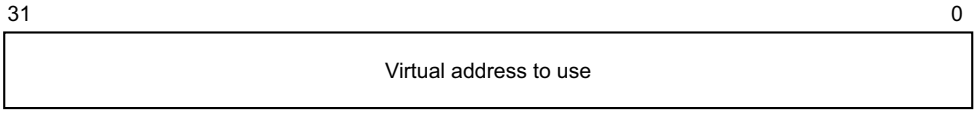
AArch32 System operation DCCMVAU performs the same function as AArch64 System operation DC CVAU.

**Attributes**

DCCMVAU is a 32-bit system operation.

#### Field descriptions

The DCCMVAU input value bit assignments are:




**Bits [31:0]**

Virtual address to use.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the DCCMVAU operation:

To perform the DCCMVAU operation:

```
MCR p15,0,<Rt>,c7,c11,1 ; DCCMVAU operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.37 DCCSW, Data Cache line Clean by Set/Way

The DCCSW characteristics are:

**Purpose**

Clean data or unified cache line by set/way.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TSW==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TSW==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

AArch32 System operation DCCSW performs the same function as AArch64 System operation DC CSW.

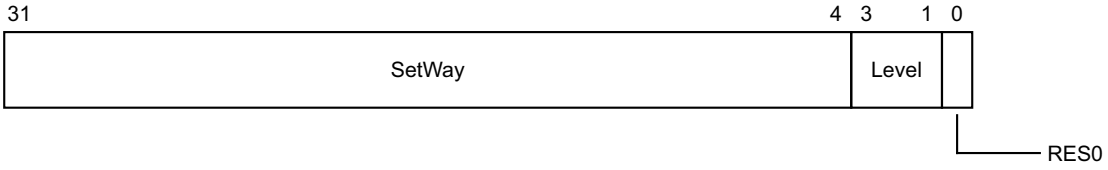
**Attributes**

DCCSW is a 32-bit system operation.

#### Field descriptions

The DCCSW input value bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**SetWay, bits [31:4]**

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = $Log_2$(ASSOCIATIVITY), L = $Log_2$(LINELEN), B = (L + S), S = $Log_2$(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**

Reserved, RES0.

### Performing the DCCSW operation:

To perform the DCCSW operation:

```
MCR p15,0,<Rt>,c7,c10,2 ; DCCSW operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 1010 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.38 DCIMVAC, Data Cache line Invalidate by VA to PoC

The DCIMVAC characteristics are:

**Purpose**

Invalidate data or unified cache line by virtual address to PoC.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

It is IMPLEMENTATION DEFINED whether, when this instruction is executed, it can generate a watchpoint. If this instruction can generate a watchpoint this is prioritized in the same way as other watchpoints.

At EL1, this instruction must be performed as DCCIMVAC if all of the following apply:

- EL2 is implemented and either:
  - EL2 is using AArch64 and the value of HCR_EL2.VM is 1.
  - EL2 is using AArch32 and the value of HCR.VM is 1.
- Execution is in Non-secure state, or EL3 is not implemented.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TPC==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TPC==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

AArch32 System operation DCIMVAC performs the same function as AArch64 System operation DC IVAC.

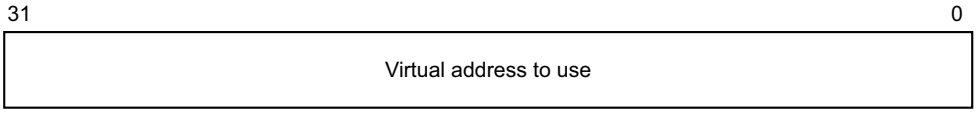
**Attributes**

DCIMVAC is a 32-bit system operation.

#### Field descriptions

The DCIMVAC input value bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:0]**

Virtual address to use.

### Performing the DCIMVAC operation:

To perform the DCIMVAC operation:

```
MCR p15,0,<Rt>,c7,c6,1 ; DCIMVAC operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 0110 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.39 DCISW, Data Cache line Invalidate by Set/Way

The DCISW characteristics are:

**Purpose**

Invalidate data or unified cache line by set/way.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

At EL1, this operation must be performed as DCCISW if all of the following apply:

- EL2 is implemented and either:
  - EL2 is using AArch64 and the value of HCR_EL2.{SWIO, VM} is not {0, 0}.
  - EL2 is using AArch32 and the value of HCR.{SWIO, VM} is not {0, 0}.
- Execution is in Non-secure state, or EL3 is not implemented.

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TSW==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TSW==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

AArch32 System operation DCISW performs the same function as AArch64 System operation DC ISW.

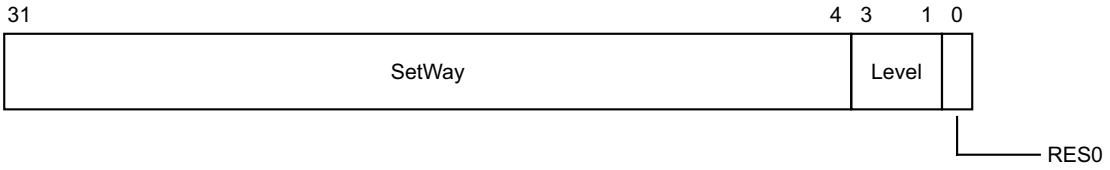

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attributes**

DCISW is a 32-bit system operation.

## Field descriptions

The DCISW input value bit assignments are:




**SetWay, bits [31:4]**

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = $Log_2$(ASSOCIATIVITY), L = $Log_2$(LINELEN), B = (L + S), S = $Log_2$(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**

Reserved, RES0.

## Performing the DCISW operation:

To perform the DCISW operation:

```
MCR p15,0,<Rt>,c7,c6,2 ; DCISW operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 0110 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.40 DFAR, Data Fault Address Register

The DFAR characteristics are:

**Purpose**

Holds the virtual address of the faulting address that caused a synchronous Data Abort exception.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

DFAR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

DFAR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

DFAR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T6==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T6==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register DFAR(NS) is architecturally mapped to AArch64 System register FAR_EL1[31:0].

AArch32 System register DFAR(S) is architecturally mapped to AArch32 System register HDFAR when EL2 is implemented.

AArch32 System register DFAR(S) is architecturally mapped to AArch64 System register FAR_EL2[31:0] when EL2 is implemented.

RW fields in this register reset to architecturally UNKNOWN values.

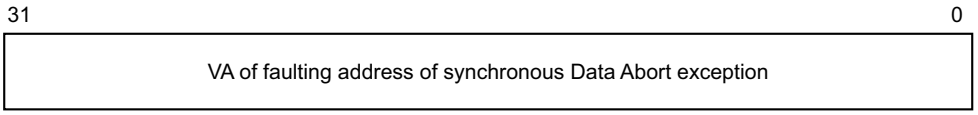

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attributes**

DFAR is a 32-bit register.

## Field descriptions

The DFAR bit assignments are:




**Bits [31:0]**

VA of faulting address of synchronous Data Abort exception.

## Accessing the DFAR:

To access the DFAR:

```
MRC p15,0,<Rt>,c6,c0,0 ; Read DFAR into Rt
MCR p15,0,<Rt>,c6,c0,0 ; Write Rt to DFAR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0110 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.41 DFSR, Data Fault Status Register

The DFSR characteristics are:

**Purpose**

Holds status information about the last data fault.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

DFSR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

DFSR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

DFSR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T5==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T5==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register DFSR is architecturally mapped to AArch64 System register ESR_EL1.

The current translation table format determines which format of the register is used.

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

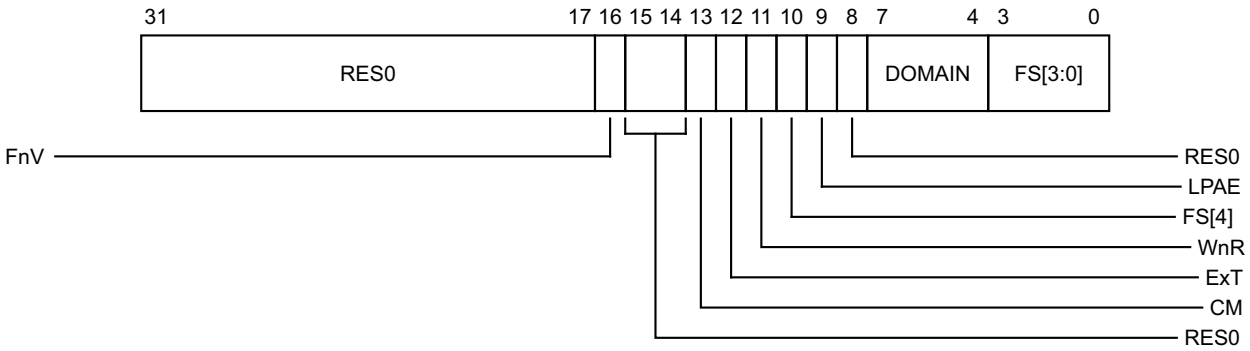
DFSR is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Field descriptions

The DFSR bit assignments are:

***When TTBCR.EAE==0:***




**Bits [31:17]**

Reserved, RES0.

**FnV, bit [16]**

FAR not Valid, for a Synchronous external abort other than a Synchronous external abort on a translation table walk.

0 DFAR is valid.

1 DFAR is not valid, and holds an UNKNOWN value.

This field is only valid for a Synchronous external abort other than a Synchronous external abort on a translation table walk. It is RES0 for all other Data Abort exceptions.

**Bits [15:14]**

Reserved, RES0.

**CM, bit [13]**

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance instruction generated the fault. The possible values of this bit are:

0 Abort not caused by execution of a cache maintenance instruction.

1 Abort caused by execution of a cache maintenance instruction.

On a synchronous Data Abort on a translation table walk, this bit is UNKNOWN.

On an asynchronous fault, this bit is UNKNOWN.

**ExT, bit [12]**

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

**WnR, bit [11]**

Write not Read bit. Indicates whether the abort was caused by a write or a read instruction. The possible values of this bit are:

0 Abort caused by a read instruction.

1 Abort caused by a write instruction.

For faults on CP15 cache maintenance instructions, including the VA to PA translation instructions, this bit always returns a value of 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**FS[4], bit [10]**

See FS[3:0], bits [3:0] for description of the FS field.

**LPAE, bit [9]**

On taking a Data Abort exception, this bit is set as follows:

0 Using the Short-descriptor translation table formats.

1 Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

**Bit [8]**

Reserved, RES0.

**DOMAIN, bits [7:4]**

Domain of the fault address. Use of this field is deprecated.

**FS[3:0], bits [3:0]**

Fault status bits. Interpreted with bit [10]. Possible values of FS[4:0] are:

00001 Alignment fault

00010 Debug event

00011 Access flag fault, level 1

00100 Fault on instruction cache maintenance

00101 Translation fault, level 1

00110 Access flag fault, level 2

00111 Translation fault, level 2

01000 Synchronous external abort, other than synchronous parity or ECC error, not on translation table walk

01001 Domain fault, level 1

01011 Domain fault, level 2

01100 Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 1

01101 Permission fault, level 1

01110 Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 2

01111 Permission fault, level 2

10000 TLB conflict abort

10100 IMPLEMENTATION DEFINED fault (Lockdown fault)

10101 IMPLEMENTATION DEFINED fault (Unsupported Exclusive access fault)

10110 Asynchronous external abort, other than synchronous parity or ECC error

11000 Asynchronous parity or ECC error on memory access

11001 Synchronous parity or ECC error on memory access, not on translation table walk

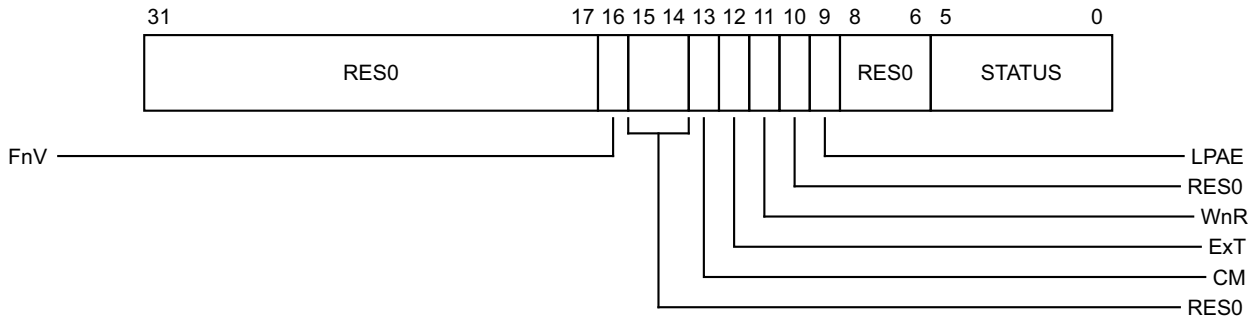
11100 Synchronous parity or ECC error on translation table walk, level 1

11110 Synchronous parity or ECC error on translation table walk, level 2

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

***When TTBCR.EAE==1:***




**Bits [31:17]**

Reserved, RES0.

**FnV, bit [16]**

FAR not Valid, for a Synchronous external abort other than a Synchronous external abort on a translation table walk.

0 DFAR is valid.

1 DFAR is not valid, and holds an UNKNOWN value.

This field is only valid for a Synchronous external abort other than a Synchronous external abort on a translation table walk. It is RES0 for all other Data Abort exceptions.

**Bits [15:14]**

Reserved, RES0.

**CM, bit [13]**

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance instruction generated the fault. The possible values of this bit are:

0 Abort not caused by execution of a cache maintenance instruction.

1 Abort caused by execution of a cache maintenance instruction.

On a synchronous Data Abort on a translation table walk, this bit is UNKNOWN.

On an asynchronous fault, this bit is UNKNOWN.

**ExT, bit [12]**

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

**WnR, bit [11]**

Write not Read bit. Indicates whether the abort was caused by a write or a read instruction. The possible values of this bit are:

0 Abort caused by a read instruction.

1 Abort caused by a write instruction.

For faults on CP15 cache maintenance instructions, including the VA to PA translation instructions, this bit always returns a value of 1.

**Bit [10]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**LPAE, bit [9]**

On taking a Data Abort exception, this bit is set as follows:

0 Using the Short-descriptor translation table formats.

1 Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

**Bits [8:6]**

Reserved, RES0.

**STATUS, bits [5:0]**

Fault status bits. Possible values of this field are:

000000 Address size fault in TTBR0 or TTBR1

000001 Address size fault, level 1

000010 Address size fault, level 2

000011 Address size fault, level 3

000101 Translation fault, level 1

000110 Translation fault, level 2

000111 Translation fault, level 3

001001 Access flag fault, level 1

001010 Access flag fault, level 2

001011 Access flag fault, level 3

001101 Permission fault, level 1

001110 Permission fault, level 2

001111 Permission fault, level 3

010000 Synchronous external abort, other than synchronous parity or ECC error, not on translation table walk

010001 Asynchronous external abort, other than synchronous parity or ECC error

010101 Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 1

010110 Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 2

010111 Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 3

011000 Synchronous parity or ECC error on memory access, not on translation table walk

011001 Asynchronous parity or ECC error on memory access

011101 Synchronous parity or ECC error on memory access on translation table walk, level 1

011110 Synchronous parity or ECC error on memory access on translation table walk, level 2

011111 Synchronous parity or ECC error on memory access on translation table walk, level 3

100001 Alignment fault

100010 Debug event

110000 TLB conflict abort

110100 IMPLEMENTATION DEFINED fault (Lockdown fault)

110101 IMPLEMENTATION DEFINED fault (Unsupported Exclusive access fault)

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a fault at level 1.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

### Accessing the DFSR:

To access the DFSR:

```
MRC p15,0,<Rt>,c5,c0,0 ; Read DFSR into Rt
MCR p15,0,<Rt>,c5,c0,0 ; Write Rt to DFSR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0101 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.42 DTLBIALL, Data TLB Invalidate All

The DTLBIALL characteristics are:

**Purpose**

Invalidate all data TLB entries for the PL1&0 translation regime, subject to the Security state and Privilege level at which the instruction is executed.

For details of the scope of this instruction see TLBIALL.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

DTLBIALL is a 32-bit system operation.

#### Field descriptions

The DTLBIALL operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the DTLBIALL operation:

To perform the DTLBIALL operation:

```
MCR p15,0,<Rt>,c8,c6,0 ; DTLBIALL operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0110 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.43 DTLBIASID, Data TLB Invalidate by ASID match

The DTLBIASID characteristics are:

**Purpose**

Invalidate data TLB entries for stage 1 of the PL1&0 translation regime that match the given ASID, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIASID.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

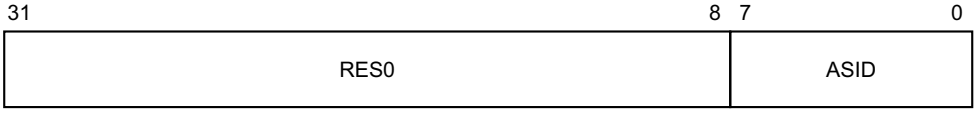
There are no configuration notes.

**Attributes**

DTLBIASID is a 32-bit system operation.

#### Field descriptions

The DTLBIASID input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this operation.

### Performing the DTLBIASID operation:

To perform the DTLBIASID operation:

```
MCR p15,0,<Rt>,c8,c6,2 ; DTLBIASID operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0110 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.44 DTLBIMVA, Data TLB Invalidate by VA

The DTLBIMVA characteristics are:

**Purpose**

Invalidate data TLB entries for stage 1 of the PL1&0 translation regime that match the given VA and ASID, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIMVA.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

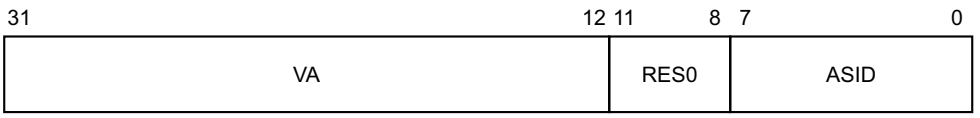
There are no configuration notes.

**Attributes**

DTLBIMVA is a 32-bit system operation.

#### Field descriptions

The DTLBIMVA input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

**Bits [11:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

### Performing the DTLBIMVA operation:

To perform the DTLBIMVA operation:

```
MCR p15,0,<Rt>,c8,c6,1 ; DTLBIMVA operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0110 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.45 ELR_hyp, Exception Link Register (Hyp mode)

The ELR_hyp characteristics are:

**Purpose**

When taking an exception to Hyp mode, holds the address to return to.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch32 System register ELR_hyp is architecturally mapped to AArch64 System register ELR_EL2.

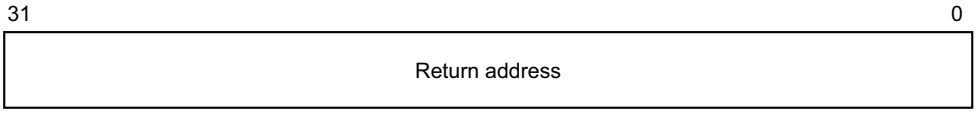
On a reset into an Exception level that is using AArch32 ELR_hyp is UNKNOWN.

**Attributes**

ELR_hyp is a 32-bit register.

#### Field descriptions

The ELR_hyp bit assignments are:




**Bits [31:0]**

Return address.

#### Accessing the ELR_hyp:

To access the ELR_hyp:

```
MRS <Rd>, ELR_hyp ; Read ELR_hyp into Rd
MSR ELR_hyp, <Rd> ; Write Rd to ELR_hyp
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| m | m1 | R |
|---|---|---|
| 1 | 1110 | 0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.46 FCSEIDR, FCSE Process ID register

The FCSEIDR characteristics are:

**Purpose**

Identifies whether the Fast Context Switch Extension (FCSE) is implemented.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

In ARMv8, the FCSE is not implemented, so this register is RAZ/WI. Software can access this register to determine that the implementation does not include the FCSE.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T13==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T13==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

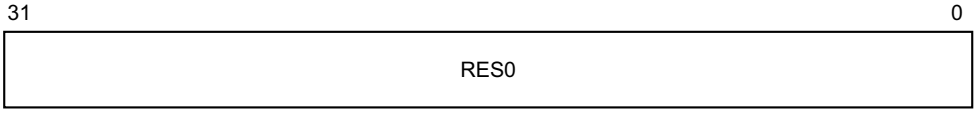
There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

FCSEIDR is a 32-bit register.

#### Field descriptions

The FCSEIDR bit assignments are:




**Bits [31:0]**

Reserved, RES0.

#### Accessing the FCSEIDR:

To access the FCSEIDR:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRC p15,0,<Rt>,c13,c0,0 ; Read FCSEIDR into Rt
MCR p15,0,<Rt>,c13,c0,0 ; Write Rt to FCSEIDR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1101 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.47 FPEXC, Floating-Point Exception Control register

The FPEXC characteristics are:

**Purpose**

Provides a global enable for the implemented Advanced SIMD and floating-point functionality, and reports floating-point status information.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | Config-RW | Config-RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | Config-RW | Config-RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CPTR_EL2.TFP==1, Non-secure EL1 accesses to this register are trapped to EL2.

If CPTR_EL3.TFP==1, EL1 and EL2 accesses to this register are trapped to EL3.

If HCPTR.TCP10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If CPACR.cp10==00, PL1 accesses to this register are UNDEFINED.

If HCPTR.TCP10==1, Non-secure EL2 accesses to this register are UNDEFINED.

If NSACR.cp10==0, Non-secure EL1 and EL2 accesses to this register are UNDEFINED.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register FPEXC is architecturally mapped to AArch64 System register FPEXC32_EL2.

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

Some or all RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

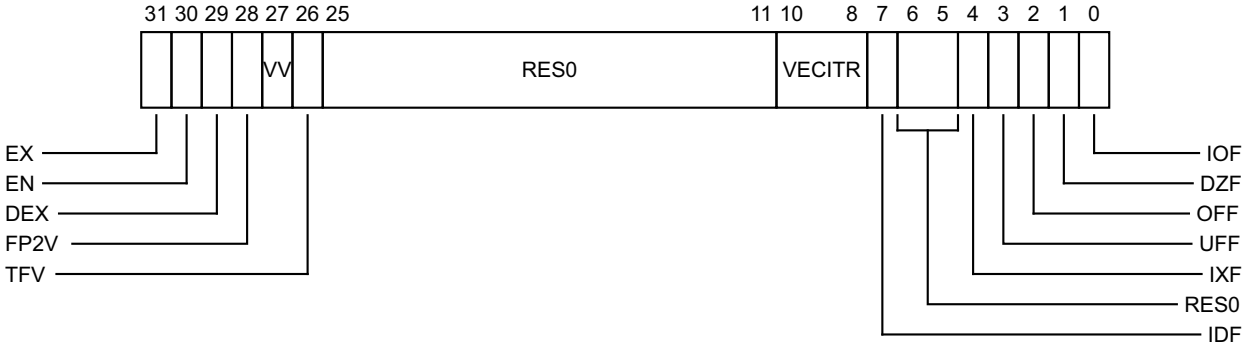
**Attributes**

FPEXC is a 32-bit register.

#### Field descriptions

The FPEXC bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**EX, bit [31]**

Exception bit. In ARMv8, this bit is RAZ/WI.

**EN, bit [30]**

Enables access to the Advanced SIMD and floating-point functionality from all Exception levels, except that setting this field to 0 does not disable the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

`0` Accesses to the FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers, are UNDEFINED at all Exception levels.

`1` This control permits access to the Advanced SIMD and floating-point functionality at all Exception levels.

Execution of floating-point and Advanced SIMD instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.
- If executing in Non-secure state:
  - HCPTR.TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
  - CPACR.ASEDIS.
  - If executing in Non-secure state, HCPTR.TASE and NSACR.NSTRCDIS.

See the descriptions of the controls for more information.

**Note**

When executing at EL0 using AArch32 with EL1 using AArch64, the PE behaves as if the value of FPEXC.EN bit is 1.

When this register has an architecturally-defined reset value, this field resets to `0`.

**DEX, bit [29]**

Defined synchronous floating-point exception bit.

This bit indicates the cause of an Undefined Instruction exception from the attempted execution of a floating-point instruction, and whether the FPEXC.TFV bit is valid.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The meaning of this bit is:

0 The exception occurred because of the execution of an unallocated instruction in the CP10 or CP11 encoding space. If the FPEXC.TFV bit is a RW bit then it is invalid and UNKNOWN. If the FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} bits are RW bits then they are invalid and UNKNOWN.

1 The FPEXC.TFV bit is valid and indicates the cause of the exception.

On an exception caused by a trapped floating-point exception, the exception-handling routine must clear this bit to 0.

On an implementation that both does not support trapping of floating-point exceptions and implements the FPSCR.{Stride, Len} fields as RAZ, this bit is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**FP2V, bit [28]**

FPINST2 instruction valid bit. In ARMv8, this bit is RES0.

**VV, bit [27]**

VECITR valid bit. In ARMv8, this bit is RES0.

**TFV, bit [26]**

Trapped Fault Valid bit. Valid only when the value of FPEXC.DEX is 1. When valid, it indicates the cause of the exception and therefore whether the FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} bits are valid.

0 The exception was caused by the execution of a floating-point VABS, VADD, VDIV, VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VMOV, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VSQRT, or VSUB instruction when one or both of FPSCR.{Stride, Len} was non-zero. If the FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} bits are RW then they are invalid and UNKNOWN.

1 FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} indicate the presence of trapped floating-point exceptions that had occurred at the time of the exception. Bits are set for all trapped exceptions that had occurred at the time of the exception.

This bit returns a status value and ignores writes.

When the value of FPEXC.DEX is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

On an implementation that supports the trapping of floating-point exceptions and implements FPSCR.{Stride, Len} as RAZ, this bit is RAO/WI.

**Bits [25:11]**

Reserved, RES0.

**VECITR, bits [10:8]**

Vector iteration count. In ARMv8, this field is RES1.

**IDF, bit [7]**

Input Denormal trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Input Denormal exception occurred while FPSCR.IDE was 1:

0 Input denormal exception has not occurred.

1 Input denormal exception has occurred.

Input Denormal exceptions can occur only when FPSCR.FZ is 1.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bits [6:5]**

Reserved, RES0.

**IXF, bit [4]**

Inexact trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Inexact exception occurred while FPSCR.IXE was 1:

0 Inexact exception has not occurred.

1 Inexact exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**UFF, bit [3]**

Underflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Underflow exception occurred while FPSCR.UFE was 1:

0 Underflow exception has not occurred.

1 Underflow exception has occurred.

Underflow trapped exceptions can occur only when FPSCR.FZ is 0.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**OFF, bit [2]**

Overflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Overflow exception occurred while FPSCR.OFE was 1:

0 Overflow exception has not occurred.

1 Overflow exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**DZF, bit [1]**

Divide-by-zero trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether a Divide-by-zero exception occurred while FPSCR.DZE was 1:

0 Divide-by-zero exception has not occurred.

1 Divide-by-zero exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**IOF, bit [0]**

Invalid Operation trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Invalid Operation exception occurred while FPSCR.IOE was 1:

`0` Invalid Operation exception has not occurred.

`1` Invalid Operation exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

## Accessing the FPEXC:

To access the FPEXC:

```
VMRS <Rt>, FPEXC ; Read FPEXC into Rt
VMSR FPEXC, <Rt> ; Write Rt to FPEXC
```

Register access is encoded as follows:

| spec_reg |
|---|
| 1000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.48 FPSCR, Floating-Point Status and Control Register

The FPSCR characteristics are:

**Purpose**

Provides floating-point system status information and control.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | Config-RW | Config-RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | Config-RW | Config-RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CPACR_EL1.FPEN==00, PL0 accesses to this register are trapped to EL1.

If CPACR_EL1.FPEN==01, PL0 accesses to this register are trapped to EL1.

If CPACR_EL1.FPEN==10, PL0 accesses to this register are trapped to EL1.

If CPTR_EL2.TFP==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If CPTR_EL3.TFP==1, EL0, EL1, and EL2 accesses to this register are trapped to EL3.

If HCPTR.TCP10==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If CPACR.cp10==00, PL0 and PL1 accesses to this register are UNDEFINED.

If CPACR.cp10==01, PL0 accesses to this register are UNDEFINED.

If HCPTR.TCP10==1, Non-secure EL2 accesses to this register are UNDEFINED.

If NSACR.cp10==0, Non-secure EL0, EL1, and EL2 accesses to this register are UNDEFINED.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

The named fields in this register map to the equivalent fields in the AArch64 FPCR and FPSR.

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to non-zero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

RW fields in this register reset to architecturally UNKNOWN values.

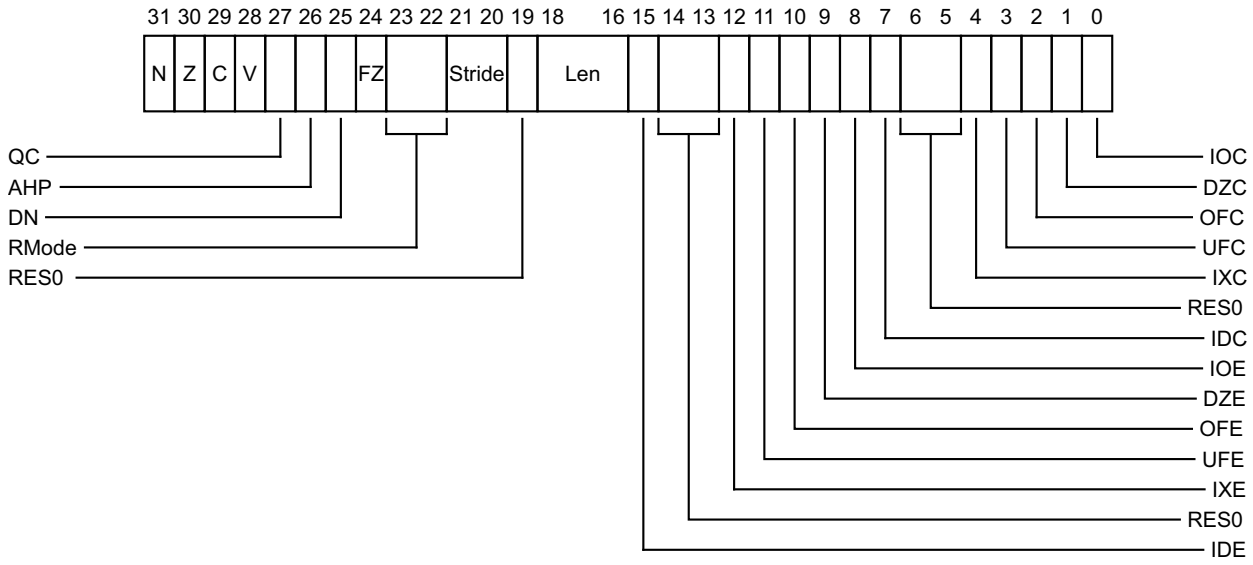
**Attributes**

FPSCR is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The FPSCR bit assignments are:




**N, bit [31]**

Negative condition flag. This is updated by floating-point comparison operations.

**Z, bit [30]**

Zero condition flag. This is updated by floating-point comparison operations.

**C, bit [29]**

Carry condition flag. This is updated by floating-point comparison operations.

**V, bit [28]**

Overflow condition flag. This is updated by floating-point comparison operations.

**QC, bit [27]**

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.

**AHP, bit [26]**

Alternative half-precision control bit:

`0` IEEE half-precision format selected.

`1` Alternative half-precision format selected.

**DN, bit [25]**

Default NaN mode control bit:

`0` NaN operands propagate through to the output of a floating-point operation.

`1` Any operation involving one or more NaNs returns the Default NaN.

The value of this bit only controls scalar floating-point arithmetic. Advanced SIMD arithmetic always uses the Default NaN setting, regardless of the value of the DN bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**FZ, bit [24]**

Flush-to-zero mode control bit:

0 Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.

1 Flush-to-zero mode enabled.

The value of this bit only controls scalar floating-point arithmetic. Advanced SIMD arithmetic always uses the Flush-to-zero setting, regardless of the value of the FZ bit.

**RMode, bits [23:22]**

Rounding Mode control field. The encoding of this field is:

00 Round to Nearest (RN) mode

01 Round towards Plus Infinity (RP) mode

10 Round towards Minus Infinity (RM) mode

11 Round towards Zero (RZ) mode.

The specified rounding mode is used by almost all scalar floating-point instructions. Advanced SIMD arithmetic always uses the Round to Nearest setting, regardless of the value of the RMode bits.

**Stride, bits [21:20]**

It is IMPLEMENTATION DEFINED whether this field is RW or RAZ.

If this field is RW and is set to a value other than zero, some floating-point instruction encodings are UNDEFINED. The instruction pseudocode identifies these instructions.

ARM strongly recommends that software never sets this field to a value other than zero.

The value of this field is ignored when processing Advanced SIMD instructions.

**Bit [19]**

Reserved, RES0.

**Len, bits [18:16]**

It is IMPLEMENTATION DEFINED whether this field is RW or RAZ.

If this field is RW and is set to a value other than zero, some floating-point instruction encodings are UNDEFINED. The instruction pseudocode identifies these instructions.

ARM strongly recommends that software never sets this field to a value other than zero.

The value of this field is ignored when processing Advanced SIMD instructions.

**IDE, bit [15]**

Input Denormal exception trap enable. Possible values are:

0 Untrapped exception handling selected. If the floating-point exception occurs then the IDC bit is set to 1.

1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IDC bit. The trap handling software can decide whether to set the IDC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RES0.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

**Bits [14:13]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IXE, bit [12]**

Inexact exception trap enable. Possible values are:

0 Untrapped exception handling selected. If the floating-point exception occurs then the IXC bit is set to 1.

1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IXC bit. The trap handling software can decide whether to set the IXC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RES0.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

**UFE, bit [11]**

Underflow exception trap enable. Possible values are:

0 Untrapped exception handling selected. If the floating-point exception occurs then the UFC bit is set to 1.

1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the UFC bit. The trap handling software can decide whether to set the UFC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RES0.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

**OFE, bit [10]**

Overflow exception trap enable. Possible values are:

0 Untrapped exception handling selected. If the floating-point exception occurs then the OFC bit is set to 1.

1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the OFC bit. The trap handling software can decide whether to set the OFC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RES0.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

**DZE, bit [9]**

Division by Zero exception trap enable. Possible values are:

0 Untrapped exception handling selected. If the floating-point exception occurs then the DZC bit is set to 1.

1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the DZC bit. The trap handling software can decide whether to set the DZC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RES0.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

**IOE, bit [8]**

Invalid Operation exception trap enable. Possible values are:

0 Untrapped exception handling selected. If the floating-point exception occurs then the IOC bit is set to 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IOC bit. The trap handling software can decide whether to set the IOC bit to 1.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RES0.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

**IDC, bit [7]**

Input Denormal cumulative exception bit. This bit is set to 1 to indicate that the Input Denormal exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IDE bit.

Advanced SIMD instructions set this bit if the Input Denormal exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IDE bit.

**Bits [6:5]**

Reserved, RES0.

**IXC, bit [4]**

Inexact cumulative exception bit. This bit is set to 1 to indicate that the Inexact exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IXE bit.

Advanced SIMD instructions set this bit if the Inexact exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IXE bit.

**UFC, bit [3]**

Underflow cumulative exception bit. This bit is set to 1 to indicate that the Underflow exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the UFE bit.

Advanced SIMD instructions set this bit if the Underflow exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the UFE bit.

**OFC, bit [2]**

Overflow cumulative exception bit. This bit is set to 1 to indicate that the Overflow exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the OFE bit.

Advanced SIMD instructions set this bit if the Overflow exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the OFE bit.

**DZC, bit [1]**

Division by Zero cumulative exception bit. This bit is set to 1 to indicate that the Division by Zero exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the DZE bit.

Advanced SIMD instructions set this bit if the Division by Zero exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the DZE bit.

**IOC, bit [0]**

Invalid Operation cumulative exception bit. This bit is set to 1 to indicate that the Invalid Operation exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IOE bit.

Advanced SIMD instructions set this bit if the Invalid Operation exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IOE bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the FPSCR:

To access the FPSCR:

```
VMRS <Rt>, FPSCR ; Read FPSCR into Rt
VMSR FPSCR, <Rt> ; Write Rt to FPSCR
```

Register access is encoded as follows:

| spec_reg |
|---|
| 0001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.49 FPSID, Floating-Point System ID register

The FPSID characteristics are:

**Purpose**

Provides top-level information about the floating-point implementation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | Config-RW | Config-RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | Config-RW | Config-RW |

When access to this register is permitted, write accesses are ignored.

This register largely duplicates information held in the MIDR. ARM deprecates use of it.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CPTR_EL2.TFP==1, Non-secure EL1 accesses to this register are trapped to EL2.

If CPTR_EL3.TFP==1, EL1 and EL2 accesses to this register are trapped to EL3.

If HCPTR.TCP10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HCR.TID0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If CPACR.cp10==00, PL1 accesses to this register are UNDEFINED.

If HCPTR.TCP10==1, Non-secure EL2 accesses to this register are UNDEFINED.

If NSACR.cp10==0, Non-secure EL1 and EL2 accesses to this register are UNDEFINED.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

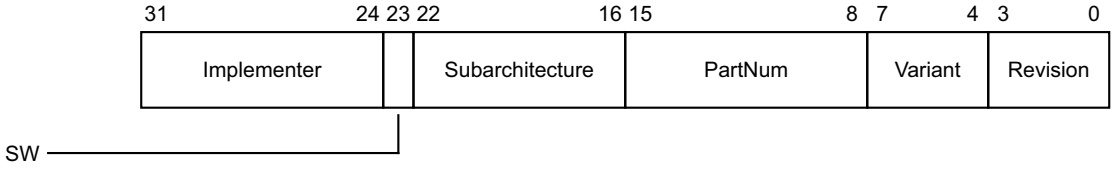
**Attributes**

FPSID is a 32-bit register.

#### Field descriptions

The FPSID bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Implementer, bits [31:24]**

Implementer codes are the same as those used for the MIDR.

For an implementation by ARM this field is `0x41`, the ASCII code for A.

**SW, bit [23]**

Software bit. Defined values are:

`0` The system includes hardware support for the floating-point instructions provided by the Floating-point extension.

`1` The system provides only software emulation of floating-point instructions provided by the Floating-point extension.

In ARMv8-A the only permitted value is `0`.

**Subarchitecture, bits [22:16]**

Subarchitecture version number. For an implementation by ARM, defined values are:

`0000000` VFPv1 architecture with an IMPLEMENTATION DEFINED subarchitecture.

`0000001` VFPv2 architecture with Common VFP subarchitecture v1.

`0000010` VFPv3 architecture, or later, with Common VFP subarchitecture v2. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.

`0000011` VFPv3 architecture, or later, with Null subarchitecture. The entire floating-point implementation is in hardware, and no software support code is required. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers. This value can be used only by an implementation that does not support the trap enable bits in the FPSCR.

`0000100` VFPv3 architecture, or later, with Common VFP subarchitecture v3, and support for trap enable bits in FPSCR. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.

For a subarchitecture designed by ARM the most significant bit of this field, register bit[22], is 0. Values with a most significant bit of 0 that are not listed here are reserved.

When the subarchitecture designer is not ARM, the most significant bit of this field, register bit[22], must be 1. Each implementer must maintain its own list of subarchitectures it has designed, starting at subarchitecture version number `0x40`.

**PartNum, bits [15:8]**

An IMPLEMENTATION DEFINED part number for the floating-point implementation, assigned by the implementer.

**Variant, bits [7:4]**

An IMPLEMENTATION DEFINED variant number. Typically, this field distinguishes between different production variants of a single product.

**Revision, bits [3:0]**

An IMPLEMENTATION DEFINED revision number for the floating-point implementation.

## Accessing the FPSID:

To access the FPSID:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
VMRS <Rt>, FPSID ; Read FPSID into Rt
VMSR FPSID, <Rt> ; Write Rt to FPSID
```

Register access is encoded as follows:

| spec_reg |
|---|
| 0000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.50 HACR, Hyp Auxiliary Configuration Register

The HACR characteristics are:

**Purpose**

Controls trapping to Hyp mode of IMPLEMENTATION DEFINED aspects of Non-secure EL1 or EL0 operation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HACR is architecturally mapped to AArch64 System register HACR_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

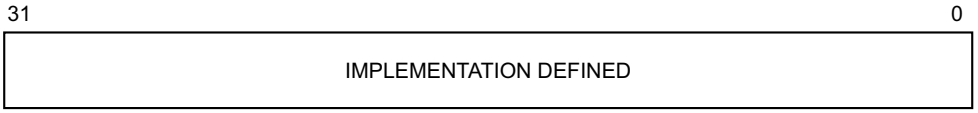
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HACR is a 32-bit register.

### Field descriptions

The HACR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the HACR:

To access the HACR:

```
MRC p15,4,<Rt>,c1,c1,7 ; Read HACR into Rt
MCR p15,4,<Rt>,c1,c1,7 ; Write Rt to HACR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0001 | 0001 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.51 HACTLR, Hyp Auxiliary Control Register

The HACTLR characteristics are:

**Purpose**

Controls IMPLEMENTATION DEFINED features of Hyp mode operation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HACTLR is architecturally mapped to AArch64 System register ACTLR_EL2[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

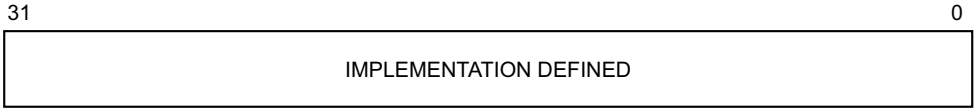
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HACTLR is a 32-bit register.

#### Field descriptions

The HACTLR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the HACTLR:

To access the HACTLR:

```
MRC p15,4,<Rt>,c1,c0,1 ; Read HACTLR into Rt
MCR p15,4,<Rt>,c1,c0,1 ; Write Rt to HACTLR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0001 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.52 HACTLR2, Hyp Auxiliary Control Register 2

The HACTLR2 characteristics are:

**Purpose**

Provides additional space to the HACTLR register to hold IMPLEMENTATION DEFINED trap functionality.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HACTLR2 is architecturally mapped to AArch64 System register ACTLR_EL2[63:32].

It is IMPLEMENTATION DEFINED whether this register is implemented, or whether it causes UNDEFINED exceptions when accessed.

The implementation of this register can be detected by examining bits [7:4] of the ID_MMFR4/ID_MMFR4_EL1 register.

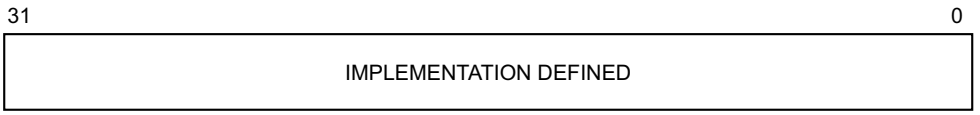
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HACTLR2 is a 32-bit register.

#### Field descriptions

The HACTLR2 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

## Accessing the HACTLR2:

To access the HACTLR2:

```
MRC p15,4,<Rt>,c1,c0,3 ; Read HACTLR2 into Rt
MCR p15,4,<Rt>,c1,c0,3 ; Write Rt to HACTLR2
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0001 | 0000 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.53 HADFSR, Hyp Auxiliary Data Fault Status Register

The HADFSR characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED syndrome information for Data Abort exceptions taken to Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T5==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T5==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HADFSR is architecturally mapped to AArch64 System register AFSR0_EL2.

This is an optional register. An implementation that does not require this register can implement it as RES0.

If EL2 is not implemented, this register is RES0 from EL3.

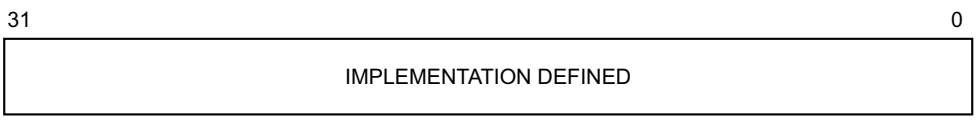
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HADFSR is a 32-bit register.

#### Field descriptions

The HADFSR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the HADFSR:

To access the HADFSR:

```
MRC p15,4,<Rt>,c5,c1,0 ; Read HADFSR into Rt
MCR p15,4,<Rt>,c5,c1,0 ; Write Rt to HADFSR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0101 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.54 HAIFSR, Hyp Auxiliary Instruction Fault Status Register

The HAIFSR characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED syndrome information for Prefetch Abort exceptions taken to Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T5==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T5==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HAIFSR is architecturally mapped to AArch64 System register AFSR1_EL2.

This is an optional register. An implementation that does not require this register can implement it as RES0.

If EL2 is not implemented, this register is RES0 from EL3.

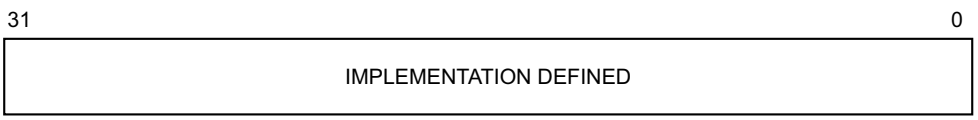
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HAIFSR is a 32-bit register.

#### Field descriptions

The HAIFSR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the HAIFSR:

To access the HAIFSR:

```
MRC p15,4,<Rt>,c5,c1,1 ; Read HAIFSR into Rt
MCR p15,4,<Rt>,c5,c1,1 ; Write Rt to HAIFSR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0101 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.55 HAMAIR0, Hyp Auxiliary Memory Attribute Indirection Register 0

The HAMAIR0 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED memory attributes for the memory attribute encodings defined by HMAIR0. These IMPLEMENTATION DEFINED attributes can only provide additional qualifiers for the memory attribute encodings, and cannot change the memory attributes defined in HMAIR0.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

If an implementation does not provide any IMPLEMENTATION DEFINED memory attributes, this register is RES0.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T10==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HAMAIR0 is architecturally mapped to AArch64 System register AMAIR_EL2[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

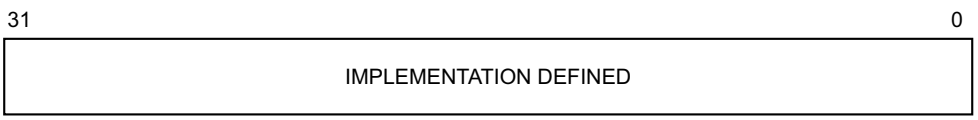
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HAMAIR0 is a 32-bit register.

#### Field descriptions

The HAMAIR0 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

## Accessing the HAMAIR0:

To access the HAMAIR0:

```
MRC p15,4,<Rt>,c10,c3,0 ; Read HAMAIR0 into Rt
MCR p15,4,<Rt>,c10,c3,0 ; Write Rt to HAMAIR0
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1010 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.56 HAMAIR1, Hyp Auxiliary Memory Attribute Indirection Register 1

The HAMAIR1 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED memory attributes for the memory attribute encodings defined by HMAIR1. These IMPLEMENTATION DEFINED attributes can only provide additional qualifiers for the memory attribute encodings, and cannot change the memory attributes defined in HMAIR1.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

If an implementation does not provide any IMPLEMENTATION DEFINED memory attributes, this register is RES0.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T10==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HAMAIR1 is architecturally mapped to AArch64 System register AMAIR_EL2[63:32].

If EL2 is not implemented, this register is RES0 from EL3.

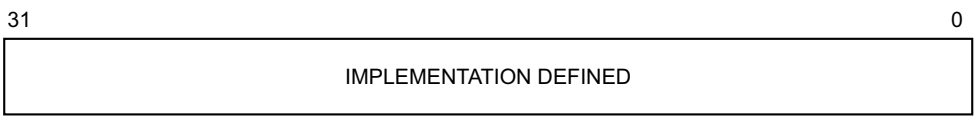
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HAMAIR1 is a 32-bit register.

#### Field descriptions

The HAMAIR1 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

## Accessing the HAMAIR1:

To access the HAMAIR1:

```
MRC p15,4,<Rt>,c10,c3,1 ; Read HAMAIR1 into Rt
MCR p15,4,<Rt>,c10,c3,1 ; Write Rt to HAMAIR1
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1010 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.57 HCPTR, Hyp Architectural Feature Trap Register

The HCPTR characteristics are:

**Purpose**

Controls:

- Trapping to Hyp mode of Non-secure access, at EL1 or EL0, to Trace, floating-point, and Advanced SIMD functionality.
- Hyp mode access to Trace, floating-point, and Advanced SIMD functionality.

**Note**

Accesses to this functionality:

- From Non-secure modes other than Hyp mode are also affected by settings in the CPACR and NSACR.
- From Hyp mode are also affected by settings in the NSACR.

Exceptions generated by the CPACR and NSACR controls are higher priority than those generated by the HCPTR controls.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CPTR_EL3.TCPAC==1, EL2 accesses to this register are trapped to EL3.

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HCPTR is architecturally mapped to AArch64 System register CPTR_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 with EL2 using AArch32, or into EL3 with EL3 using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

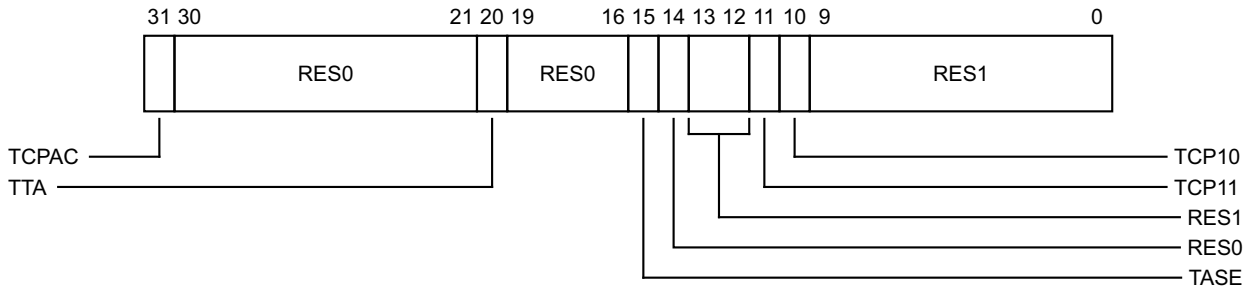
**Attributes**

HCPTR is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The HCPTR bit assignments are:




**TCPAC, bit [31]**

Traps Non-secure EL1 accesses to the CPACR to Hyp mode.

`0` This control has no effect on Non-secure EL1 accesses to the CPACR.

`1` Non-secure EL1 accesses to the CPACR are trapped to Hyp mode.

**Note**

The CPACR is not accessible at EL0.

When this register has an architecturally-defined reset value, this field resets to `0`.

**Bits [30:21]**

Reserved, RES0.

**TTA, bit [20]**

Traps Non-secure System register accesses to all implemented trace registers to Hyp mode.

`0` This control has no effect on Non-secure System register accesses to trace registers.

`1` Any Non-secure System register access to an implemented trace register is trapped to Hyp mode, unless the access is trapped to EL1 by a CPACR or NSACR control, or the access is from Non-secure EL0 and the definition of the register in the appropriate trace architecture specification indicates that the register is not accessible from EL0. A trapped instruction generates:

- A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1.
- An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

If the implementation does not include a trace macrocell, or does not include a System register interface to the trace macrocell registers, it is IMPLEMENTATION DEFINED whether this bit:

- Is RES0.
- Is RES1.
- Can be written from Hyp mode, and from Secure Monitor mode when SCR.NS is 1.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSTRCDIS is 1, for all purposes other than reading or writing the value of this field the PE behaves as if the value of this field is 1.

**Note**

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the implementation includes an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and a resulting Undefined Instruction exception is higher priority than a HCPTR.TTA Hyp Trap exception.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The architecture does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

---

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**Bits [19:16]**

Reserved, RES0.

**TASE, bit [15]**

Traps Non-secure execution of Advanced SIMD instructions to Hyp mode when the value of HCPTR.TCP10 is 0.

`0` This control has no effect on Non-secure execution of Advanced SIMD instructions.

`1` When the value of HCPTR.TCP10 is 0, any attempt to execute an Advanced SIMD instruction in Non-secure state is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates:

- A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1.
- An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

When the value of HCPTR.TCP10 is 1, the value of this field is ignored.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, then it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSASEDIS is 1, then for all purposes other than reading or writing the value of this field the PE behaves as if the value of this field is 1. This applies even if the field is implemented as RAZ/WI.

For the list of instructions affected by this field, see *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-2332.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**Bit [14]**

Reserved, RES0.

**Bits [13:12]**

Reserved, RES1.

**TCP11, bit [11]**

The value of this field is ignored. If this field is programmed with a different value to the TCP10 bit then this field is UNKNOWN on a direct read of the HCPTR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

If EL3 is implemented and is using AArch32, and the value of NSACR.cp10 is 1, for all purposes other than reading or writing the value of this field the PE behaves as if the value of this field is 1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TCP10, bit [10]**

Trap Non-secure accesses to floating-point and Advanced SIMD functionality to Hyp mode:

`0` This control has no effect on Non-secure accesses to floating-point and Advanced SIMD functionality.

`1` Any attempted access to floating-point and Advanced SIMD functionality from Non-secure state is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates:

- A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1.
- An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

The floating-point and Advanced SIMD features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

If EL3 is implemented and is using AArch32, and the value of NSACR.cp10 is 1, for all purposes other than reading or writing the value of this field the PE behaves as if the value of this field is 1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**Bits [9:0]**

Reserved, RES1.

## Accessing the HCPTR:

To access the HCPTR:

```
MRC p15,4,<Rt>,c1,c1,2 ; Read HCPTR into Rt
MCR p15,4,<Rt>,c1,c1,2 ; Write Rt to HCPTR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0001 | 0001 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.58 HCR, Hyp Configuration Register

The HCR characteristics are:

**Purpose**

Provides configuration controls for virtualization, including defining whether various Non-secure operations are trapped to Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HCR is architecturally mapped to AArch64 System register HCR_EL2[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 with EL2 using AArch32, or into EL3 with EL3 using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

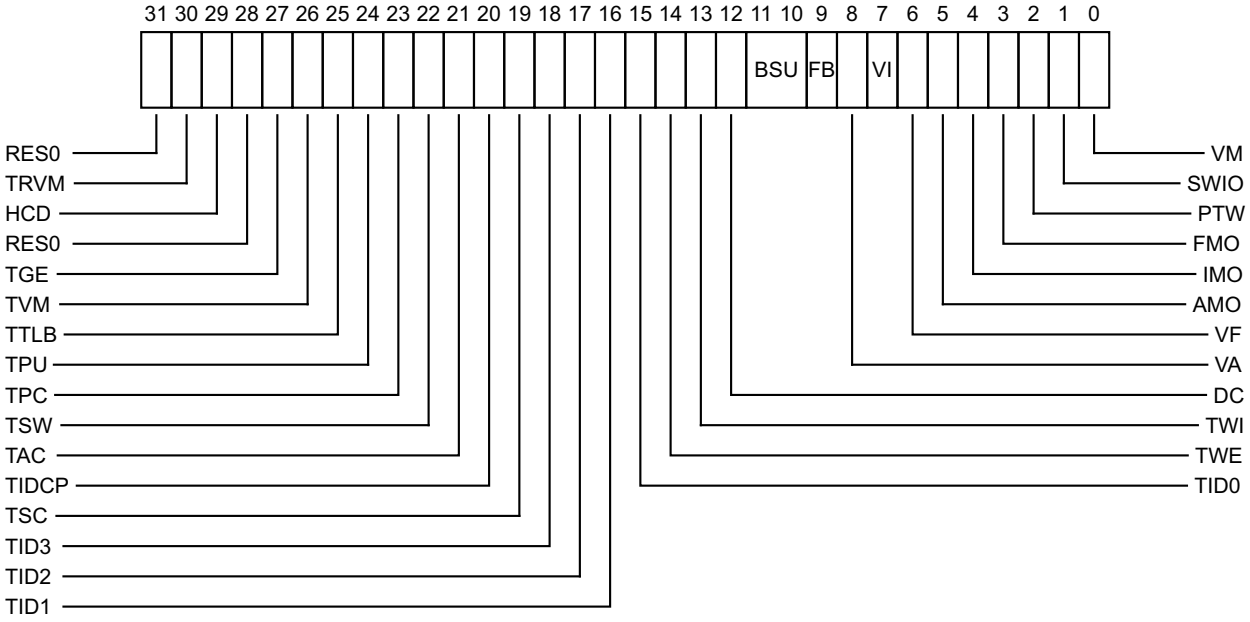
**Attributes**

HCR is a 32-bit register.

#### Field descriptions

The HCR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bit [31]**

Reserved, RES0.

**TRVM, bit [30]**

Trap Reads of Virtual Memory controls. Traps Non-secure EL1 reads of the virtual memory control registers to Hyp mode. The registers for which read accesses are trapped are as follows:

SCTLR, TTBR0, TTBR1, TTBCR, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

`0` This control has no effect on Non-secure EL1 read accesses to Virtual Memory controls.

`1` Non-secure EL1 read accesses to the specified Virtual Memory controls are trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to `0`.

**HCD, bit [29]**

HVC instruction disable. Disables Non-secure state execution of HVC instructions.

`0` HVC instruction execution is enabled at EL2 and Non-secure EL1.

`1` HVC instructions are UNDEFINED at EL2 and Non-secure EL1. The Undefined Instruction exception is taken to the Exception level at which the HVC instruction is executed.

**Note**

HVC instructions are always UNDEFINED at EL0.

This bit is only implemented if EL3 is not implemented. Otherwise, it is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**Bit [28]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TGE, bit [27]**

Trap General Exceptions, from Non-secure EL0.

`0` This control has no effect on execution at EL0.

`1` When the value of SCR.NS is 0, this control has no effect on execution at EL0.

When the value of SCR.NS is 1, then:

- All exceptions that would be routed to EL1 are routed to EL2.
- The SCTLR.M bit is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR.
- The HCR.{FMO, IMO, AMO} bits are treated as being 1 for all purposes other than returning the result of a direct read of HCR.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.
- An exception return to EL1 is treated as an illegal exception return.
- Monitor mode execution of an MSR or CPS instruction that changes CPSR.M to a Non-secure EL1 mode is an illegal change to PSTATE.M. For more information see *Illegal changes to PSTATE.M* on page G1-3771.

Also, when HCR.TGE is 1:

- If EL3 is using AArch32, an attempt to change from a Secure PL1 mode to a Non-secure EL1 mode by changing SCR.NS from 0 to 1 results in SCR.NS remaining as 0.
- The HDCR.{TDRA, TDOSA, TDA} bits are ignored and treated as being 1 other than for the purpose of a direct read of HDCR.

In the following cases the field resets to 0:

- The PE resets into EL3 with EL3 using AArch32.
- The PE resets into EL2 with EL2 using AArch32.

Otherwise, the field reset value is architecturally UNKNOWN.

When this register has an architecturally-defined reset value, this field resets to `0`.

**TVM, bit [26]**

Trap Virtual Memory controls. Traps Non-secure EL1 writes to the virtual memory control registers to Hyp mode. The registers for which write accesses are trapped are as follows:

SCTLR, TTBR0, TTBR1, TTBCR, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

`0` This control has no effect on Non-secure EL1 write accesses to EL1 virtual memory control registers.

`1` Non-secure EL1 write accesses to EL1 virtual memory control registers are trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to `0`.

**TTLB, bit [25]**

Trap TLB maintenance instructions. Traps Non-secure EL1 execution of a TLBI instruction to Hyp mode. This applies to the following instructions:

TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS, ITLBIALL, ITLBIMVA, ITLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVAAL

`0` This control has no effect on Non-secure EL1 accesses to TLB maintenance instructions.

`1` Non-secure EL1 accesses to TLB maintenance instructions are trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to `0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TPU, bit [24]**

Trap cache maintenance instructions that operate to the Point of Unification. Traps Non-secure EL1 execution of those cache maintenance instructions to Hyp mode. This applies to the following instructions:

ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAU.

**Note**

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at PL0.

`0` This control has no effect on the execution of cache maintenance instructions.

`1` Non-secure EL1 execution of the specified instructions is trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to `0`.

**TPC, bit [23]**

Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps Non-secure EL1 execution of those cache maintenance instructions to Hyp mode. This applies to the following instructions:

DCIMVAC, DCCIMVAC, DCCMVAC.

**Note**

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at PL0.

`0` This control has no effect on the execution of cache maintenance instructions.

`1` Non-secure execution of the specified instructions is trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to `0`.

**TSW, bit [22]**

Trap data or unified cache maintenance instructions that operate by Set/Way. Traps Non-secure EL1 execution of those cache maintenance instructions by set/way to Hyp mode. This applies to the following instructions:

DCISW, DCCSW, DCCISW.

**Note**

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

`0` This control has no effect on the execution of cache maintenance instructions.

`1` Non-secure execution of the specified instructions is trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to `0`.

**TAC, bit [21]**

Trap Auxiliary Control Registers. Traps Non-secure EL1 accesses to the Auxiliary Control Registers to Hyp mode, from both Execution states. This applies to the following register accesses:

ACTLR and, if implemented, ACTLR2.

`0` This control has no effect on Non-secure EL1 accesses to the Auxiliary Control Registers.

`1` Non-secure EL1 accesses to the specified registers are trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to `0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TIDCP, bit [20]**

Trap IMPLEMENTATION DEFINED functionality. Traps Non-secure EL1 accesses to the encodings for IMPLEMENTATION DEFINED System Registers to Hyp mode.

MCR and MRC instructions as follows:

- All CP15, CRn==9, Opcode1 = {0-7}, CRm == {c0-c2, c5-c8}, opcode2 == {0-7}.
- All CP15, CRn==10, Opcode1 =={0-7}, CRm == {c0, c1, c4, c8}, opcode2 == {0-7}.
- All CP15, CRn==11, Opcode1=={0-7}, CRm == {c0-c8, c15}, opcode2 == {0-7}.

When HCR.TIDCP is set to 1, it is IMPLEMENTATION DEFINED whether any of this functionality accessed from Non-secure EL0 is trapped to Hyp mode. If it is not, it is UNDEFINED, and the PE takes an Undefined Instruction exception to Non-secure Undefined mode.

`0` This control has no effect on Non-secure EL1 and EL0 accesses to the System register encodings for IMPLEMENTATION DEFINED functionality.

`1` Non-secure EL1 accesses to the specified System register encodings for IMPLEMENTATION DEFINED functionality are trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to `0`.

**TSC, bit [19]**

Trap SMC instructions. Traps Non-secure EL1 execution of SMC instructions to Hyp mode.

`0` This control has no effect on execution of SMC instructions.

`1` Any attempt to execute an SMC instruction at Non-secure EL1 is trapped to Hyp mode, regardless of the value of SCR.SCD.

The ARMv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

**Note**

- This trap is only implemented if the implementation includes EL3.
- SMC instructions are always UNDEFINED at PL0.
- This bit traps execution of the SMC instruction. It is not a routing control for the SMC exception. Hyp Trap exceptions and SMC exceptions have different preferred return addresses.

When this register has an architecturally-defined reset value, this field resets to `0`.

**TID3, bit [18]**

Trap ID group 3. Traps Non-secure EL1 reads of the following registers to Hyp mode:

ID_PFR0, ID_PFR1, ID_DFR0, ID_AFR0, ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5, MVFR0, MVFR1, MVFR2, and ID_MMFR4, except that if ID_MMFR4 is implemented as RAZ/WI then it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4 are trapped.

Also an MRC access to any of the following CP15 encodings:

- opc1 == 0, CRn == c0, CRm == {c3-c7}, opc2 == {0,1}.
- opc1 == 0, CRn == c0, CRm == c3, opc2 == 2.
- opc1 == 0, CRn == c0, CRm == c5, opc2 == {4,5}.

It is IMPLEMENTATION DEFINED whether this bit traps MRC accesses to CP15 encodings in the following range that are not already mentioned in this field description:

- opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}.

`0` This control has no effect on Non-secure EL1 reads of the ID group 3 registers.

`1` The specified Non-secure EL1 read accesses to ID group 3 registers are trapped to Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this register has an architecturally-defined reset value, this field resets to 0.

**TID2, bit [17]**

Trap ID group 2. Traps the following register accesses to Hyp mode:

- Non-secure EL1 and EL0 reads of the CTR, CCSIDR, CLIDR, and CSSELR.
- Non-secure EL1 and EL0 writes to the CSSELR.

0 This control has no effect on Non-secure EL1 and EL0 accesses to the ID group 2 registers.

1 The specified Non-secure EL1 and EL0 accesses to ID group 2 registers are trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to 0.

**TID1, bit [16]**

Trap ID group 1. Traps Non-secure EL1 reads of the following registers to Hyp mode:

TCMTR, TLBTR, REVIDR, AIDR.

0 This control has no effect on Non-secure EL1 reads of the ID group 1 registers.

1 The specified Non-secure EL1 read accesses to ID group 1 registers are trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to 0.

**TID0, bit [15]**

Trap ID group 0. Traps the following register accesses to Hyp mode:

- Non-secure EL1 reads of the JIDR.
- If the JIDR is RAZ from Non-secure EL0, Non-secure EL0 of the JIDR.
- Non-secure EL1 reads of the FPSID.

**Note**

- It is IMPLEMENTATION DEFINED whether the JIDR is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0 then the Undefined Instruction exception takes precedence over this trap.
- The FPSID is not accessible at PL0.
- When the FPSID is accessible, a VMSR FPSID, <Rt> instruction is permitted but is ignored. The execution of this instruction is not trapped by this trap.

0 This control has no effect on Non-secure EL1 reads of the ID group 0 registers.

1 The specified Non-secure EL1 read accesses to ID group 0 registers are trapped to Hyp mode.

When this register has an architecturally-defined reset value, this field resets to 0.

**TWE, bit [14]**

Traps Non-secure EL0 and EL1 execution of WFE instructions to Hyp mode:

0 This control has no effect on the execution of WFE instructions at Non-secure EL1 and Non-secure EL0.

1 Any attempt to execute a WFE instruction at Non-secure EL0 or EL1 is trapped to Hyp mode, if the instruction would otherwise have caused the PE to enter a low-power state, except that when the value of SCTLR.nTWE is 0, the trap of PL0 execution to Undefined mode takes precedence over this trap.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to `0`.

**TWI, bit [13]**

Traps Non-secure EL0 and EL1 execution of WFI instructions to Hyp mode.

`0` This control has no effect on the execution of WFI instructions at Non-secure EL1 and Non-secure EL0.

`1` Any attempt to execute a WFI instruction at Non-secure EL0 or EL1 is trapped to Hyp mode, if the instruction would otherwise have caused the PE to enter a low-power state, except that when the value of SCTLR.nTWI is 0, the trap of PL0 execution to Undefined mode takes precedence over this trap.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to `0`.

**DC, bit [12]**

Default cacheable.

`0` This control has no effect on the Non-secure EL1&0 translation regime.

`1` In Non-secure state:

- The SCTLR.M bit to behave as 0 for all purposes other than a direct read of the value of the bit.
- The HCR.VM bit to behave as 1 for all purposes other than a direct read of the value of the bit.
- When stage 1 translations of PL1&0 translation regime are disabled, the memory type produced by the first stage of the PL1&0 translation regime is Normal Non-Shareable, Inner WriteBack Read-WriteAllocate, Outer WriteBack Read-WriteAllocate.

When this bit is 0 and stage 1 translations of EL1&0 translation regime are disabled, the default memory attribute for Data accesses is Device-nGnRnE.

This bit has no effect on the EL2 and EL3 translation regimes.

This bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to `0`.

**BSU, bits [11:10]**

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from Non-secure EL1 or Non-secure EL0:

`00` No effect

`01` Inner Shareable

`10` Outer Shareable

`11` Full system


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.

When this register has an architecturally-defined reset value, this field resets to `0`.

**FB, bit [9]**

Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from Non-secure EL1:

BPIALL, TLBIALL, TLBIMVA, TLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, ITLBIALL, ITLBIMVA, ITLBIASID, TLBIMVAA, ICIALLU, TLBIMVAL, TLBIMVAAL.

`0` This field has no effect on the operation of the specified instructions.

`1` When one of the specified instruction is executed at Non-secure EL1, the instruction is broadcast within the Inner Shareable shareability domain.

When this register has an architecturally-defined reset value, this field resets to `0`.

**VA, bit [8]**

Virtual Asynchronous Abort exception.

`0` This mechanism is not making a virtual Asynchronous Abort pending.

`1` A virtual Asynchronous Abort is pending because of this mechanism.

The virtual Asynchronous Abort is only enabled when the value of HCR.AMO bit is 1.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

When this register has an architecturally-defined reset value, this field resets to `0`.

**VI, bit [7]**

Virtual IRQ exception.

`0` This mechanism is not making a virtual IRQ pending.

`1` A virtual IRQ is pending because of this mechanism.

The virtual IRQ is only enabled when the value of HCR.IMO bit is 1.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

When this register has an architecturally-defined reset value, this field resets to `0`.

**VF, bit [6]**

Virtual FIQ exception.

`0` This mechanism is not making a virtual FIQ pending.

`1` A virtual FIQ is pending because of this mechanism.

The virtual FIQ is only enabled when the value of HCR.FMO bit is 1.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

When this register has an architecturally-defined reset value, this field resets to `0`.

**AMO, bit [5]**

Asynchronous Abort Mask Override. When this bit is set to 1, it overrides the effect of CPSR.A, and enables virtual exception signaling by the VA bit.

If the value of HCR.TGE is 1, then in Non-secure state the HCR.AMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

When this register has an architecturally-defined reset value, this field resets to `0`.

**IMO, bit [4]**

IRQ Mask Override. When this bit is set to 1, it overrides the effect of CPSR.I, and enables virtual exception signaling by the VI bit.

If the value of HCR.TGE is 1, then in Non-secure state the HCR.IMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this register has an architecturally-defined reset value, this field resets to 0.

**FMO, bit [3]**

FIQ Mask Override. When this bit is set to 1, it overrides the effect of CPSR.F, and enables virtual exception signaling by the VF bit.

If the value of HCR.TGE is 1, then in Non-secure state the HCR.FMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

When this register has an architecturally-defined reset value, this field resets to 0.

**PTW, bit [2]**

Protected Table Walk. In the Non-secure PL1&0 translation regime, a translation table access made as part of a stage 1 translation table walk is subject to a stage 2 translation. The combining of the memory type attributes from the two stages of translation means the access might be made to a type of Device memory. If this occurs then the value of this bit determines the behavior:

0 The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.

1 The memory access generates a stage 2 Permission fault.

This bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to 0.

**SWIO, bit [1]**

Set/Way Invalidation Override. Causes Non-secure EL1 execution of the data cache invalidate by set/way instructions to be treated as data cache clean and invalidate by set/way.

0 This control has no effect on the operation of data cache invalidate by set/way instructions.

1 Data cache invalidate by set/way instructions operate as data cache clean and invalidate by set/way.

When this bit is set to 1, DCISW is executed as DCCISW.

As a result of changes to the behavior of DCISW, this bit is redundant in ARMv8. This bit can be implemented as RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to 0.

**VM, bit [0]**

Virtualization enable. Enables stage 2 address translation for the Non-secure EL1&0 translation regime. Possible values of this bit are:

0 Non-secure EL1&0 stage 2 address translation disabled.

1 Non-secure EL1&0 stage 2 address translation enabled.

If the HCR.DC bit is set to 1, then the behavior of the PE when executing in a Non-secure mode other than Hyp mode is consistent with HCR.VM being 1, regardless of the actual value of HCR.VM, other than the value returned by an explicit read of HCR.VM.

When the value of this bit is 1, data cache invalidate instructions executed at Non-secure EL1 operate as data cache clean and invalidate instructions. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR.SWIO bit.

This bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to 0.

## Accessing the HCR:

To access the HCR:

```
MRC p15,4,<Rt>,c1,c1,0 ; Read HCR into Rt
MCR p15,4,<Rt>,c1,c1,0 ; Write Rt to HCR
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0001 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.59 HCR2, Hyp Configuration Register 2

The HCR2 characteristics are:

**Purpose**

Provides additional configuration controls for virtualization.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HCR2 is architecturally mapped to AArch64 System register HCR_EL2[63:32].

If EL2 is not implemented, this register is RES0 from EL3.

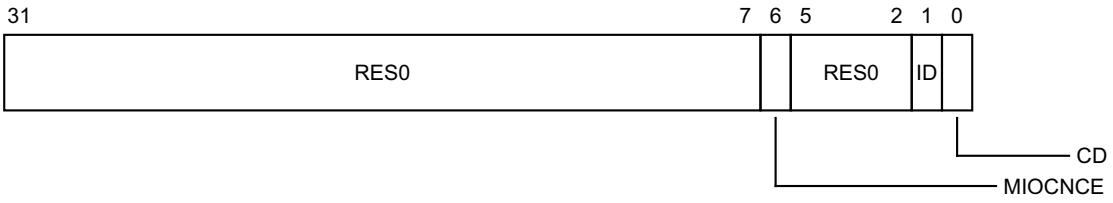
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 with EL2 using AArch32, or into EL3 with EL3 using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HCR2 is a 32-bit register.

#### Field descriptions

The HCR2 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:7]**

Reserved, RES0.

**MIOCNCE, bit [6]**

Mismatched Inner/Outer Cacheable Non-Coherency Enable, for the Non-secure PL1&0 translation regime.

`0` For the Non-secure PL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there must be no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

`1` For the Non-secure PL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there might be a loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

For more information see *Mismatched memory attributes* on page E2-2392.

The value of this field has no effect on translation regimes other than the Non-secure PL1&0 translation regime.

This field can be implemented as RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**Bits [5:2]**

Reserved, RES0.

**ID, bit [1]**

Stage 2 Instruction cache disable. For the Non-secure PL1&0 translation regime, when HCR.VM==1, this control forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

`0` This control has no effect on stage 2 of the Non-secure PL1&0 translation regime.

`1` For the Non-secure PL1&0 translation regime, forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

This bit has no effect on the EL2 translation regime.

When this register has an architecturally-defined reset value, this field resets to `0`.

**CD, bit [0]**

Stage 2 Data cache disable. When HCR.VM==1, this forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable for the Non-secure PL1&0 translation regime.

`0` This control has no effect on stage 2 of the Non-secure PL1&0 translation regime for data accesses and translation table walks.

`1` For the Non-secure PL1&0 translation regime, forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

This bit has no effect on the EL2 translation regime.

When this register has an architecturally-defined reset value, this field resets to `0`.

## Accessing the HCR2:

To access the HCR2:

```
MRC p15,4,<Rt>,c1,c1,4 ; Read HCR2 into Rt
MCR p15,4,<Rt>,c1,c1,4 ; Write Rt to HCR2
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0001 | 0001 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.60 HDFAR, Hyp Data Fault Address Register

The HDFAR characteristics are:

**Purpose**

Holds the virtual address of the faulting address that caused a synchronous Data Abort exception that is taken to Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

Any execution in a Non-secure EL1 mode, or in Secure state, makes the HDFAR UNKNOWN.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T6==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T6==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HDFAR is architecturally mapped to AArch64 System register FAR_EL2[31:0].

AArch32 System register HDFAR is architecturally mapped to AArch32 System register DFAR (S) when EL2 is implemented.

If EL2 is not implemented, this register is RES0 from EL3.

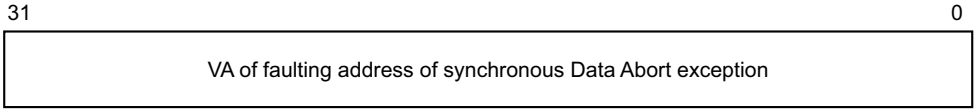
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HDFAR is a 32-bit register.

### Field descriptions

The HDFAR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:0]**

VA of faulting address of synchronous Data Abort exception.

## Accessing the HDFAR:

To access the HDFAR:

```
MRC p15,4,<Rt>,c6,c0,0 ; Read HDFAR into Rt
MCR p15,4,<Rt>,c6,c0,0 ; Write Rt to HDFAR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0110 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.61 HIFAR, Hyp Instruction Fault Address Register

The HIFAR characteristics are:

**Purpose**

Holds the virtual address of the faulting address that caused a synchronous Prefetch Abort exception that is taken to Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

Any execution in a Non-secure EL1 mode, or in Secure state, makes the HIFAR UNKNOWN.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T6==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T6==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HIFAR is architecturally mapped to AArch64 System register FAR_EL2[63:32].

AArch32 System register HIFAR is architecturally mapped to AArch32 System register IFAR (S) when EL2 is implemented.

If EL2 is not implemented, this register is RES0 from EL3.

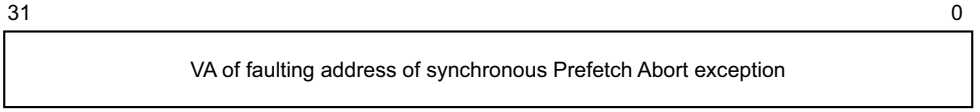
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HIFAR is a 32-bit register.

#### Field descriptions

The HIFAR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:0]**

VA of faulting address of synchronous Prefetch Abort exception.

### Accessing the HIFAR:

To access the HIFAR:

```
MRC p15,4,<Rt>,c6,c0,2 ; Read HIFAR into Rt
MCR p15,4,<Rt>,c6,c0,2 ; Write Rt to HIFAR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0110 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.62 HMAIR0, Hyp Memory Attribute Indirection Register 0

The HMAIR0 characteristics are:

**Purpose**

Along with HMAIR1, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations for memory accesses from Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

AttrIndx[2], from the translation table descriptor, selects the appropriate HMAIR: setting AttrIndx[2] to 0 selects HMAIR0.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T10==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HMAIR0 is architecturally mapped to AArch64 System register MAIR_EL2[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

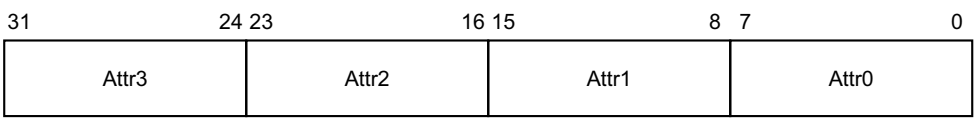
**Attributes**

HMAIR0 is a 32-bit register.

### Field descriptions

The HMAIR0 bit assignments are:

***When TTBCR.EAE==1:***





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attr<n>, bits [8n+7:8n], for n = 0 to 3**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

| Attr<n>[7:4] | Meaning |
|---|---|
| 0000 | Device memory. See encoding of Attr<n>[3:0] for the type of Device memory. |
| 00RW, RW not 00 | Normal Memory, Outer Write-through transient |
| 0100 | Normal Memory, Outer Non-cacheable |
| 01RW, RW not 00 | Normal Memory, Outer Write-back transient |
| 10RW | Normal Memory, Outer Write-through non-transient |
| 11RW | Normal Memory, Outer Write-back non-transient |

R = Outer Read Allocate Policy, W = Outer Write Allocate Policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

| Attr<n>[3:0] | Meaning when Attr<n>[7:4] is 0000 | Meaning when Attr<n>[7:4] is not 0000 |
|---|---|---|
| 0000 | Device-nGnRnE memory | UNPREDICTABLE |
| 00RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through transient |
| 0100 | Device-nGnRE memory | Normal memory, Inner Non-cacheable |
| 01RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back transient |
| 1000 | Device-nGRE memory | Normal Memory, Inner Write-through non-transient (RW=00) |
| 10RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through non-transient |
| 1100 | Device-GRE memory | Normal Memory, Inner Write-back non-transient (RW=00) |
| 11RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back non-transient |

R = Inner Read Allocate Policy, W = Inner Write Allocate Policy.

The R and W bits in some Attr<n> fields have the following meanings:

| R or W | Meaning |
|---|---|
| 0 | Do not allocate |
| 1 | Allocate |

## Accessing the HMAIR0:

To access the HMAIR0:

```
MRC p15,4,<Rt>,c10,c2,0 ; Read HMAIR0 into Rt
MCR p15,4,<Rt>,c10,c2,0 ; Write Rt to HMAIR0
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1010 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.63 HMAIR1, Hyp Memory Attribute Indirection Register 1

The HMAIR1 characteristics are:

**Purpose**

Along with HMAIR0, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations for memory accesses from Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

AttrIndx[2], from the translation table descriptor, selects the appropriate HMAIR: setting AttrIndx[2] to 1 selects HMAIR1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T10==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HMAIR1 is architecturally mapped to AArch64 System register MAIR_EL2[63:32].

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

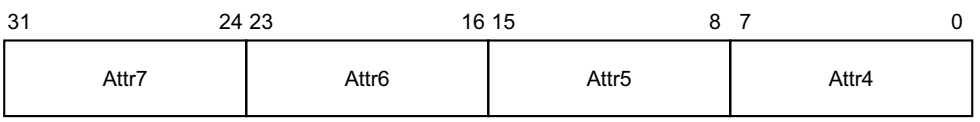
**Attributes**

HMAIR1 is a 32-bit register.

## Field descriptions

The HMAIR1 bit assignments are:

***When TTBCR.EAE==1:***





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attr<n>, bits [8(n-4)+7:8(n-4)], for n = 4 to 7**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

| Attr<n>[7:4] | Meaning |
|---|---|
| 0000 | Device memory. See encoding of Attr<n>[3:0] for the type of Device memory. |
| 00RW, RW not 00 | Normal Memory, Outer Write-through transient |
| 0100 | Normal Memory, Outer Non-cacheable |
| 01RW, RW not 00 | Normal Memory, Outer Write-back transient |
| 10RW | Normal Memory, Outer Write-through non-transient |
| 11RW | Normal Memory, Outer Write-back non-transient |

R = Outer Read Allocate Policy, W = Outer Write Allocate Policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

| Attr<n>[3:0] | Meaning when Attr<n>[7:4] is 0000 | Meaning when Attr<n>[7:4] is not 0000 |
|---|---|---|
| 0000 | Device-nGnRnE memory | UNPREDICTABLE |
| 00RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through transient |
| 0100 | Device-nGnRE memory | Normal memory, Inner Non-cacheable |
| 01RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back transient |
| 1000 | Device-nGRE memory | Normal Memory, Inner Write-through non-transient (RW=00) |
| 10RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through non-transient |
| 1100 | Device-GRE memory | Normal Memory, Inner Write-back non-transient (RW=00) |
| 11RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back non-transient |

R = Inner Read Allocate Policy, W = Inner Write Allocate Policy.

The R and W bits in some Attr<n> fields have the following meanings:

| R or W | Meaning |
|---|---|
| 0 | Do not allocate |
| 1 | Allocate |

### Accessing the HMAIR1:

To access the HMAIR1:

```
MRC p15,4,<Rt>,c10,c2,1 ; Read HMAIR1 into Rt
MCR p15,4,<Rt>,c10,c2,1 ; Write Rt to HMAIR1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1010 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.64 HPFAR, Hyp IPA Fault Address Register

The HPFAR characteristics are:

**Purpose**

Holds the faulting IPA for some aborts on a stage 2 translation taken to Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

Execution in any Non-secure mode other than Hyp mode makes this register UNKNOWN.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T6==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T6==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HPFAR is architecturally mapped to AArch64 System register HPFAR_EL2[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

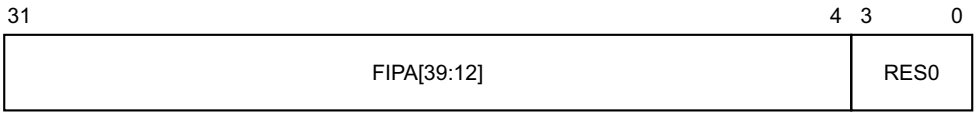
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HPFAR is a 32-bit register.

### Field descriptions

The HPFAR bit assignments are:




**FIPA[39:12], bits [31:4]**

Bits [39:12] of the faulting intermediate physical address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [3:0]**

Reserved, RES0.

## Accessing the HPFAR:

To access the HPFAR:

```
MRC p15,4,<Rt>,c6,c0,4 ; Read HPFAR into Rt
MCR p15,4,<Rt>,c6,c0,4 ; Write Rt to HPFAR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0110 | 0000 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.65 HRMR, Hyp Reset Management Register

The HRMR characteristics are:

**Purpose**

If EL2 is the highest implemented Exception level:

- A write to the register can request a Warm reset.
- If EL2 can use AArch32 and AArch64, specifies the Execution state that the PE boots into on a Warm reset.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T12==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T12==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HRMR is architecturally mapped to AArch64 System register RMR_EL2.

Only implemented if EL2 is the highest implemented Exception level. In this case:

- If EL2 can use AArch32 and AArch64 then this register must be implemented.
- If EL2 cannot use AArch64 then it is IMPLEMENTATION DEFINED whether the register is implemented.

When this register is not implemented its encoding is UNDEFINED.

See the field descriptions for the reset values. These apply whenever the register is implemented.

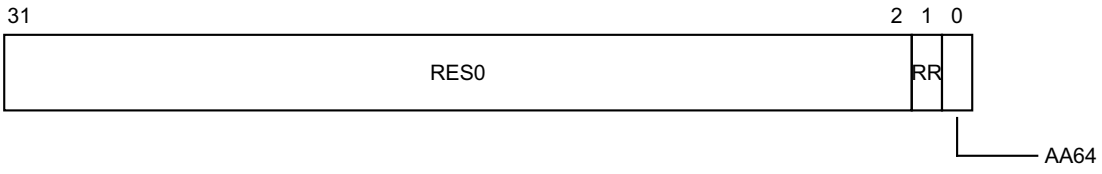
**Attributes**

HRMR is a 32-bit register.

### Field descriptions

The HRMR bit assignments are:

***When EL2 implemented, EL3 not implemented:***




**Bits [31:2]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**RR, bit [1]**

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to `0`. This reset value applies on a Cold reset and on a Warm reset.

**AA64, bit [0]**

When EL2 can use AArch32, determines which Execution state the PE boots into after a Warm reset:

`0` AArch32.

`1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If EL2 cannot use AArch64 this bit is RAZ/WI.

If this field is implemented as an RW field, this field resets to `0`. This reset value applies on a Cold reset.

### Accessing the HRMR:

To access the HRMR when EL2 implemented, EL3 not implemented:

```
MRC p15,4,<Rt>,c12,c0,2 ; Read HRMR into Rt
MCR p15,4,<Rt>,c12,c0,2 ; Write Rt to HRMR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1100 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.66 HSCTLR, Hyp System Control Register

The HSCTLR characteristics are:

**Purpose**

Provides top level control of the system operation in Hyp mode. This register provides Hyp mode control of features controlled by the Banked SCTLR bits, and shows the values of the non-Banked SCTLR bits.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HSCTLR is architecturally mapped to AArch64 System register SCTLR_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 with EL2 using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

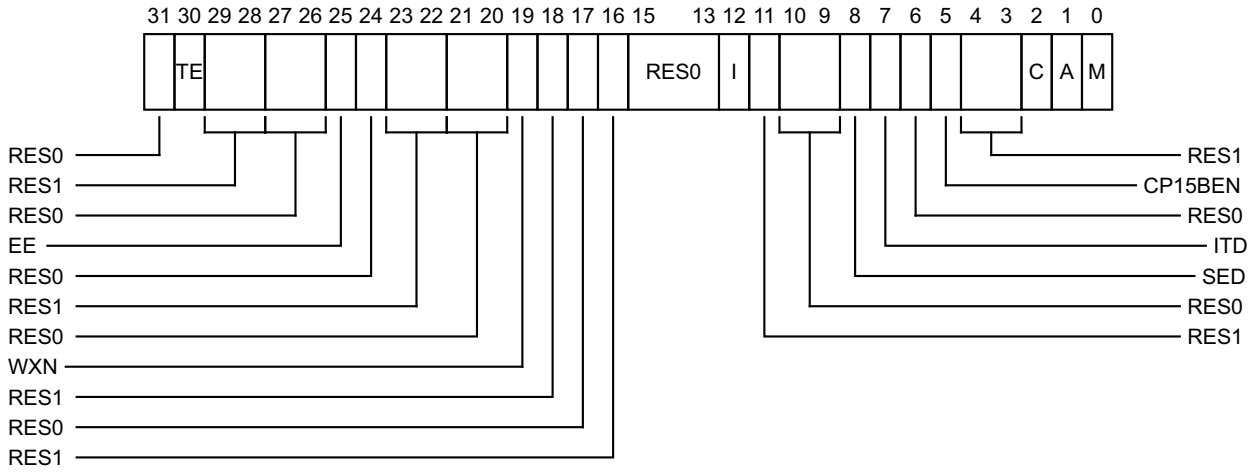
**Attributes**

HSCTLR is a 32-bit register.

### Field descriptions

The HSCTLR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bit [31]**

Reserved, RES0.

**TE, bit [30]**

T32 Exception Enable. This bit controls whether exceptions to EL2 are taken to A32 or T32 state:

0 Exceptions, including reset, taken to A32 state.

1 Exceptions, including reset, taken to T32 state.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bits [29:28]**

Reserved, RES1.

**Bits [27:26]**

Reserved, RES0.

**EE, bit [25]**

The value of the PSTATE.E bit on entry to Hyp mode, the endianness of stage 1 translation table walks in the PL2 translation regime, and the endianness of stage 2 translation table walks in the PL1&0 translation regime.

The possible values of this bit are:

0 Little-endian. PSTATE.E is cleared to 0 on entry to Hyp mode. Stage 1 translation table walks in the PL2 translation regime, and stage 2 translation table walks in the PL1&0 translation regime are little-endian.

1 Big-endian. PSTATE.E is set to 1 on entry to Hyp mode. Stage 1 translation table walks in the PL2 translation regime, and stage 2 translation table walks in the PL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception Levels higher than EL0, this bit is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to an IMPLEMENTATION DEFINED value.

**Bit [24]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [23:22]**

Reserved, RES1.

**Bits [21:20]**

Reserved, RES0.

**WXN, bit [19]**

Write permission implies XN (Execute-never). For the EL2 translation regime, this bit can force all memory regions that are writeable to be treated as XN. The possible values of this bit are:

`0` This control has no effect on memory access permissions.

`1` Any region that is writeable in the EL2 translation regime is forced to XN for accesses from software executing at EL2.

The WXN bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bit [18]**

Reserved, RES1.

**Bit [17]**

Reserved, RES0.

**Bit [16]**

Reserved, RES1.

**Bits [15:13]**

Reserved, RES0.

**I, bit [12]**

Instruction cache enable. This is an enable bit for instruction caches at EL2:

`0` All instruction access to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache.

If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

`1` All instruction access to Normal memory from EL2 can be cached at all levels of instruction and unified cache.

If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the PL1&0 translation regime.

When this register has an architecturally-defined reset value, this field resets to `0`.

**Bit [11]**

Reserved, RES1.

**Bits [10:9]**

Reserved, RES0.

**SED, bit [8]**

SETEND instruction disable. Disables SETEND instructions at PL2.

`0` SETEND instruction execution is enabled at PL2.

`1` SETEND instructions are UNDEFINED at PL2.

If the implementation does not support mixed-endian operation at EL2, this bit is RES1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**ITD, bit [7]**

IT Disable. Disables some uses of IT instructions at PL2.

0 All IT instruction functionality is enabled at PL2.

1 Any attempt at PL2 to execute any of the following is UNDEFINED:

- All encodings of the IT instruction with hw1[3:0]!=1000.
- All encodings of the subsequent instruction with the following values for hw1:

  11xxxxxxxxxxxxx
  All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.

  1011xxxxxxxxxxx
  All instructions in *Miscellaneous 16-bit instructions* on page F3-2471.

  10100xxxxxxxxxx
  ADD Rd, PC, #imm

  01001xxxxxxxxxx
  LDR Rd, [PC, #imm]

  0100x1xxx1111xxx
  ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.

  010001xx1xxxx111
  ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn.

  These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

  It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

  - A 16-bit instruction, that can only be followed by another 16-bit instruction.
  - The first half of a 32-bit instruction.

  This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

  An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

ITD is optional, but if it is implemented in the SCTLR then it must also be implemented in the HSCTLR. If it is not implemented then this bit is RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.

**Bit [6]**

Reserved, RES0.

**CP15BEN, bit [5]**

CP15 memory barrier instruction enable. Enables accesses to the CP15 DMB, DSB, and ISB barrier operations from PL2.

0 PL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.

1 PL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR then it must also be implemented in the HSCTLR. If it is not implemented then this bit is RAO/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to a value that is architecturally UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [4:3]**

Reserved, RES1.

**C, bit [2]**

Cache enable, for data caching.

0 All data access to Normal memory from EL2, and all accesses to the EL2 translation tables, are Non-cacheable for all levels of data and unified cache.

1 All data access to Normal memory from EL2, and all accesses to the EL2 translation tables, can be cached at all levels of data and unified cache.

This bit has no effect on the PL1&0 translation regime.

When this register has an architecturally-defined reset value, this field resets to 0.

**A, bit [1]**

Alignment check enable. This is the enable bit for Alignment fault checking at EL2:

0 Alignment fault checking disabled when executing at EL2.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

1 Alignment fault checking enabled when executing at EL2.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**M, bit [0]**

MMU enable for EL2 stage 1 address translation. Possible values of this bit are:

0 EL2 stage 1 address translation disabled.

See the HSCTLR.I field for the behavior of instruction accesses to Normal memory.

1 EL2 stage 1 address translation enabled.

When this register has an architecturally-defined reset value, this field resets to 0.

## Accessing the HSCTLR:

To access the HSCTLR:

```
MRC p15,4,<Rt>,c1,c0,0 ; Read HSCTLR into Rt
MCR p15,4,<Rt>,c1,c0,0 ; Write Rt to HSCTLR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0001 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.67 HSR, Hyp Syndrome Register

The HSR characteristics are:

**Purpose**

Holds syndrome information for an exception taken to Hyp mode.

See also *Use of the HSR* on page G4-4105.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

Execution in any Non-secure PE mode other than Hyp mode makes this register UNKNOWN.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of HSR is UNKNOWN. The value written to HSR must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T5==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T5==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HSR is architecturally mapped to AArch64 System register ESR_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

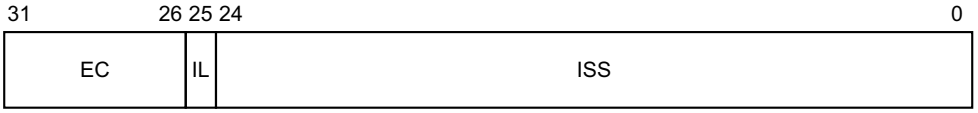
**Attributes**

HSR is a 32-bit register.

## Field descriptions

The HSR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**EC, bits [31:26]**

Exception Class. Indicates the reason for the exception that this register holds information about. Possible values of this field are:

**EC == 000000, Exceptions with an unknown reason**

Unknown reason.

See *ISS encoding for exceptions with an unknown reason* on page G6-4379.

**EC == 000001, Exception from a WFI or WFE instruction**

Trapped WFI or WFE instruction.

Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.

See *ISS encoding for an exception from a WFI or WFE instruction* on page G6-4379.

**EC == 000011, Exception from an MCR or MRC access**

Trapped MCR or MRC access to CP15.

See *ISS encoding for an exception from an MCR or MRC access* on page G6-4380.

**EC == 000100, Exception from an MCRR or MRRC access**

Trapped MCRR or MRRC access to CP15.

See *ISS encoding for an exception from an MCRR or MRRC access* on page G6-4382.

**EC == 000101, Exception from an MCR or MRC access**

Trapped MCR or MRC access to CP14.

See *ISS encoding for an exception from an MCR or MRC access* on page G6-4380.

**EC == 000110, Exception from an LDC or STC access to CP14**

Trapped LDC or STC access to CP14.

See *ISS encoding for an exception from an LDC or STC access to CP14* on page G6-4383.

**EC == 000111, Exception from an access to SIMD or floating-point functionality, resulting from HCPTR**

Exception from an access to SIMD or floating-point functionality, as a result of HCPTR.

Excludes exceptions resulting from when Advanced SIMD and floating-point are not implemented. These are reported with EC value `0b000000`.

See *ISS encoding for an exception from an access to SIMD or floating-point functionality, resulting from HCPTR* on page G6-4385.

**EC == 001000, Exception from an MCR or MRC access**

Trapped MRC or VMRS access to CP10, for ID group traps.

See *ISS encoding for an exception from an MCR or MRC access* on page G6-4380.

**EC == 001100, Exception from an MCRR or MRRC access**

Trapped MRRC or MCRR access to CP14.

See *ISS encoding for an exception from an MCRR or MRRC access* on page G6-4382.

**EC == 001110, Exception from an Illegal state or PC alignment fault**

Illegal state exception taken to AArch32 state.

See *ISS encoding for an exception from an Illegal state or PC alignment fault* on page G6-4390.

**EC == 010001, Exception from HVC or SVC instruction execution**

SVC taken to Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See *ISS encoding for an exception from HVC or SVC instruction execution* on page G6-4386.

**EC == 010010, Exception from HVC or SVC instruction execution**

HVC executed.

See *ISS encoding for an exception from HVC or SVC instruction execution* on page G6-4386.

**EC == 010011, Exception from SMC instruction execution**

Trapped SMC instruction.

See *ISS encoding for an exception from SMC instruction execution* on page G6-4387.

**EC == 100000, Exception from a Prefetch abort**

Prefetch Abort routed to Hyp mode.

See *ISS encoding for an exception from a Prefetch abort* on page G6-4388.

**EC == 100001, Exception from a Prefetch abort**

Prefetch Abort taken from Hyp mode.

See *ISS encoding for an exception from a Prefetch abort* on page G6-4388.

**EC == 100010, Exception from an Illegal state or PC alignment fault**

PC alignment fault exception.

See *ISS encoding for an exception from an Illegal state or PC alignment fault* on page G6-4390.

**EC == 100100, Exception from a Data abort**

Data Abort routed to Hyp mode.

See *ISS encoding for an exception from a Data abort* on page G6-4391.

**EC == 100101, Exception from a Data abort**

Data Abort taken from Hyp mode.

See *ISS encoding for an exception from a Data abort* on page G6-4391.

All other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

**IL, bit [25]**

Instruction length bit. Indicates the size of the instruction that has been trapped to Hyp mode. When this bit is valid, possible values of this bit are:

`0` 16-bit instruction trapped.

`1` 32-bit instruction trapped.

This field is RES1 and not valid for the following cases:

- When the EC field is `0b000000`, indicating an exception with an unknown reason.
- Prefetch Aborts.
- Data Aborts that do not have valid ISS information, or for which the ISS is not valid.
- When the EC value is `0b001110`, indicating an Illegal state exception.

**Note**

This is a change from the behavior in ARMv7, where the IL field is UNK/SBZP for the corresponding cases.

The IL field is not valid and is UNKNOWN on an exception from a PC alignment fault.

**ISS, bits [24:0]**

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

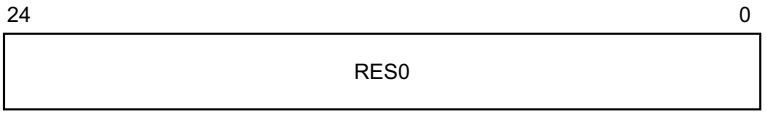
The following subsections describe each ISS format.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### ISS encoding for exceptions with an unknown reason

This encoding is used by Unknown reason.

The ISS encoding for these exceptions is:




**Bits [24:0]**

Reserved, RES0.

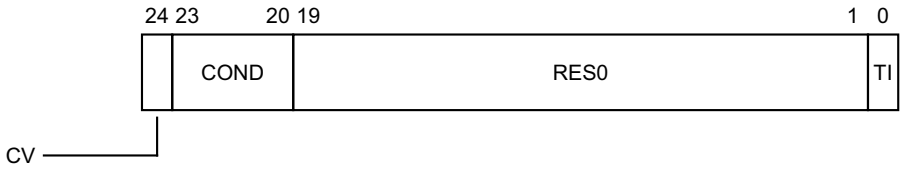
*Undefined Instruction exception, when the value of HCR.TGE is 1* on page G1-3791 describes the configuration settings for a trap that returns an HSR.EC value of `0b000000`.

### ISS encoding for an exception from a WFI or WFE instruction

This encoding is used by Trapped WFI or WFE instruction.

Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.

The ISS encoding for these exceptions is:




**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

**Bits [19:1]**

Reserved, RES0.

**TI, bit [0]**

Trapped instruction. Possible values of this bit are:

`0` WFI trapped.

`1` WFE trapped.

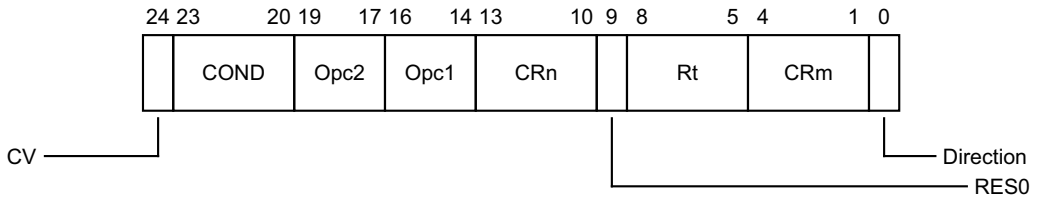
*Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page G1-3865 describes the configuration settings for this trap.

### ISS encoding for an exception from an MCR or MRC access

This encoding is used by:

- Trapped MCR or MRC access to CP15.
- Trapped MCR or MRC access to CP14.
- Trapped MRC or VMRS access to CP10, for ID group traps.

The ISS encoding for these exceptions is:




**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

**Opc2, bits [19:17]**

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value `0b000`.

**Opc1, bits [16:14]**

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value `0b111`.

**CRn, bits [13:10]**

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

**Bit [9]**

Reserved, RES0.

**Rt, bits [8:5]**

The Rt value from the issued instruction, the general-purpose register used for the transfer.

**CRm, bits [4:1]**

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value `0b0000`.

**Direction, bit [0]**

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0` Write to coprocessor. MCR instruction.

`1` Read from coprocessor. MRC or VMRS instruction.

The following sections describe configuration settings for traps that are reported using EC value `0b000011`:

- *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the ID registers* on page G1-3862.
- *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations* on page G1-3860.
- *Traps to Hyp mode of Non-secure EL1 execution of cache maintenance instructions* on page G1-3860.
- *Traps to Hyp mode of Non-secure EL1 execution of TLB maintenance instructions* on page G1-3859.
- *Traps to Hyp mode of Non-secure EL1 accesses to the Auxiliary Control Register* on page G1-3860.
- *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page G1-3872.
- *Traps to Hyp mode of Non-secure EL1 accesses to the CPACR* on page G1-3867.
- *Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers* on page G1-3858.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to CP15 System registers* on page G1-3868.

The following sections describe configuration settings for traps that are reported using EC value `0b000101`:

- *ID group 0, Primary device identification registers* on page G1-3863.
- *Traps to Hyp mode of Non-secure System register accesses to trace registers* on page G1-3867.
- *Trapping Non-secure System register accesses to Debug ROM registers* on page G1-3870.
- *Trapping Non-secure System register accesses to powerdown debug registers* on page G1-3870.
- *Trapping general Non-secure System register accesses to debug registers* on page G1-3871.

The following sections describes configuration settings for traps that are reported using EC value `0b001000`:

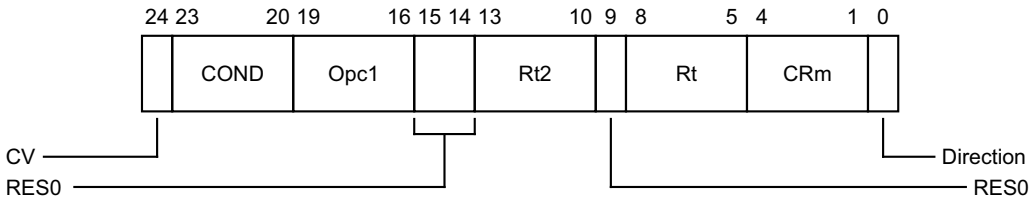
- *ID group 0, Primary device identification registers* on page G1-3863.
- *ID group 3, Detailed feature identification registers* on page G1-3864.

### *ISS encoding for an exception from an MCRR or MRRC access*

This encoding is used by:

- Trapped MCRR or MRRC access to CP15.
- Trapped MRRC or MCRR access to CP14.

The ISS encoding for these exceptions is:




**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

**Opc1, bits [19:16]**

The Opc1 value from the issued instruction.

**Bits [15:14]**

Reserved, RES0.

**Rt2, bits [13:10]**

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.

**Bit [9]**

Reserved, RES0.

**Rt, bits [8:5]**

The Rt value from the issued instruction, the first general-purpose register used for the transfer.

**CRm, bits [4:1]**

The CRm value from the issued instruction.

**Direction, bit [0]**

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0` Write to coprocessor. MCRR instruction.

`1` Read from coprocessor. MRRC instruction.

The following sections describe configuration settings for traps that are reported using EC value `0b000100`:

- *Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers* on page G1-3858.
- *General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to CP15 System registers* on page G1-3868.

The following sections describe configuration settings for traps that are reported using EC value `0b001100`:

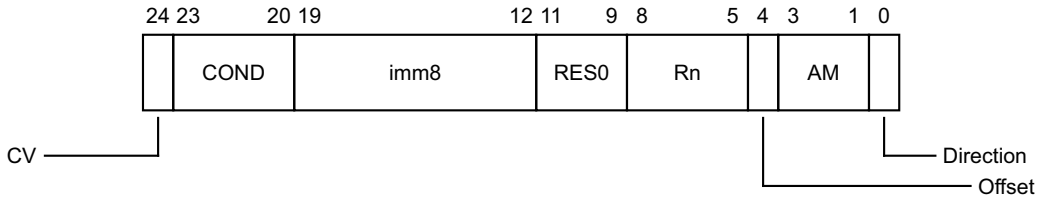
- *Traps to Hyp mode of Non-secure System register accesses to trace registers* on page G1-3867.
- *Trapping Non-secure System register accesses to Debug ROM registers* on page G1-3870.

### *ISS encoding for an exception from an LDC or STC access to CP14*

This encoding is used by Trapped LDC or STC access to CP14.

The ISS encoding for these exceptions is:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

**imm8, bits [19:12]**

The immediate value from the issued instruction.

**Bits [11:9]**

Reserved, RES0.

**Rn, bits [8:5]**

The Rn value from the issued instruction. Valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction.

When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

**Offset, bit [4]**

Indicates whether the offset is added or subtracted:

`0` Subtract offset.

`1` Add offset.

This bit corresponds to the U bit in the instruction encoding.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**AM, bits [3:1]**

Addressing mode. The permitted values of this field are:

`000` Immediate unindexed.

`001` Immediate post-indexed.

`010` Immediate offset.

`011` Immediate pre-indexed.

`100` Literal unindexed.
A32 instruction set only.
For a trapped LDC T32 instruction or STC T32 instruction, this encoding is reserved.

`110` Literal offset.
For the STC instruction, valid only in the A32 instruction set.
For a trapped STC T32 instruction, this encoding is reserved.

The values `0b101` and `0b111` are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

**Direction, bit [0]**

Indicates the direction of the trapped instruction. The possible values of this bit are:

`0` Write to memory. STC instruction.

`1` Read from memory. LDC instruction.

The only architected uses of these instructions to access CP14 are:

- An STC to write to DBGDTRRXint.
- An LDC to read DBGDTRTXint.

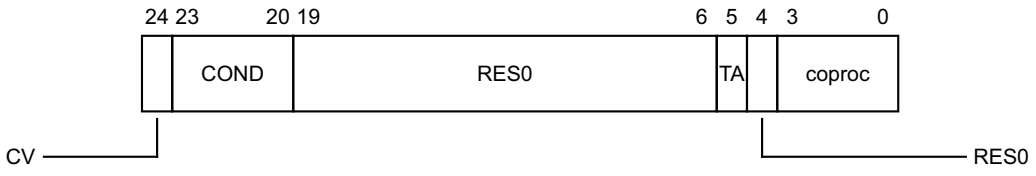
*Trapping general Non-secure System register accesses to debug registers* on page G1-3871 describes the configuration settings for the trap that is reported using EC value `0b000110`.

### *ISS encoding for an exception from an access to SIMD or floating-point functionality, resulting from HCPTR*

This encoding is used by Exception from an access to SIMD or floating-point functionality, as a result of HCPTR.

Excludes exceptions resulting from when Advanced SIMD and floating-point are not implemented. These are reported with EC value `0b000000`.

The ISS encoding for these exceptions is:




**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

**COND, bits [23:20]**

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

**Bits [19:6]**

Reserved, RES0.

**TA, bit [5]**

Indicates trapped use of Advanced SIMD functionality. The possible values of this bit are:

`0` Exception was not caused by trapped use of Advanced SIMD functionality.

`1` Exception was caused by trapped use of Advanced SIMD functionality.

Any use of an Advanced SIMD instruction that is not also a floating-point instruction that is trapped to Hyp mode because of a trap configured in the HCPTR sets this bit to 1.

For a list of these instructions, see *Controls of Advanced SIMD operation that do not apply to floating-point operation* on page E1-2332.

**Bit [4]**

Reserved, RES0.

**coproc, bits [3:0]**

The number of the coprocessor accessed by the trapped operation. This is always `0b1010`.

The following sections describe the configuration settings for the traps that are reported using EC value `0b000111`:

- *General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers* on page G1-3865.
- *Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality* on page G1-3866.

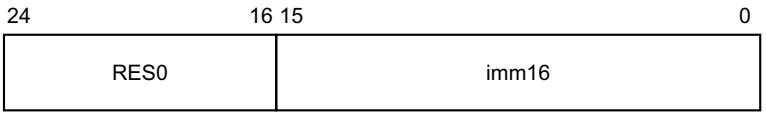
### *ISS encoding for an exception from HVC or SVC instruction execution*

This encoding is used by:

- SVC taken to Hyp mode.
- HVC executed.

The ISS encoding for these exceptions is:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [24:16]**

Reserved, RES0.

**imm16, bits [15:0]**

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, this is the value of the imm16 field of the issued instruction.

For an SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

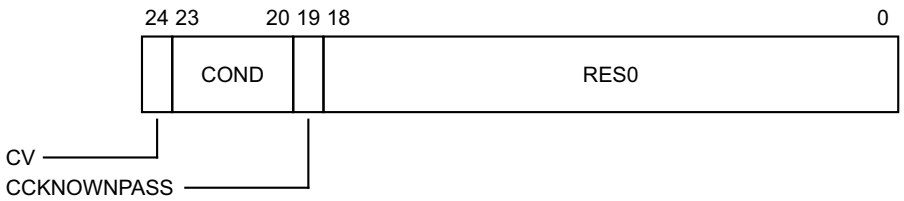
The HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

*Supervisor Call exception, when the value of HCR.TGE is 1* on page G1-3791 describes the configuration settings for the trap reported with EC value `0b010001`.

### *ISS encoding for an exception from SMC instruction execution*

This encoding is used by Trapped SMC instruction.

The ISS encoding for these exceptions is:




**CV, bit [24]**

Condition code valid. Possible values of this bit are:

`0` The COND field is not valid.

`1` The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

**COND, bits [23:20]**

The condition code for the trapped instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to `0b1110`.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to `0b1110`, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to `0b1110`, or to the value of any condition that applied to the instruction.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

**CCKNOWNPASS, bit [19]**

Indicates whether the instruction might have failed its condition code check.

`0` The instruction was unconditional, or was conditional and passed its condition code check.

`1` The instruction was conditional, and might have failed its condition code check.

**Bits [18:0]**

Reserved, RES0.

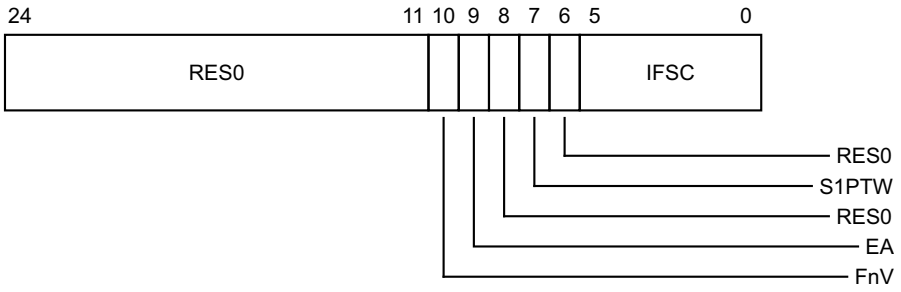
*Traps to Hyp mode of Non-secure EL1 execution of SMC instructions* on page G1-3861 describes the configuration settings for this trap, for instructions executed in Non-secure PL1 modes.

### *ISS encoding for an exception from a Prefetch abort*

This encoding is used by:

- Prefetch Abort routed to Hyp mode.
- Prefetch Abort taken from Hyp mode.

The ISS encoding for these exceptions is:




**Bits [24:11]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**FnV, bit [10]**

FAR not Valid, for a Synchronous external abort other than a Synchronous external abort on a translation table walk.

`0` HIFAR is valid.

`1` HIFAR is not valid, and holds an UNKNOWN value.

This field is only valid when the IFSC field is `0b010000`, that is it is valid only for a Synchronous external abort other than a Synchronous external abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

**EA, bit [9]**

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of external aborts.

For any abort other than an External abort this bit returns a value of 0.

**Bit [8]**

Reserved, RES0.

**S1PTW, bit [7]**

For a stage 2 fault, indicates whether the fault was a fault on the stage 2 translation of an address accessed during a stage 1 translation table walk:

`0` Fault not on a stage 2 translation for a stage 1 translation table walk.

`1` Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For a fault other than a stage 2 fault, this bit is RES0.

**Bit [6]**

Reserved, RES0.

**IFSC, bits [5:0]**

Instruction Fault Status Code. Possible values of this field are:

`000000` Address size fault, translation table base register

`000001` Address size fault, level 1

`000010` Address size fault, level 2

`000011` Address size fault, level 3

`000101` Translation fault, level 1

`000110` Translation fault, level 2

`000111` Translation fault, level 3

`001001` Access flag fault, level 1

`001010` Access flag fault, level 2

`001011` Access flag fault, level 3

`001101` Permission fault, level 1

`001110` Permission fault, level 2

`001111` Permission fault, level 3

`010000` Synchronous external abort, other than synchronous parity or ECC error, not on translation table walk

`011000` Synchronous parity or ECC error on memory access, not on translation table walk

`010101` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 1

`010110` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 2


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`010111` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 3

`011101` Synchronous parity or ECC error on memory access on translation table walk, level 1

`011110` Synchronous parity or ECC error on memory access on translation table walk, level 2

`011111` Synchronous parity or ECC error on memory access on translation table walk, level 3

`100001` Alignment fault

`100010` Debug event, only when the EC value is `0b100001`

`110000` TLB conflict abort

All other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

The following sections describe cases where Prefetch Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value `0b100000`:

- *Abort exceptions, when the value of HCR.TGE is 1* on page G1-3792.
- *Routing debug exceptions to EL2* on page G1-3795.

### ISS encoding for an exception from an Illegal state or PC alignment fault

This encoding is used by:

- Illegal state exception taken to AArch32 state.
- PC alignment fault exception.

The ISS encoding for these exceptions is:




**Bits [24:0]**

Reserved, RES0.

For more information about the Illegal state exception, see:

- *Illegal changes to PSTATE.M* on page G1-3771.
- *Illegal return events from AArch32 state* on page G1-3797.
- *Legal returns that set PSTATE.IL to 1* on page G1-3799.
- *The Illegal Execution state exception* on page G1-3799.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

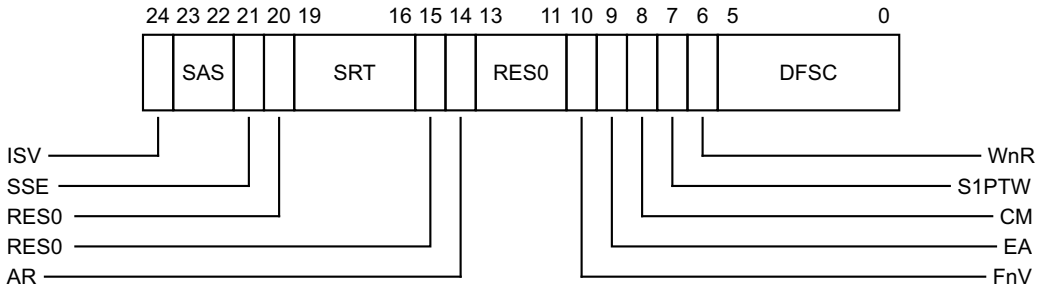
For more information about the PC alignment fault exception, see *Branching to an unaligned PC* on page K1-5424.

### *ISS encoding for an exception from a Data abort*

This encoding is used by:

- Data Abort routed to Hyp mode.
- Data Abort taken from Hyp mode.

The ISS encoding for these exceptions is:




**ISV, bit [24]**

Instruction syndrome valid. Indicates whether the syndrome information in ISS[23:14] is valid.

`0` No valid instruction syndrome. ISS[23:14] are RES0.

`1` ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults except those generated by a stage 2 translation. For Data Abort exceptions generated by a stage 2 translation, this bit is 1 and a valid instruction syndrome is returned, only if all of the following are true:

- The instruction that generated the Data Abort exception:
  - Is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
  - Is not performing register writeback.
  - Is not using the PC as a source or destination register.

A stage 2 abort on a stage 1 translation table lookup sets ISV to 0.

It is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a Synchronous external abort on stage 2 translation table walks.

In the A32 instruction set, LDR*T and STR*T instructions always perform register writeback and therefore never return a valid instruction syndrome.

**SAS, bits [23:22]**

Syndrome Access Size. When ISV is 1, indicates the size of the access attempted by the faulting operation.

`00` Byte

`01` Halfword

`10` Word

`11` Doubleword

This field is RES0 when the value of ISV is 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SSE, bit [21]**

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

`0` Sign-extension not required.

`1` Data item must be sign-extended.

For all other operations this bit is 0.

This field is RES0 when the value of ISV is 0.

**Bit [20]**

Reserved, RES0.

**SRT, bits [19:16]**

Syndrome Register transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction, as viewed in AArch64 state.

This field is RES0 when the value of ISV is 0.

**Bit [15]**

Reserved, RES0.

**AR, bit [14]**

Acquire/Release. When ISV is 1, the possible values of this bit are:

`0` Instruction did not have acquire/release semantics.

`1` Instruction did have acquire/release semantics.

This field is RES0 when the value of ISV is 0.

**Bits [13:11]**

Reserved, RES0.

**FnV, bit [10]**

FAR not Valid, for a Synchronous external abort other than a Synchronous external abort on a translation table walk.

`0` HDFAR is valid.

`1` HDFAR is not valid, and holds an UNKNOWN value.

This field is only valid when the DFSC field is `0b010000`, that is it is valid only for a Synchronous external abort other than a Synchronous external abort on a translation table walk. It is RES0 for all other Data Abort exceptions.

**EA, bit [9]**

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of external aborts.

For any abort other than an External abort this bit returns a value of 0.

**CM, bit [8]**

Cache maintenance. For a synchronous fault, identifies fault that comes from a cache maintenance or address translation instruction. For synchronous faults, the possible values of this bit are:

`0` Fault not generated by a cache maintenance or address translation instruction.

`1` Fault generated by a cache maintenance or address translation instruction.

For asynchronous faults, this bit is 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**S1PTW, bit [7]**

For a stage 2 fault, indicates whether the fault was a fault on the stage 2 translation of an address accessed during a stage 1 translation table walk:

0 Fault not on a stage 2 translation for a stage 1 translation table walk.

1 Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For a fault other than a stage 2 fault, this bit is RES0.

**WnR, bit [6]**

Write not Read. Indicates whether a synchronous abort was caused by a write instruction or a read instruction. The possible values of this bit are:

0 Abort caused by a read instruction.

1 Abort caused by a write instruction.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For an asynchronous Data Abort exception this bit is UNKNOWN.

**DFSC, bits [5:0]**

Data Fault Status Code. Possible values of this field are:

000000 Address size fault, translation table base register

000001 Address size fault, level 1

000010 Address size fault, level 2

000011 Address size fault, level 3

000101 Translation fault, level 1

000110 Translation fault, level 2

000111 Translation fault, level 3

001001 Access flag fault, level 1

001010 Access flag fault, level 2

001011 Access flag fault, level 3

001101 Permission fault, level 1

001110 Permission fault, level 2

001111 Permission fault, level 3

010000 Synchronous external abort, other than synchronous parity or ECC error, not on translation table walk

011000 Synchronous parity or ECC error on memory access, not on translation table walk

010001 Asynchronous external abort, other than asynchronous parity or ECC error

011001 Asynchronous parity or ECC error on memory access

010101 Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 1

010110 Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 2

010111 Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 3

011101 Synchronous parity or ECC error on memory access on translation table walk, level 1

011110 Synchronous parity or ECC error on memory access on translation table walk, level 2

011111 Synchronous parity or ECC error on memory access on translation table walk, level 3

100001 Alignment fault

100010 Debug event, only when the EC value is 0b100100


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`110000` TLB conflict abort

`110100` IMPLEMENTATION DEFINED fault (Lockdown fault)

`110101` IMPLEMENTATION DEFINED fault (Unsupported Exclusive access fault)

All other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

The following describe cases where Data Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value `0b100100`:

- *Abort exceptions, when the value of HCR.TGE is 1* on page G1-3792.
- *Routing debug exceptions to EL2* on page G1-3795.

The following describe cases that can cause a Data Abort exception that is taken to Hyp mode, and reported in the HSR with EC value of `0b100000` or `0b100100`:

- *Hyp mode control of Non-secure access permissions* on page G4-4044.
- *Memory fault reporting in Hyp mode* on page G4-4103.

### Accessing the HSR:

To access the HSR:

```
MRC p15,4,<Rt>,c5,c2,0 ; Read HSR into Rt
MCR p15,4,<Rt>,c5,c2,0 ; Write Rt to HSR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0101 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.68 HSTR, Hyp System Trap Register

The HSTR characteristics are:

**Purpose**

Controls trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to the CP15 primary coprocessor registers, {c0-c3,c5-c13,c15} using MCR, MRC, MCRR, and MRRC instructions.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HSTR is architecturally mapped to AArch64 System register HSTR_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 with EL2 using AArch32, or into EL3 with EL3 using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

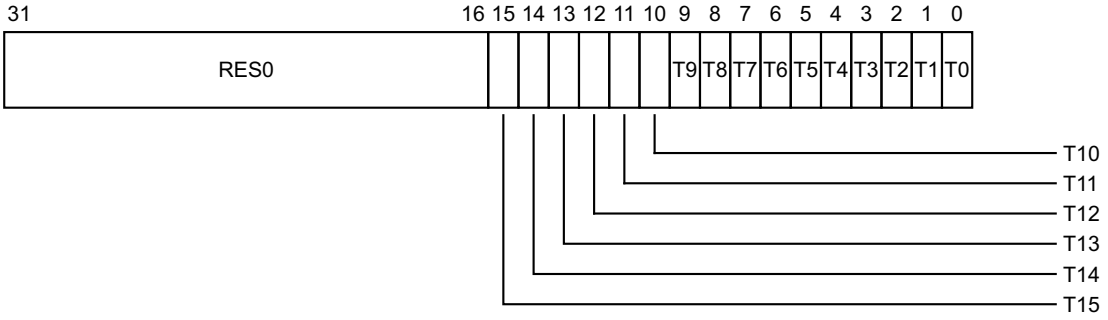
**Attributes**

HSTR is a 32-bit register.

### Field descriptions

The HSTR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:16]**

Reserved, RES0.

**T<n>, bit [n], for n = 0 to 15**

HSTR.{T0-T3, T5-T13, T15} trap Non-secure EL0 and EL1 accesses, using MCR, MRC, MCRR, and MRRC instructions, to the CP15 System registers, by the accessed primary CP15 register number, {c0-c3, c5-c13, c15}.

`0` This control has no effect on Non-secure EL0 or EL1 accesses to the CP15 System registers.

`1` Any Non-secure EL1 MCR, MRC, MCRR, or MRRC access to a corresponding register is trapped to Hyp mode.

A Non-secure EL0 MCR, MRC, MCRR, or MRRC access to the corresponding CP15 System register is trapped to Hyp mode only if it is not UNDEFINED when this bit is zero.

The primary CP15 register number is the value of <CRn> for an MCR or MRC instruction, or the value of <CRm> for an MCRR or MRRC instruction.

For example, when HSTR.T7 is 1:

- Any 32-bit CP15 access from a Non-secure EL1 mode, using an MCR or MRC instruction with <CRn> set to c7, is trapped to Hyp mode.
- Any 64-bit CP15 access from a Non-secure EL1 mode, using an MCRR or MRRC instruction with <CRm> set to c7, is trapped to Hyp mode.

Fields T4 and T14 are RES0.

When this register has an architecturally-defined reset value, this field resets to `0`.

## Accessing the HSTR:

To access the HSTR:

```
MRC p15,4,<Rt>,c1,c1,3 ; Read HSTR into Rt
MCR p15,4,<Rt>,c1,c1,3 ; Write Rt to HSTR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0001 | 0001 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.69 HTCR, Hyp Translation Control Register

The HTCR characteristics are:

**Purpose**

Controls translation table walks required for the stage 1 translation of memory accesses from Hyp mode, and holds cacheability and shareability information for the accesses.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

Used in conjunction with HTTBR, that defines the translation table base address for the translations.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T2==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T2==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HTCR is architecturally mapped to AArch64 System register TCR_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

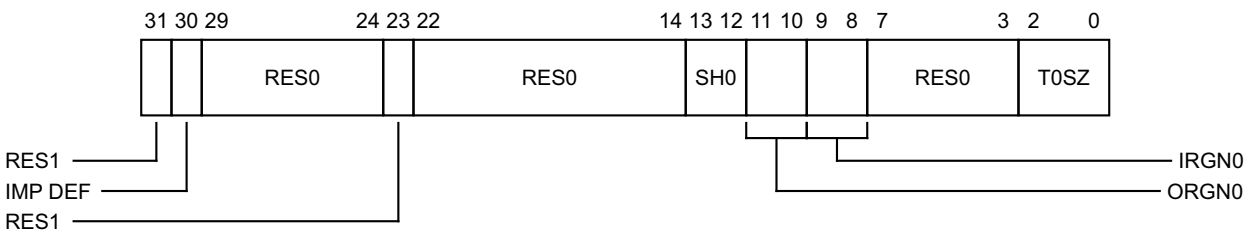
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HTCR is a 32-bit register.

#### Field descriptions

The HTCR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bit [31]**

Reserved, RES1.

**IMP DEF, bit [30]**

IMPLEMENTATION DEFINED.

**Bits [29:24]**

Reserved, RES0.

**Bit [23]**

Reserved, RES1.

**Bits [22:14]**

Reserved, RES0.

**SH0, bits [13:12]**

Shareability attribute for memory associated with translation table walks using HTTBR.

00 Non-shareable

10 Outer Shareable

11 Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

**ORGN0, bits [11:10]**

Outer cacheability attribute for memory associated with translation table walks using HTTBR.

00 Normal memory, Outer Non-cacheable

01 Normal memory, Outer Write-Back Write-Allocate Cacheable

10 Normal memory, Outer Write-Through Cacheable

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable

**IRGN0, bits [9:8]**

Inner cacheability attribute for memory associated with translation table walks using HTTBR.

00 Normal memory, Inner Non-cacheable

01 Normal memory, Inner Write-Back Write-Allocate Cacheable

10 Normal memory, Inner Write-Through Cacheable

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable

**Bits [7:3]**

Reserved, RES0.

**T0SZ, bits [2:0]**

The size offset of the memory region addressed by HTTBR. The region size is $2^{(32-T0SZ)}$ bytes.

## Accessing the HTCR:

To access the HTCR:

```
MRC p15,4,<Rt>,c2,c0,2 ; Read HTCR into Rt
MCR p15,4,<Rt>,c2,c0,2 ; Write Rt to HTCR
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0010 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.70 HTPIDR, Hyp Software Thread ID Register

The HTPIDR characteristics are:

**Purpose**

Provides a location where software running in Hyp mode can store thread identifying information that is not visible to Non-secure software executing at EL0 or EL1, for hypervisor management purposes.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

The PE never updates this register. This means the register is always UNKNOWN on reset.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T13==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T13==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HTPIDR is architecturally mapped to AArch64 System register TPIDR_EL2[31:0].

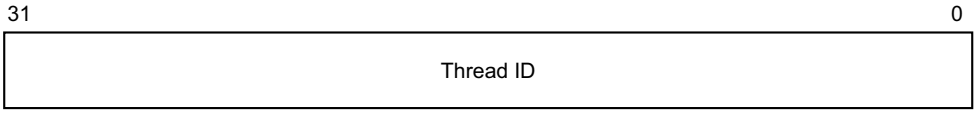
If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HTPIDR is a 32-bit register.

## Field descriptions

The HTPIDR bit assignments are:




**Bits [31:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the HTPIDR:

To access the HTPIDR:

```
MRC p15,4,<Rt>,c13,c0,2 ; Read HTPIDR into Rt
MCR p15,4,<Rt>,c13,c0,2 ; Write Rt to HTPIDR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1101 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.71 HTTBR, Hyp Translation Table Base Register

The HTTBR characteristics are:

**Purpose**

Holds the base address of the translation table for the stage 1 translation of memory accesses from Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

Used in conjunction with the HTCR.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T2==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T2==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HTTBR is architecturally mapped to AArch64 System register TTBR0_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

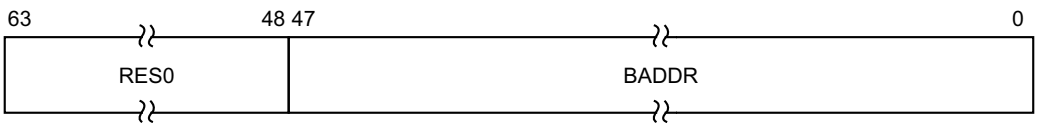
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HTTBR is a 64-bit register.

#### Field descriptions

The HTTBR bit assignments are:




**Bits [63:48]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**BADDR, bits [47:0]**

Translation table base address, bits[47:x], Bits [x-1:0] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of HTCR.T0SZ as follows:

- If HTCR.T0SZ is 0 or 1, x = 5 - HTCR.T0SZ.
- If HTCR.T0SZ is greater than 1, x = 14 - HTCR.T0SZ.

If bits[47:40] of the Translation table base address are not zero, an Address size fault is generated.

## Accessing the HTTBR:

To access the HTTBR:

```
MRRC p15,4,<Rt>,<Rt2>,c2 ; Read HTTBR[31:0] into Rt and HTTBR[63:32] into Rt2
MCRR p15,4,<Rt>,<Rt2>,c2 ; Write Rt to HTTBR[31:0] and Rt2 to HTTBR[63:32]
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0100 | 0010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.72 HVBAR, Hyp Vector Base Address Register

The HVBAR characteristics are:

**Purpose**

Holds the vector base address for any exception that is taken to Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T12==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T12==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register HVBAR is architecturally mapped to AArch64 System register VBAR_EL2[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

HVBAR is a 32-bit register.

### Field descriptions

The HVBAR bit assignments are:




**Bits [31:5]**

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

**Bits [4:0]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the HVBAR:

To access the HVBAR:

```
MRC p15,4,<Rt>,c12,c0,0 ; Read HVBAR into Rt
MCR p15,4,<Rt>,c12,c0,0 ; Write Rt to HVBAR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1100 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.73 ICIALLU, Instruction Cache Invalidate All to PoU

The ICIALLU characteristics are:

**Purpose**

Invalidate all instruction caches to PoU. If branch predictors are architecturally visible, also flush branch predictors.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TPU==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TPU==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

AArch32 System operation ICIALLU performs the same function as AArch64 System operation IC IALLU.

**Attributes**

ICIALLU is a 32-bit system operation.

#### Field descriptions

The ICIALLU operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the ICIALLU operation:

To perform the ICIALLU operation:

```
MCR p15,0,<Rt>,c7,c5,0 ; ICIALLU operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.74 ICIALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The ICIALLUIS characteristics are:

**Purpose**

Invalidate all instruction caches Inner Shareable to PoU. If branch predictors are architecturally visible, also flush branch predictors.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TPU==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TPU==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

AArch32 System operation ICIALLUIS performs the same function as AArch64 System operation IC IALLUIS.

**Attributes**

ICIALLUIS is a 32-bit system operation.

#### Field descriptions

The ICIALLUIS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the ICIALLUIS operation:

To perform the ICIALLUIS operation:

```
MCR p15,0,<Rt>,c7,c1,0 ; ICIALLUIS operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.75 ICIMVAU, Instruction Cache line Invalidate by VA to PoU

The ICIMVAU characteristics are:

**Purpose**

Invalidate instruction cache line by virtual address to PoU.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TPU==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TPU==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T7==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

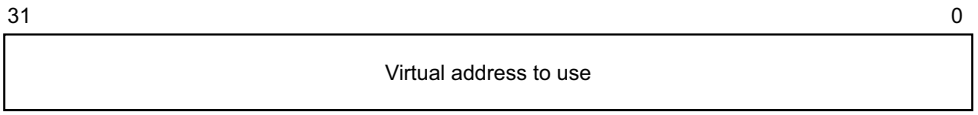
AArch32 System operation ICIMVAU performs the same function as AArch64 System operation IC IVAU.

**Attributes**

ICIMVAU is a 32-bit system operation.

## Field descriptions

The ICIMVAU input value bit assignments are:




**Bits [31:0]**

Virtual address to use.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Performing the ICIMVAU operation:

To perform the ICIMVAU operation:

```
MCR p15,0,<Rt>,c7,c5,1 ; ICIMVAU operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 0101 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.76 ID_AFR0, Auxiliary Feature Register 0

The ID_AFR0 characteristics are:

**Purpose**

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with the Main ID Register, MIDR.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register ID_AFR0 is architecturally mapped to AArch64 System register ID_AFR0_EL1.

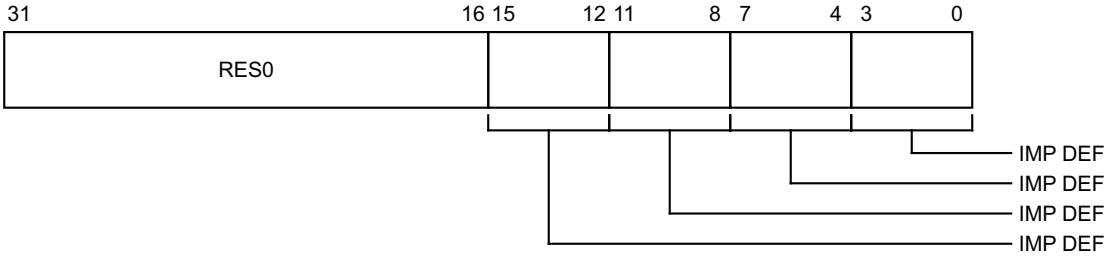
**Attributes**

ID_AFR0 is a 32-bit register.

#### Field descriptions

The ID_AFR0 bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:16]**

Reserved, RES0.

**IMPLEMENTATION DEFINED, bits [15:12]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [11:8]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [7:4]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [3:0]**

IMPLEMENTATION DEFINED.

### Accessing the ID_AFR0:

To access the ID_AFR0:

```
MRC p15,0,<Rt>,c0,c1,3 ; Read ID_AFR0 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0001 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.77 ID_DFR0, Debug Feature Register 0

The ID_DFR0 characteristics are:

**Purpose**

Provides top level information about the debug system in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with the Main ID Register, MIDR.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

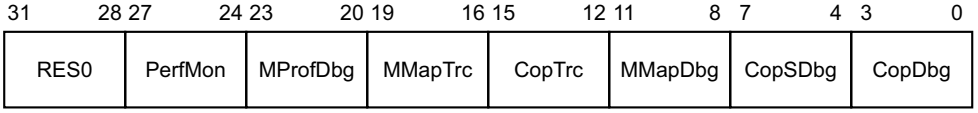
AArch32 System register ID_DFR0 is architecturally mapped to AArch64 System register ID_DFR0_EL1.

**Attributes**

ID_DFR0 is a 32-bit register.

#### Field descriptions

The ID_DFR0 bit assignments are:




**Bits [31:28]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**PerfMon, bits [27:24]**

Performance Monitors. Support for coprocessor-based ARM Performance Monitors Extension, for A and R profile processors. Defined values are:

`0000` Performance Monitors Extension System registers not implemented.

`0001` Support for Performance Monitors Extension version 1 (PMUv1) System registers.

`0010` Support for Performance Monitors Extension version 2 (PMUv2) System registers.

`0011` Support for Performance Monitors Extension version 3 (PMUv3) System registers.

`1111` IMPLEMENTATION DEFINED form of Performance Monitors System registers supported. PMUv3 not supported.

All other values are reserved.

In ARMv8-A the permitted values are `0000`, `0011`, and `1111`.

In ARMv7, the value `0000` can mean that PMUv1 is implemented. PMUv1 is not permitted in an ARMv8 implementation.

**MProfDbg, bits [23:20]**

M Profile Debug. Support for memory-mapped debug model for M profile processors. Defined values are:

`0000` Not supported.

`0001` Support for M profile Debug architecture, with memory-mapped access.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**MMapTrc, bits [19:16]**

Memory Mapped Trace. Support for memory-mapped trace model. Defined values are:

`0000` Not supported.

`0001` Support for ARM trace architecture, with memory-mapped access.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and `0001`.

In the Trace registers, the ETMIDR gives more information about the implementation.

**CopTrc, bits [15:12]**

Coprocessor Trace. Support for coprocessor-based trace model. Defined values are:

`0000` Not supported.

`0001` Support for ARM trace architecture, with CP14 access.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and `0001`.

In the Trace registers, the ETMIDR gives more information about the implementation.

**MMapDbg, bits [11:8]**

Memory Mapped Debug. Support for v7 memory-mapped debug model, for A and R profile processors.

In ARMv8-A this field is RES0.

The optional memory map defined by ARMv8 is not compatible with ARMv7.

**CopSDbg, bits [7:4]**

Coprocessor Secure Debug. Support for a Secure debug model accessed through a conceptual coprocessor, for an A profile processor that includes EL3.

If EL3 is not implemented and the implemented Security state is Non-Secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**CopDbg, bits [3:0]**

Coprocessor Debug. Support for coprocessor based debug model, for A and R profile processors. Defined values are:

`0000` Not supported.

`0010` Support for ARMv6, v6 Debug architecture, with CP14 access.

`0011` Support for ARMv6, v6.1 Debug architecture, with CP14 access.

`0100` Support for ARMv7, v7 Debug architecture, with CP14 access.

`0101` Support for ARMv7, v7.1 Debug architecture, with CP14 access.

`0110` Support for ARMv8 debug architecture, with CP14 access.

All other values are reserved.

In ARMv8-A the permitted values are `0000`, and `0110`.

## Accessing the ID_DFR0:

To access the ID_DFR0:

```
MRC p15,0,<Rt>,c0,c1,2 ; Read ID_DFR0 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0001 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.78 ID_ISAR0, Instruction Set Attribute Register 0

The ID_ISAR0 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

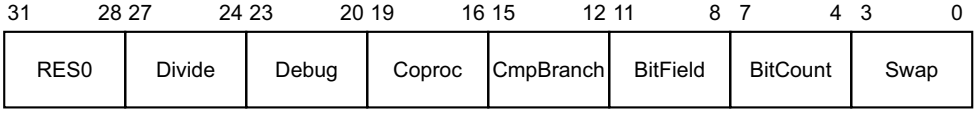
AArch32 System register ID_ISAR0 is architecturally mapped to AArch64 System register ID_ISAR0_EL1.

**Attributes**

ID_ISAR0 is a 32-bit register.

### Field descriptions

The ID_ISAR0 bit assignments are:




**Bits [31:28]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Divide, bits [27:24]**

Indicates the implemented Divide instructions. Defined values are:

0000 None implemented.

0001 Adds SDIV and UDIV in the T32 instruction set.

0010 As for 0001, and adds SDIV and UDIV in the A32 instruction set.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

**Debug, bits [23:20]**

Indicates the implemented Debug instructions. Defined values are:

0000 None implemented.

0001 Adds BKPT.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**Coproc, bits [19:16]**

Indicates the implemented Coprocessor instructions. Defined values are:

0000 None implemented, except for instructions separately attributed by the architecture, including CP15, CP14, and Advanced SIMD and VFP.

0001 Adds generic CDP, LDC, MCR, MRC, and STC.

0010 As for 0001, and adds generic CDP2, LDC2, MCR2, MRC2, and STC2.

0011 As for 0010, and adds generic MCRR and MRRC.

0100 As for 0011, and adds generic MCRR2 and MRRC2.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**CmpBranch, bits [15:12]**

Indicates the implemented combined Compare and Branch instructions in the T32 instruction set. Defined values are:

0000 None implemented.

0001 Adds CBNZ and CBZ.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**BitField, bits [11:8]**

Indicates the implemented BitField instructions. Defined values are:

0000 None implemented.

0001 Adds BFC, BFI, SBFX, and UBFX.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**BitCount, bits [7:4]**

Indicates the implemented Bit Counting instructions. Defined values are:

0000 None implemented.

0001 Adds CLZ.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Swap, bits [3:0]**

Indicates the implemented Swap instructions in the A32 instruction set. Defined values are:

`0000` None implemented.

`0001` Adds SWP and SWPB.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

### Accessing the ID_ISAR0:

To access the ID_ISAR0:

```
MRC p15,0,<Rt>,c0,c2,0 ; Read ID_ISAR0 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.79 ID_ISAR1, Instruction Set Attribute Register 1

The ID_ISAR1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_ISAR0, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

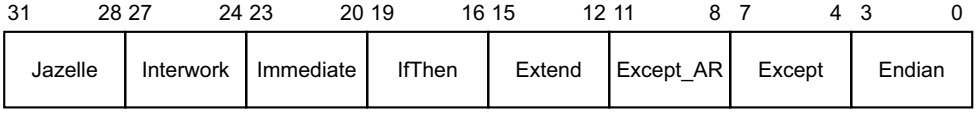
AArch32 System register ID_ISAR1 is architecturally mapped to AArch64 System register ID_ISAR1_EL1.

**Attributes**

ID_ISAR1 is a 32-bit register.

#### Field descriptions

The ID_ISAR1 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Jazelle, bits [31:28]**

Indicates the implemented Jazelle extension instructions. Defined values are:

`0000` No support for Jazelle.

`0001` Adds the BXJ instruction, and the J bit in the PSR. This setting might indicate a trivial implementation of the Jazelle extension.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Interwork, bits [27:24]**

Indicates the implemented Interworking instructions. Defined values are:

`0000` None implemented.

`0001` Adds the BX instruction, and the T bit in the PSR.

`0010` As for `0001`, and adds the BLX instruction. PC loads have BX-like behavior.

`0011` As for `0010`, and guarantees that data-processing instructions in the A32 instruction set with the PC as the destination and the S bit clear have BX-like behavior.

All other values are reserved.

In ARMv8-A the only permitted value is `0011`.

**Immediate, bits [23:20]**

Indicates the implemented data-processing instructions with long immediates. Defined values are:

`0000` None implemented.

`0001` Adds:

- The MOVT instruction.
- The MOV instruction encodings with zero-extended 16-bit immediates.
- The T32 ADD and SUB instruction encodings with zero-extended 12-bit immediates, and the other ADD, ADR, and SUB encodings cross-referenced by the pseudocode for those encodings.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**IfThen, bits [19:16]**

Indicates the implemented If-Then instructions in the T32 instruction set. Defined values are:

`0000` None implemented.

`0001` Adds the IT instructions, and the IT bits in the PSRs.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Extend, bits [15:12]**

Indicates the implemented Extend instructions. Defined values are:

`0000` No scalar sign-extend or zero-extend instructions are implemented, where scalar instructions means non-Advanced SIMD instructions.

`0001` Adds the SXTB, SXTH, UXTB, and UXTH instructions.

`0010` As for `0001`, and adds the SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Except_AR, bits [11:8]**

Indicates the implemented A and R profile exception-handling instructions. Defined values are:

`0000` None implemented.

`0001` Adds the SRS and RFE instructions, and the A and R profile forms of the CPS instruction.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Except, bits [7:4]**

Indicates the implemented exception-handling instructions in the ARM instruction set. Defined values are:

`0000` Not implemented. This indicates that the User bank and Exception return forms of the LDM and STM instructions are not implemented.

`0001` Adds the LDM (exception return), LDM (user registers), and STM (user registers) instruction versions.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Endian, bits [3:0]**

Indicates the implemented Endian instructions. Defined values are:

`0000` None implemented.

`0001` Adds the SETEND instruction, and the E bit in the PSRs.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and `0001`.

## Accessing the ID_ISAR1:

To access the ID_ISAR1:

```
MRC p15,0,<Rt>,c0,c2,1 ; Read ID_ISAR1 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.80 ID_ISAR2, Instruction Set Attribute Register 2

The ID_ISAR2 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

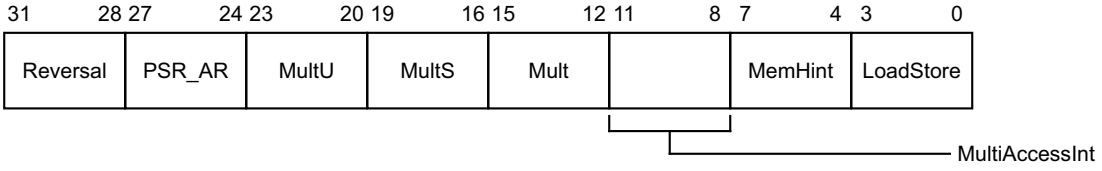
AArch32 System register ID_ISAR2 is architecturally mapped to AArch64 System register ID_ISAR2_EL1.

**Attributes**

ID_ISAR2 is a 32-bit register.

### Field descriptions

The ID_ISAR2 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Reversal, bits [31:28]**

Indicates the implemented Reversal instructions. Defined values are:

`0000` None implemented.

`0001` Adds the REV, REV16, and REVSH instructions.

`0010` As for `0001`, and adds the RBIT instruction.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

**PSR_AR, bits [27:24]**

Indicates the implemented A and R profile instructions to manipulate the PSR. Defined values are:

`0000` None implemented.

`0001` Adds the MRS and MSR instructions, and the exception return forms of data-processing instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

The exception return forms of the data-processing instructions are:

- In the A32 instruction set, data-processing instructions with the PC as the destination and the S bit set. These instructions might be affected by the WithShifts attribute.
- In the T32 instruction set, the SUBS PC,LR,#N instruction.

**MultU, bits [23:20]**

Indicates the implemented advanced unsigned Multiply instructions. Defined values are:

`0000` None implemented.

`0001` Adds the UMULL and UMLAL instructions.

`0010` As for `0001`, and adds the UMAAL instruction.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

**MultS, bits [19:16]**

Indicates the implemented advanced signed Multiply instructions. Defined values are:

`0000` None implemented.

`0001` Adds the SMULL and SMLAL instructions.

`0010` As for `0001`, and adds the SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, and SMULWT instructions. Also adds the Q bit in the PSRs.

`0011` As for `0010`, and adds the SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSDX, SMLSLD, SMLSLDX, SMMLA, SMMLAR, SMMLS, SMMLSR, SMMUL, SMMULR, SMUAD, SMUADX, SMUSD, and SMUSDX instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0011`.

**Mult, bits [15:12]**

Indicates the implemented additional Multiply instructions. Defined values are:

`0000` No additional instructions implemented. This means only MUL is implemented.

`0001` Adds the MLA instruction.

`0010` As for `0001`, and adds the MLS instruction.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**MultiAccessInt, bits [11:8]**

Indicates the support for interruptible multi-access instructions. Defined values are:

`0000` No support. This means the LDM and STM instructions are not interruptible.

`0001` LDM and STM instructions are restartable.

`0010` LDM and STM instructions are continuable.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**MemHint, bits [7:4]**

Indicates the implemented Memory Hint instructions. Defined values are:

`0000` None implemented.

`0001` Adds the PLD instruction.

`0010` Adds the PLD instruction. (`0001` and `0010` have identical effects.)

`0011` As for `0001` (or `0010`), and adds the PLI instruction.

`0100` As for `0011`, and adds the PLDW instruction.

All other values are reserved.

In ARMv8-A the only permitted value is `0100`.

**LoadStore, bits [3:0]**

Indicates the implemented additional load/store instructions. Defined values are:

`0000` No additional load/store instructions implemented.

`0001` Adds the LDRD and STRD instructions.

`0010` As for `0001`, and adds the Load Acquire (LDAB, LDAH, LDA, LDAEXB, LDAEXH, LDAEX, LDAEXD) and Store Release (STLB, STLH, STL, STLEXB, STLEXH, STLEX, STLEXD) instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

### Accessing the ID_ISAR2:

To access the ID_ISAR2:

```
MRC p15,0,<Rt>,c0,c2,2 ; Read ID_ISAR2 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0010 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.81 ID_ISAR3, Instruction Set Attribute Register 3

The ID_ISAR3 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR4, and ID_ISAR5.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

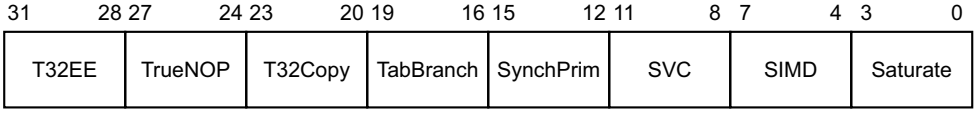
AArch32 System register ID_ISAR3 is architecturally mapped to AArch64 System register ID_ISAR3_EL1.

**Attributes**

ID_ISAR3 is a 32-bit register.

#### Field descriptions

The ID_ISAR3 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**T32EE, bits [31:28]**

Indicates the implemented T32EE instructions. Defined values are:

`0000` None implemented.

`0001` Adds the ENTERX and LEAVEX instructions, and modifies the load behavior to include null checking.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**TrueNOP, bits [27:24]**

Indicates the implemented true NOP instructions. Defined values are:

`0000` None implemented. This means there are no NOP instructions that do not have any register dependencies.

`0001` Adds true NOP instructions in both the T32 and A32 instruction sets. This also permits additional NOP-compatible hints.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**T32Copy, bits [23:20]**

Indicates the support for T32 non flag-setting MOV instructions. Defined values are:

`0000` Not supported. This means that in the T32 instruction set, encoding T1 of the MOV (register) instruction does not support a copy from a low register to a low register.

`0001` Adds support for T32 instruction set encoding T1 of the MOV (register) instruction, copying from a low register to a low register.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**TabBranch, bits [19:16]**

Indicates the implemented Table Branch instructions in the T32 instruction set. Defined values are:

`0000` None implemented.

`0001` Adds the TBB and TBH instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**SynchPrim, bits [15:12]**

Used in conjunction with ID_ISAR4.SynchPrim_frac to indicate the implemented Synchronization Primitive instructions. Defined values are:

`0000` If SynchPrim_frac == `0000`, no Synchronization Primitives implemented.

`0001` If SynchPrim_frac == `0000`, adds the LDREX and STREX instructions.
If SynchPrim_frac == `0011`, also adds the CLREX, LDREXB, STREXB, and STREXH instructions.

`0010` If SynchPrim_frac == `0000`, as for [`0001`, `0011`] and also adds the LDREXD and STREXD instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In ARMv8-A the only permitted value is `0010`.

**SVC, bits [11:8]**

Indicates the implemented SVC instructions. Defined values are:

`0000` Not implemented.

`0001` Adds the SVC instruction.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In ARMv8-A the only permitted value is 0001.

**SIMD, bits [7:4]**

Indicates the implemented SIMD instructions. Defined values are:

0000 None implemented.

0001 Adds the SSAT and USAT instructions, and the Q bit in the PSRs.

0011 As for 0001, and adds the PKHBT, PKHTB, QADD16, QADD8, QASX, QSUB16, QSUB8, QSAX, SADD16, SADD8, SASX, SEL, SHADD16, SHADD8, SHASX, SHSUB16, SHSUB8, SHSAX, SSAT16, SSUB16, SSUB8, SSAX, SXTAB16, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSUB16, UHSUB8, UHSAX, UQADD16, UQADD8, UQASX, UQSUB16, UQSUB8, UQSAX, USAD8, USADA8, USAT16, USUB16, USUB8, USAX, UXTAB16, and UXTB16 instructions. Also adds support for the GE[3:0] bits in the PSRs.

All other values are reserved.

In ARMv8-A the only permitted value is 0011.

The SIMD field relates only to implemented instructions that perform SIMD operations on the general-purpose registers. In an implementation that supports floating-point and Advanced SIMD instructions, MVFR0, MVFR1, and MVFR2 give information about the implemented Advanced SIMD instructions.

**Saturate, bits [3:0]**

Indicates the implemented Saturate instructions. Defined values are:

0000 None implemented. This means no non-Advanced SIMD saturate instructions are implemented.

0001 Adds the QADD, QDADD, QDSUB, and QSUB instructions, and the Q bit in the PSRs.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

## Accessing the ID_ISAR3:

To access the ID_ISAR3:

```
MRC p15,0,<Rt>,c0,c2,3 ; Read ID_ISAR3 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0010 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.82 ID_ISAR4, Instruction Set Attribute Register 4

The ID_ISAR4 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, and ID_ISAR5.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

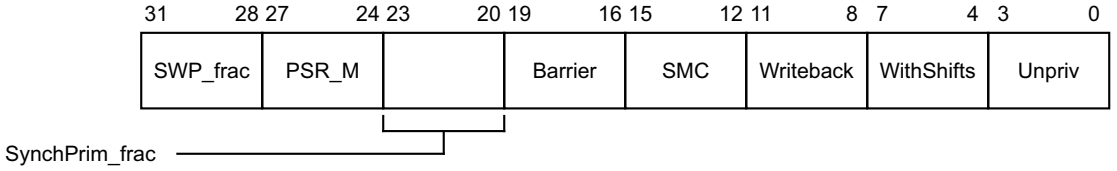
AArch32 System register ID_ISAR4 is architecturally mapped to AArch64 System register ID_ISAR4_EL1.

**Attributes**

ID_ISAR4 is a 32-bit register.

### Field descriptions

The ID_ISAR4 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SWP_frac, bits [31:28]**

Indicates support for the memory system locking the bus for SWP or SWPB instructions. Defined values are:

`0000` SWP or SWPB instructions not implemented.

`0001` SWP or SWPB implemented but only in a uniprocessor context. SWP and SWPB do not guarantee whether memory accesses from other masters can come between the load memory access and the store memory access of the SWP or SWPB.

All other values are reserved. This field is valid only if the ID_ISAR0.Swap_instrs field is `0000`.

In ARMv8-A the only permitted value is `0000`.

**PSR_M, bits [27:24]**

Indicates the implemented M profile instructions to modify the PSRs. Defined values are:

`0000` None implemented.

`0001` Adds the M profile forms of the CPS, MRS, and MSR instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**SynchPrim_frac, bits [23:20]**

Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented Synchronization Primitive instructions. Possible values are:

`0000` If SynchPrim == `0000`, no Synchronization Primitives implemented. If SynchPrim == `0001`, adds the LDREX and STREX instructions. If SynchPrim == `0010`, also adds the CLREX, LDREXB, LDREXH, STREXB, STREXH, LDREXD, and STREXD instructions.

`0011` If SynchPrim == `0001`, adds the LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In ARMv8-A the only permitted value is `0000`.

**Barrier, bits [19:16]**

Indicates the implemented Barrier instructions in the A32 and T32 instruction sets. Defined values are:

`0000` None implemented. Barrier operations are provided only as CP15 operations.

`0001` Adds the DMB, DSB, and ISB barrier instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**SMC, bits [15:12]**

Indicates the implemented SMC instructions. Defined values are:

`0000` None implemented.

`0001` Adds the SMC instruction.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Writeback, bits [11:8]**

Indicates the support for Writeback addressing modes. Defined values are:

`0000` Basic support. Only the LDM, STM, PUSH, POP, SRS, and RFE instructions support writeback addressing modes. These instructions support all of their writeback addressing modes.

`0001` Adds support for all of the writeback addressing modes.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**WithShifts, bits [7:4]**

Indicates the support for instructions with shifts. Defined values are:

0000 Nonzero shifts supported only in MOV and shift instructions.

0001 Adds support for shifts of loads and stores over the range LSL 0-3.

0011 As for 0001, and adds support for other constant shift options, both on load/store and other instructions.

0100 As for 0011, and adds support for register-controlled shift options.

All other values are reserved.

In ARMv8-A the only permitted value is 0100.

**Unpriv, bits [3:0]**

Indicates the implemented unprivileged instructions. Defined values are:

0000 None implemented. No T variant instructions are implemented.

0001 Adds the LDRBT, LDRT, STRBT, and STRT instructions.

0010 As for 0001, and adds the LDRHT, LDRSBT, LDRSHT, and STRHT instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

### Accessing the ID_ISAR4:

To access the ID_ISAR4:

```
MRC p15,0,<Rt>,c0,c2,4 ; Read ID_ISAR4 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0010 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.83 ID_ISAR5, Instruction Set Attribute Register 5

The ID_ISAR5 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, and ID_ISAR4.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

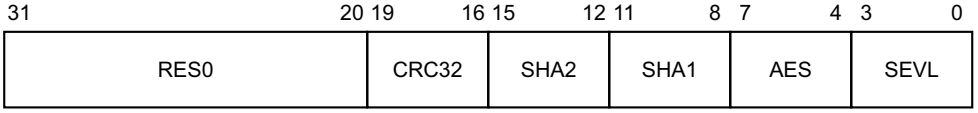
AArch32 System register ID_ISAR5 is architecturally mapped to AArch64 System register ID_ISAR5_EL1.

**Attributes**

ID_ISAR5 is a 32-bit register.

#### Field descriptions

The ID_ISAR5 bit assignments are:




**Bits [31:20]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**CRC32, bits [19:16]**

Indicates whether CRC32 instructions are implemented in AArch32.

0000 No CRC32 instructions implemented.

0001 CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW instructions implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

**SHA2, bits [15:12]**

Indicates whether SHA2 instructions are implemented in AArch32.

0000 No SHA2 instructions implemented.

0001 SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

**SHA1, bits [11:8]**

Indicates whether SHA1 instructions are implemented in AArch32.

0000 No SHA1 instructions implemented.

0001 SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

**AES, bits [7:4]**

Indicates whether AES instructions are implemented in AArch32.

0000 No AES instructions implemented.

0001 AESE, AESD, AESMC, and AESIMC implemented.

0010 As for 0001, plus PMULL/PMULL2 instructions operating on 64-bit data quantities.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0010.

**SEVL, bits [3:0]**

Indicates whether the SEVL instruction is implemented in AArch32.

0000 SEVL is implemented as a NOP.

0001 SEVL is implemented as Send Event Local.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

### Accessing the ID_ISAR5:

To access the ID_ISAR5:

```
MRC p15,0,<Rt>,c0,c2,5 ; Read ID_ISAR5 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0010 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.84 ID_MMFR0, Memory Model Feature Register 0

The ID_MMFR0 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_MMFR1, ID_MMFR2, and ID_MMFR3.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

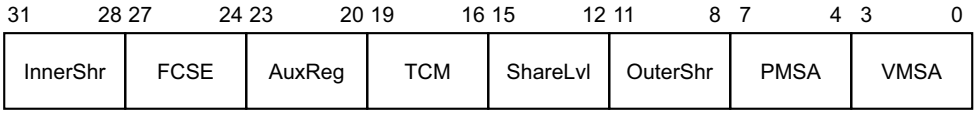
AArch32 System register ID_MMFR0 is architecturally mapped to AArch64 System register ID_MMFR0_EL1.

**Attributes**

ID_MMFR0 is a 32-bit register.

#### Field descriptions

The ID_MMFR0 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**InnerShr, bits [31:28]**

Innermost Shareability. Indicates the innermost shareability domain implemented. Defined values are:

`0000` Implemented as Non-cacheable.

`0001` Implemented with hardware coherency support.

`1111` Shareability ignored.

All other values are reserved.

In ARMv8-A the permitted values are `0000`, `0001`, and `1111`.

This field is valid only if the implementation distinguishes between Inner Shareable and Outer Shareable, by implementing two levels of shareability, as indicated by the value of the Shareability levels field, bits[15:12].

When the Shareability level field is zero, this field is UNK.

**FCSE, bits [27:24]**

Indicates whether the implementation includes the FCSE. Defined values are:

`0000` Not supported.

`0001` Support for FCSE.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**AuxReg, bits [23:20]**

Auxiliary Registers. Indicates support for Auxiliary registers. Defined values are:

`0000` None supported.

`0001` Support for Auxiliary Control Register only.

`0010` Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

**Note**

Accesses to unimplemented Auxiliary registers are UNDEFINED.

**TCM, bits [19:16]**

Indicates support for TCMs and associated DMAs. Defined values are:

`0000` Not supported.

`0001` Support is IMPLEMENTATION DEFINED. ARMv7 requires this setting.

`0010` Support for TCM only, ARMv6 implementation.

`0011` Support for TCM and DMA, ARMv6 implementation.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**ShareLvl, bits [15:12]**

Shareability Levels. Indicates the number of shareability levels implemented. Defined values are:

`0000` One level of shareability implemented.

`0001` Two levels of shareability implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**OuterShr, bits [11:8]**

Outermost Shareability. Indicates the outermost shareability domain implemented. Defined values are:

`0000` Implemented as Non-cacheable.

`0001` Implemented with hardware coherency support.

`1111` Shareability ignored.

All other values are reserved.

In ARMv8-A the permitted values are `0000`, `0001`, and `1111`.

**PMSA, bits [7:4]**

Indicates support for a PMSA. Defined values are:

`0000` Not supported.

`0001` Support for IMPLEMENTATION DEFINED PMSA.

`0010` Support for PMSAv6, with a Cache Type Register implemented.

`0011` Support for PMSAv7, with support for memory subsections. ARMv7-R profile.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**VMSA, bits [3:0]**

Indicates support for a VMSA. Defined values are:

`0000` Not supported.

`0001` Support for IMPLEMENTATION DEFINED VMSA.

`0010` Support for VMSAv6, with Cache and TLB Type Registers implemented.

`0011` Support for VMSAv7, with support for remapping and the Access flag. ARMv7-A profile.

`0100` As for `0011`, and adds support for the PXN bit in the Short-descriptor translation table format descriptors.

`0101` As for `0100`, and adds support for the Long-descriptor translation table format.

All other values are reserved.

In ARMv8-A the only permitted value is `0101`.

### Accessing the ID_MMFR0:

To access the ID_MMFR0:

```
MRC p15,0,<Rt>,c0,c1,4 ; Read ID_MMFR0 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0001 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.85 ID_MMFR1, Memory Model Feature Register 1

The ID_MMFR1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_MMFR0, ID_MMFR2, and ID_MMFR3.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

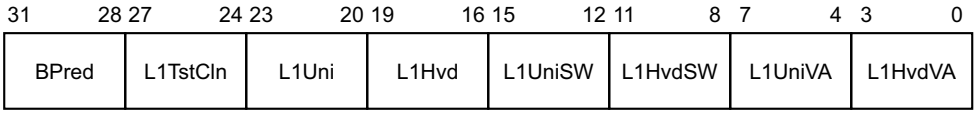
AArch32 System register ID_MMFR1 is architecturally mapped to AArch64 System register ID_MMFR1_EL1.

**Attributes**

ID_MMFR1 is a 32-bit register.

### Field descriptions

The ID_MMFR1 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**BPred, bits [31:28]**

Branch Predictor. Indicates branch predictor management requirements. Defined values are:

`0000` No branch predictor, or no MMU present. Implies a fixed MPU configuration.

`0001` Branch predictor requires flushing on:

- Enabling or disabling a stage of address translation.
- Writing new data to instruction locations.
- Writing new mappings to the translation tables.
- Changes to the TTBR0, TTBR1, or TTBCR registers.
- Changes to the ContextID or ASID, or to the FCSE ProcessID if this is supported.

`0010` Branch predictor requires flushing on:

- Enabling or disabling a stage of address translation.
- Writing new data to instruction locations.
- Writing new mappings to the translation tables.
- Any change to the TTBR0, TTBR1, or TTBCR registers without a change to the corresponding ContextID or ASID, or FCSE ProcessID if this is supported.

`0011` Branch predictor requires flushing only on writing new data to instruction locations.

`0100` For execution correctness, branch predictor requires no flushing at any time.

All other values are reserved.

In ARMv8-A the permitted values are `0010`, `0011`, or `0100`. For values other than `0000` and `0100` the ARM Architecture Reference Manual, or the product documentation, might give more information about the required maintenance.

**L1TstCln, bits [27:24]**

Level 1 cache Test and Clean. Indicates the supported Level 1 data cache test and clean operations, for Harvard or unified cache implementations. Defined values are:

`0000` None supported.

`0001` Supported Level 1 data cache test and clean operations are:

- Test and clean data cache.

`0010` As for `0001`, and adds:

- Test, clean, and invalidate data cache.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**L1Uni, bits [23:20]**

Level 1 Unified cache. Indicates the supported entire Level 1 cache maintenance operations, for a unified cache implementation. Defined values are:

`0000` None supported.

`0001` Supported entire Level 1 cache operations are:

- Invalidate cache, including branch predictor if appropriate.
- Invalidate branch predictor, if appropriate.

`0010` As for `0001`, and adds:

- Clean cache, using a recursive model that uses the cache dirty status bit.
- Clean and invalidate cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**L1Hvd, bits [19:16]**

Level 1 Harvard cache. Indicates the supported entire Level 1 cache maintenance operations, for a Harvard cache implementation. Defined values are:

`0000` None supported.

`0001` Supported entire Level 1 cache operations are:

- Invalidate instruction cache, including branch predictor if appropriate.
- Invalidate branch predictor, if appropriate.

`0010` As for `0001`, and adds:

- Invalidate data cache.
- Invalidate data cache and instruction cache, including branch predictor if appropriate.

`0011` As for `0010`, and adds:

- Clean data cache, using a recursive model that uses the cache dirty status bit.
- Clean and invalidate data cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**L1UniSW, bits [15:12]**

Level 1 Unified cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a unified cache implementation. Defined values are:

`0000` None supported.

`0001` Supported Level 1 unified cache line maintenance operations by set/way are:

- Clean cache line by set/way.

`0010` As for `0001`, and adds:

- Clean and invalidate cache line by set/way.

`0011` As for `0010`, and adds:

- Invalidate cache line by set/way.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**L1HvdSW, bits [11:8]**

Level 1 Harvard cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a Harvard cache implementation. Defined values are:

`0000` None supported.

`0001` Supported Level 1 Harvard cache line maintenance operations by set/way are:

- Clean data cache line by set/way.
- Clean and invalidate data cache line by set/way.

`0010` As for `0001`, and adds:

- Invalidate data cache line by set/way.

`0011` As for `0010`, and adds:

- Invalidate instruction cache line by set/way.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**L1UniVA, bits [7:4]**

Level 1 Unified cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a unified cache implementation. Defined values are:

`0000` None supported.

`0001` Supported Level 1 unified cache line maintenance operations by VA are:

- Clean cache line by VA.
- Invalidate cache line by VA.
- Clean and invalidate cache line by VA.

`0010` As for `0001`, and adds:

- Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**L1HvdVA, bits [3:0]**

Level 1 Harvard cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a Harvard cache implementation. Defined values are:

`0000` None supported.

`0001` Supported Level 1 Harvard cache line maintenance operations by VA are:

- Clean data cache line by VA.
- Invalidate data cache line by VA.
- Clean and invalidate data cache line by VA.
- Clean instruction cache line by VA.

`0010` As for `0001`, and adds:

- Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

## Accessing the ID_MMFR1:

To access the ID_MMFR1:

```
MRC p15,0,<Rt>,c0,c1,5 ; Read ID_MMFR1 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0001 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.86 ID_MMFR2, Memory Model Feature Register 2

The ID_MMFR2 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_MMFR0, ID_MMFR1, and ID_MMFR3.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

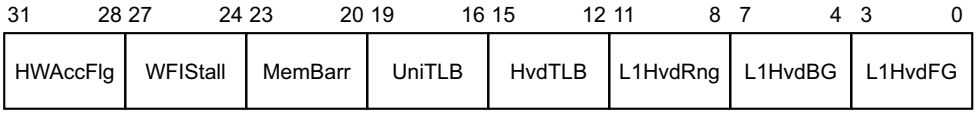
AArch32 System register ID_MMFR2 is architecturally mapped to AArch64 System register ID_MMFR2_EL1.

**Attributes**

ID_MMFR2 is a 32-bit register.

### Field descriptions

The ID_MMFR2 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**HWAccFlg, bits [31:28]**

Hardware Access Flag. In earlier versions of the ARM Architecture, this field indicates support for a Hardware Access flag, as part of the VMSAv7 implementation. Defined values are:

`0000` Not supported.

`0001` Support for VMSAv7 Access flag, updated in hardware.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**WFIStall, bits [27:24]**

Wait For Interrupt Stall. Indicates the support for Wait For Interrupt (WFI) stalling. Defined values are:

`0000` Not supported.

`0001` Support for WFI stalling.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and `0001`.

**MemBarr, bits [23:20]**

Memory Barrier. Indicates the supported CP15 memory barrier operations:

`0000` None supported.

`0001` Supported CP15 Memory barrier operations are:

- Data Synchronization Barrier (DSB), which in previous versions of the ARM architecture was named Data Write Barrier (DWB).

`0010` As for `0001`, and adds:

- Instruction Synchronization Barrier (ISB), which in previous versions of the ARM architecture was called Prefetch Flush.
- Data Memory Barrier (DMB).

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

ARM deprecates the use of these operations. ID_ISAR4.Barrier_instrs indicates the level of support for the preferred barrier instructions.

**UniTLB, bits [19:16]**

Unified TLB. Indicates the supported TLB maintenance operations, for a unified TLB implementation. Defined values are:

`0000` Not supported.

`0001` Supported unified TLB maintenance operations are:

- Invalidate all entries in the TLB.
- Invalidate TLB entry by VA.

`0010` As for `0001`, and adds:

- Invalidate TLB entries by ASID match.

`0011` As for `0010`, and adds:

- Invalidate instruction TLB and data TLB entries by VA All ASID. This is a shared unified TLB operation.

`0100` As for `0011`, and adds:

- Invalidate Hyp mode unified TLB entry by VA.
- Invalidate entire Non-secure PL1&0 unified TLB.
- Invalidate entire Hyp mode unified TLB.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0101 As for 0100, and adds the following operations: TLBIMVALIS, TLBIMVAALIS, TLBIMVALHIS, TLBIMVAL, TLBIMVAAL, TLBIMVALH.

0110 As for 0101, and adds the following operations: TLBIIPAS2IS, TLBIIPAS2LIS, TLBIIPAS2, TLBIIPAS2L.

All other values are reserved.

In ARMv8-A the only permitted value is 0110.

**HvdTLB, bits [15:12]**

If the Unified TLB field (UniTLB, bits [19:16]) is not 0000, then the meaning of this field is IMPLEMENTATION DEFINED. ARM deprecates the use of this field by software.

**L1HvdRng, bits [11:8]**

Level 1 Harvard cache Range. Indicates the supported Level 1 cache maintenance range operations, for a Harvard cache implementation. Defined values are:

0000 Not supported.

0001 Supported Level 1 Harvard cache maintenance range operations are:

- Invalidate data cache range by VA.
- Invalidate instruction cache range by VA.
- Clean data cache range by VA.
- Clean and invalidate data cache range by VA.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**L1HvdBG, bits [7:4]**

Level 1 Harvard cache Background fetch. Indicates the supported Level 1 cache background fetch operations, for a Harvard cache implementation. When supported, background fetch operations are non-blocking operations. Defined values are:

0000 Not supported.

0001 Supported Level 1 Harvard cache background fetch operations are:

- Fetch instruction cache range by VA.
- Fetch data cache range by VA.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**L1HvdFG, bits [3:0]**

Level 1 Harvard cache Foreground fetch. Indicates the supported Level 1 cache foreground fetch operations, for a Harvard cache implementation. When supported, foreground fetch operations are blocking operations. Defined values are:

0000 Not supported.

0001 Supported Level 1 Harvard cache foreground fetch operations are:

- Fetch instruction cache range by VA.
- Fetch data cache range by VA.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

## Accessing the ID_MMFR2:

To access the ID_MMFR2:

```
MRC p15,0,<Rt>,c0,c1,6 ; Read ID_MMFR2 into Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0001 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.87 ID_MMFR3, Memory Model Feature Register 3

The ID_MMFR3 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_MMFR0, ID_MMFR1, and ID_MMFR2.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

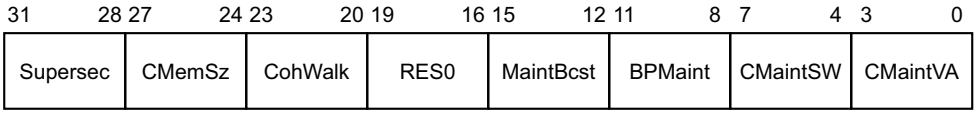
AArch32 System register ID_MMFR3 is architecturally mapped to AArch64 System register ID_MMFR3_EL1.

**Attributes**

ID_MMFR3 is a 32-bit register.

### Field descriptions

The ID_MMFR3 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Supersec, bits [31:28]**

Supersections. On a VMSA implementation, indicates whether Supersections are supported. Defined values are:

`0000` Supersections supported.

`1111` Supersections not supported.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and `1111`.

**CMemSz, bits [27:24]**

Cached Memory Size. Indicates the physical memory size supported by the caches. Defined values are:

`0000` 4GB, corresponding to a 32-bit physical address range.

`0001` 64GB, corresponding to a 36-bit physical address range.

`0010` 1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In ARMv8-A the permitted values are `0000`, `0001`, and `0010`.

**CohWalk, bits [23:20]**

Coherent Walk. Indicates whether Translation table updates require a clean to the point of unification. Defined values are:

`0000` Updates to the translation tables require a clean to the point of unification to ensure visibility by subsequent translation table walks.

`0001` Updates to the translation tables do not require a clean to the point of unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Bits [19:16]**

Reserved, RES0.

**MaintBcst, bits [15:12]**

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast. Defined values are:

`0000` Cache, TLB, and branch predictor operations only affect local structures.

`0001` Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.

`0010` Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

**BPMaint, bits [11:8]**

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations, Defined values are:

`0000` None supported.

`0001` Supported branch predictor maintenance operations are:

- Invalidate all branch predictors.

`0010` As for `0001`, and adds:

- Invalidate branch predictors by VA.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In ARMv8-A the only permitted value is 0010.

**CMaintSW, bits [7:4]**

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches. Defined values are:

0000 None supported.

0001 Supported hierarchical cache maintenance instructions by set/way are:

- Invalidate data cache by set/way.
- Clean data cache by set/way.
- Clean and invalidate data cache by set/way.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

In a unified cache implementation, the data cache maintenance instructions apply to the unified caches.

**CMaintVA, bits [3:0]**

Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches. Defined values are:

0000 None supported.

0001 Supported hierarchical cache maintenance instructions by VA are:

- Invalidate data cache by VA.
- Clean data cache by VA.
- Clean and invalidate data cache by VA.
- Invalidate instruction cache by VA.
- Invalidate all instruction cache entries.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

In a unified cache implementation, data cache maintenance instructions apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

### Accessing the ID_MMFR3:

To access the ID_MMFR3:

```
MRC p15,0,<Rt>,c0,c1,7 ; Read ID_MMFR3 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0001 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.88 ID_MMFR4, Memory Model Feature Register 4

The ID_MMFR4 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If EL2 is using AArch32 and HCR.TID3==1 then:

- If the register is not RAZ/WI then Non-secure EL1 read accesses to this register are trapped to Hyp mode.
- Otherwise, it is IMPLEMENTATION DEFINED whether Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If EL2 is using AArch64 and HCR_EL2.TID3==1 then:

- If the register is not RAZ/WI then Non-secure EL1 read accesses to this register are trapped to EL2.
- Otherwise, it is IMPLEMENTATION DEFINED whether Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register ID_MMFR4 is architecturally mapped to AArch64 System register ID_MMFR4_EL1.

In an implementation that does not include ACTLR2 and HACTLR2 this register is RAZ/WI.

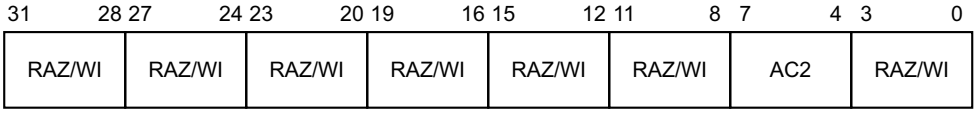
**Attributes**

ID_MMFR4 is a 32-bit register.

#### Field descriptions

The ID_MMFR4 bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**AC2, bits [7:4]**

Indicates the extension of the ACTLR and HACTLR registers using ACTLR2 and HACTLR2. Defined values are:

`0000` ACTLR2 and HACTLR2 are not implemented.

`0001` ACTLR2 and HACTLR2 are implemented.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and `0001`.

## Accessing the ID_MMFR4:

To access the ID_MMFR4:

```
MRC p15,0,<Rt>,c0,c2,6 ; Read ID_MMFR4 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0010 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.89 ID_PFR0, Processor Feature Register 0

The ID_PFR0 characteristics are:

**Purpose**

Gives top-level information about the instruction sets supported by the PE in AArch32.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_PFR1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

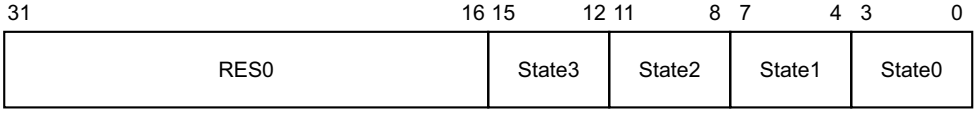
AArch32 System register ID_PFR0 is architecturally mapped to AArch64 System register ID_PFR0_EL1.

**Attributes**

ID_PFR0 is a 32-bit register.

#### Field descriptions

The ID_PFR0 bit assignments are:




**Bits [31:16]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**State3, bits [15:12]**

T32EE instruction set support. Defined values are:

0000 Not implemented.

0001 T32EE instruction set implemented.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**State2, bits [11:8]**

Jazelle extension support. Defined values are:

0000 Not implemented.

0001 Jazelle extension implemented, without clearing of JOSCR.CV on exception entry.

0010 Jazelle extension implemented, with clearing of JOSCR.CV on exception entry.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**State1, bits [7:4]**

T32 instruction set support. Defined values are:

0000 T32 instruction set not implemented.

0001 T32 encodings before the introduction of Thumb-2 technology implemented:

- All instructions are 16-bit.
- A BL or BLX is a pair of 16-bit instructions.
- 32-bit instructions other than BL and BLX cannot be encoded.

0011 T32 encodings after the introduction of Thumb-2 technology implemented, for all 16-bit and 32-bit T32 basic instructions.

All other values are reserved.

In ARMv8-A the only permitted value is 0011.

**State0, bits [3:0]**

A32 instruction set support. Defined values are:

0000 A32 instruction set not implemented.

0001 A32 instruction set implemented.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

## Accessing the ID_PFR0:

To access the ID_PFR0:

```
MRC p15,0,<Rt>,c0,c1,0 ; Read ID_PFR0 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.90 ID_PFR1, Processor Feature Register 1

The ID_PFR1 characteristics are:

**Purpose**

Gives information about the AArch32 programmers' model.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

Must be interpreted with ID_PFR0.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

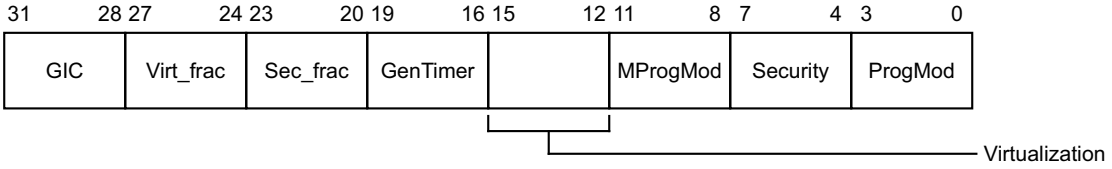
AArch32 System register ID_PFR1 is architecturally mapped to AArch64 System register ID_PFR1_EL1.

**Attributes**

ID_PFR1 is a 32-bit register.

### Field descriptions

The ID_PFR1 bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**GIC, bits [31:28]**

System register GIC CPU interface. Defined values are:

`0000` No System register interface to the GIC CPU interface is supported.

`0001` System register interface to the GIC CPU interface is supported.

All other values are reserved.

**Virt_frac, bits [27:24]**

Virtualization fractional field. When the Virtualization field is `0000`, determines the support for features from the ARMv7 Virtualization Extensions. Defined values are:

`0000` No features from the ARMv7 Virtualization Extensions are implemented.

`0001` The following features of the ARMv7 Virtualization Extensions are implemented:

- The SCR.SIF bit, if EL3 is implemented.
- The modifications to the SCR.AW and SCR.FW bits described in the Virtualization Extensions, if EL3 is implemented.
- The MSR (Banked register) and MRS (Banked register) instructions.
- The ERET instruction.

All other values are reserved.

In ARMv8-A the permitted values are `0000` and `0001`.

**Note**

The ID_ISAR registers do not identify whether the instructions added by the Virtualization Extensions are implemented.

This field is only valid when ID_PFR1[15:12] == 0, otherwise it holds the value `0000`.

**Sec_frac, bits [23:20]**

Security fractional field. When the Security field is `0000`, determines the support for features from the ARMv7 Security Extensions. Defined values are:

`0000` No features from the ARMv7 Security Extensions are implemented.

`0001` The following features from the ARMv7 Security Extensions are implemented:

- The VBAR register.
- The TTBCR.PD0 and TTBCR.PD1 bits.

`0010` As for `0001`, plus the ability to access Secure or Non-secure physical memory is supported.

All other values are reserved.

In ARMv8-A the permitted values are `0000`, `0001`, and `0010`.

This field is only valid when ID_PFR1[7:4] == 0, otherwise it holds the value `0000`.

**GenTimer, bits [19:16]**

Generic Timer support. Defined values are:

`0000` Not implemented.

`0001` Generic Timer implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**Virtualization, bits [15:12]**

Virtualization support. Defined values are:

`0000` EL2, Hyp mode, and the HVC instruction not implemented.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

0001 EL2, Hyp mode, the HVC instruction, and all the features described by Virt_frac == 0001 implemented.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

**Note**

The ID_ISARs do not identify whether the HVC instruction is implemented.

**MProgMod, bits [11:8]**

M profile programmers' model support. Defined values are:

0000 Not supported.

0010 Support for two-stack programmers' model.

All other values are reserved.

In ARMv8-A the only permitted value is 0000.

**Security, bits [7:4]**

Security support. Defined values are:

0000 EL3, Monitor mode, and the SMC instruction not implemented.

0001 EL3, Monitor mode, the SMC instruction, and all the features described by Sec_frac == 0001 implemented.

0010 As for 0001, and adds the ability to set the NSACR.RFR bit. Not permitted in ARMv8 as the NSACR.RFR bit is RES0.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0001.

**ProgMod, bits [3:0]**

Support for the standard programmers' model for ARMv4 and later. Model must support User, FIQ, IRQ, Supervisor, Abort, Undefined, and System modes. Defined values are:

0000 Not supported.

0001 Supported.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

## Accessing the ID_PFR1:

To access the ID_PFR1:

```
MRC p15,0,<Rt>,c0,c1,1 ; Read ID_PFR1 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.91 IFAR, Instruction Fault Address Register

The IFAR characteristics are:

**Purpose**

Holds the virtual address of the faulting address that caused a synchronous Prefetch Abort exception.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

IFAR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

IFAR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

IFAR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T6==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T6==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register IFAR(NS) is architecturally mapped to AArch64 System register FAR_EL1[63:32].

AArch32 System register IFAR(S) is architecturally mapped to AArch32 System register HIFAR when EL2 is implemented.

AArch32 System register IFAR(S) is architecturally mapped to AArch64 System register FAR_EL2[63:32] when EL2 is implemented.

RW fields in this register reset to architecturally UNKNOWN values.

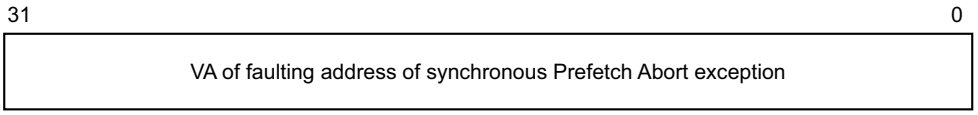

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attributes**

IFAR is a 32-bit register.

## Field descriptions

The IFAR bit assignments are:




**Bits [31:0]**

VA of faulting address of synchronous Prefetch Abort exception.

## Accessing the IFAR:

To access the IFAR:

```
MRC p15,0,<Rt>,c6,c0,2 ; Read IFAR into Rt
MCR p15,0,<Rt>,c6,c0,2 ; Write Rt to IFAR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0110 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.92 IFSR, Instruction Fault Status Register

The IFSR characteristics are:

**Purpose**

Holds status information about the last instruction fault.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

IFSR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

IFSR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

IFSR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T5==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T5==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register IFSR is architecturally mapped to AArch64 System register IFSR32_EL2.

The current translation table format determines which format of the register is used.

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

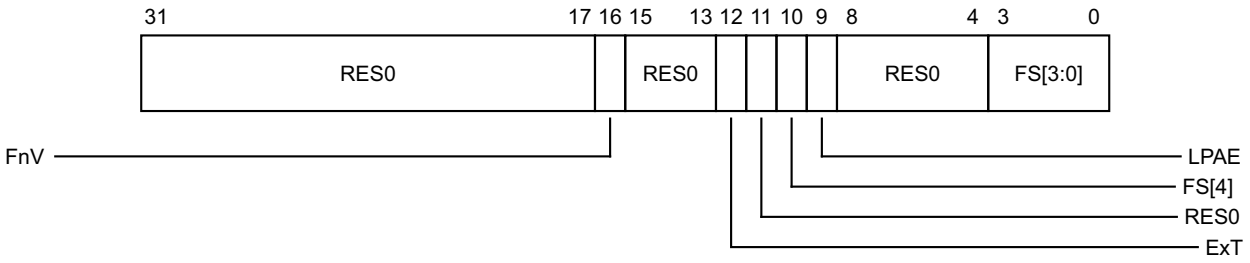
IFSR is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The IFSR bit assignments are:

***When TTBCR.EAE==0:***




**Bits [31:17]**

Reserved, RES0.

**FnV, bit [16]**

FAR not Valid, for a Synchronous external abort other than a Synchronous external abort on a translation table walk.

`0` IFAR is valid.

`1` IFAR is not valid, and holds an UNKNOWN value.

This field is only valid for a Synchronous external abort other than a Synchronous external abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

**Bits [15:13]**

Reserved, RES0.

**ExT, bit [12]**

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

**Bit [11]**

Reserved, RES0.

**FS[4], bit [10]**

See FS[3:0], bits [3:0] for description of the FS field.

**LPAE, bit [9]**

On taking a Data Abort exception, this bit is set as follows:

`0` Using the Short-descriptor translation table formats.

`1` Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

**Bits [8:4]**

Reserved, RES0.

**FS[3:0], bits [3:0]**

Fault status bits. Interpreted with bit [10]. Possible values of FS[4:0] are:

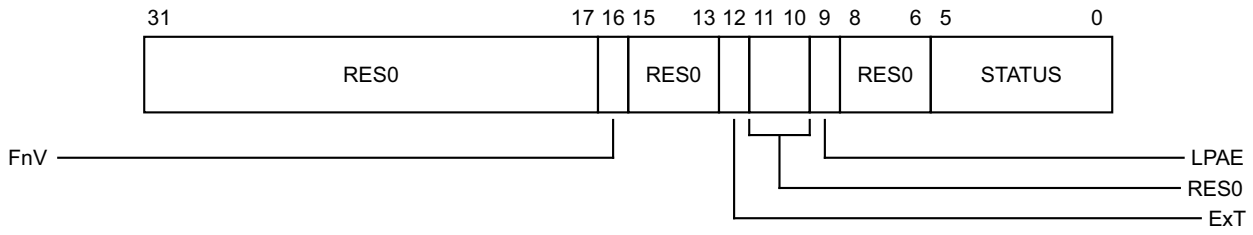
`00001` PC alignment fault


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| | |
|---|---|
| 00010 | Debug event |
| 00011 | Access flag fault, level 1 |
| 00101 | Translation fault, level 1 |
| 00110 | Access flag fault, level 2 |
| 00111 | Translation fault, level 2 |
| 01000 | Synchronous external abort, other than synchronous parity or ECC error, not on translation table walk |
| 01001 | Domain fault, level 1 |
| 01011 | Domain fault, level 2 |
| 01100 | Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 1 |
| 01101 | Permission fault, level 1 |
| 01110 | Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 2 |
| 01111 | Permission fault, level 2 |
| 10000 | TLB conflict abort |
| 10100 | IMPLEMENTATION DEFINED fault (Lockdown fault) |
| 11001 | Synchronous parity or ECC error on memory access, not on translation table walk |
| 11100 | Synchronous parity or ECC error on translation table walk, level 1 |
| 11110 | Synchronous parity or ECC error on translation table walk, level 2 |

All other values are reserved.

***When TTBCR.EAE==1:***




**Bits [31:17]**

Reserved, RES0.

**FnV, bit [16]**

FAR not Valid, for a Synchronous external abort other than a Synchronous external abort on a translation table walk.

| | |
|---|---|
| 0 | IFAR is valid. |
| 1 | IFAR is not valid, and holds an UNKNOWN value. |

This field is only valid for a Synchronous external abort other than a Synchronous external abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

**Bits [15:13]**

Reserved, RES0.

**ExT, bit [12]**

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For aborts other than external aborts this bit always returns 0.

**Bits [11:10]**

Reserved, RES0.

**LPAE, bit [9]**

On taking a Data Abort exception, this bit is set as follows:

`0` Using the Short-descriptor translation table formats.

`1` Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

**Bits [8:6]**

Reserved, RES0.

**STATUS, bits [5:0]**

Fault status bits. Possible values of this field are:

`000000` Address size fault in TTBR0 or TTBR1

`000001` Address size fault, level 1

`000010` Address size fault, level 2

`000011` Address size fault, level 3

`000101` Translation fault, level 1

`000110` Translation fault, level 2

`000111` Translation fault, level 3

`001001` Access flag fault, level 1

`001010` Access flag fault, level 2

`001011` Access flag fault, level 3

`001101` Permission fault, level 1

`001110` Permission fault, level 2

`001111` Permission fault, level 3

`010000` Synchronous external abort, other than synchronous parity or ECC error, not on translation table walk

`010101` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 1

`010110` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 2

`010111` Synchronous external abort, other than synchronous parity or ECC error, on translation table walk, level 3

`011000` Synchronous parity or ECC error on memory access, not on translation table walk

`011101` Synchronous parity or ECC error on memory access on translation table walk, level 1

`011110` Synchronous parity or ECC error on memory access on translation table walk, level 2

`011111` Synchronous parity or ECC error on memory access on translation table walk, level 3

`100001` Alignment fault

`100010` Debug event

`110000` TLB conflict abort

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a fault at level 1.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

### Accessing the IFSR:

To access the IFSR:

```
MRC p15,0,<Rt>,c5,c0,1 ; Read IFSR into Rt
MCR p15,0,<Rt>,c5,c0,1 ; Write Rt to IFSR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0101 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.93 ISR, Interrupt Status Register

The ISR characteristics are:

**Purpose**

Shows whether an IRQ, FIQ, or external abort is pending. If EL2 is implemented, an indicated pending abort might be a physical abort or a virtual abort.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T12==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T12==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

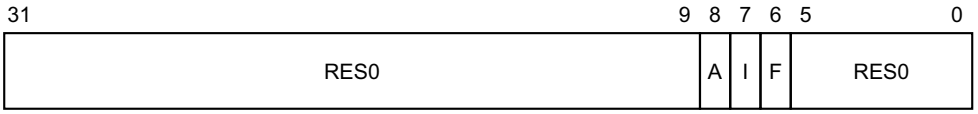
AArch32 System register ISR is architecturally mapped to AArch64 System register ISR_EL1.

**Attributes**

ISR is a 32-bit register.

## Field descriptions

The ISR bit assignments are:




**Bits [31:9]**

Reserved, RES0.

**A, bit [8]**

Asynchronous external abort pending bit:

0 No pending asynchronous external abort.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 An asynchronous external abort is pending.

**I, bit [7]**

IRQ pending bit. Indicates whether an IRQ interrupt is pending:

0 No pending IRQ.

1 An IRQ interrupt is pending.

**F, bit [6]**

FIQ pending bit. Indicates whether an FIQ interrupt is pending.

0 No pending FIQ.

1 An FIQ interrupt is pending.

**Bits [5:0]**

Reserved, RES0.

### Accessing the ISR:

To access the ISR:

```
MRC p15,0,<Rt>,c12,c1,0 ; Read ISR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1100 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.94 ITLBIALL, Instruction TLB Invalidate All

The ITLBIALL characteristics are:

**Purpose**

Invalidate all instruction TLB entries for the PL1&0 translation regime, subject to the Privilege level and Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIALL.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

ITLBIALL is a 32-bit system operation.

#### Field descriptions

The ITLBIALL operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the ITLBIALL operation:

To perform the ITLBIALL operation:

```
MCR p15,0,<Rt>,c8,c5,0 ; ITLBIALL operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.95 ITLBIASID, Instruction TLB Invalidate by ASID match

The ITLBIASID characteristics are:

**Purpose**

Invalidate instruction TLB entries for stage 1 of the PL1&0 translation regime that match the given ASID, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIASID.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

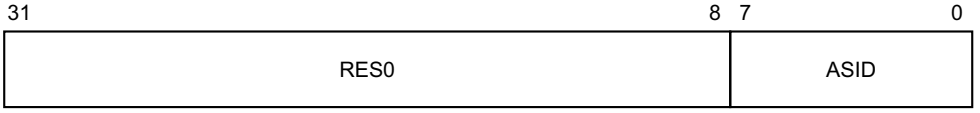
There are no configuration notes.

**Attributes**

ITLBIASID is a 32-bit system operation.

#### Field descriptions

The ITLBIASID input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this operation.

### Performing the ITLBIASID operation:

To perform the ITLBIASID operation:

```
MCR p15,0,<Rt>,c8,c5,2 ; ITLBIASID operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0101 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.96 ITLBIMVA, Instruction TLB Invalidate by VA

The ITLBIMVA characteristics are:

**Purpose**

Invalidate TLB entries for stage 1 of the PL1&0 translation regime stage 1 that match the given VA and ASID, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIMVA.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

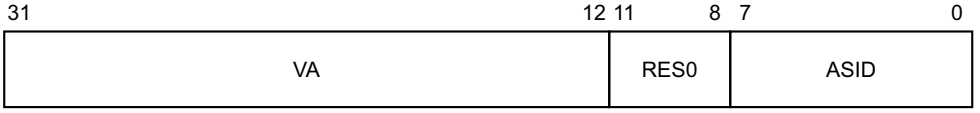
There are no configuration notes.

**Attributes**

ITLBIMVA is a 32-bit system operation.

### Field descriptions

The ITLBIMVA input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

**Bits [11:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

### Performing the ITLBIMVA operation:

To perform the ITLBIMVA operation:

```
MCR p15,0,<Rt>,c8,c5,1 ; ITLBIMVA operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0101 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.97 JIDR, Jazelle ID Register

The JIDR characteristics are:

**Purpose**

A Jazelle register, which identified the Jazelle architecture version.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| IMP DEF | IMP DEF | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| IMP DEF | RO | RO |

For EL0 using AArch32, it is IMPLEMENTATION DEFINED whether JIDR is RAZ or UNDEFINED.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID0==1, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID0==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

**Configurations**

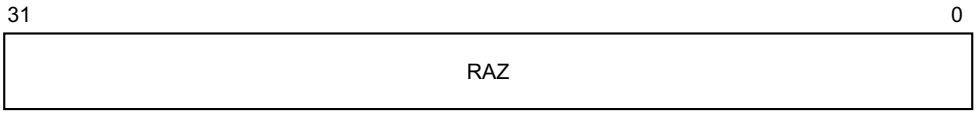
There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

JIDR is a 32-bit register.

#### Field descriptions

The JIDR bit assignments are:




**Bits [31:0]**

Reserved, RAZ.

#### Accessing the JIDR:

To access the JIDR:

```
MRC p14,7,<Rt>,c0,c0,0 ; Read JIDR into Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 111 | 0000 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.98 JMCR, Jazelle Main Configuration Register

The JMCR characteristics are:

**Purpose**

A Jazelle register, which provides control of the Jazelle extension.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| IMP DEF | IMP DEF | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| IMP DEF | RW | RW |

For EL0 using AArch32, it is IMPLEMENTATION DEFINED whether JMCR is RAZ/WI or UNDEFINED.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

JMCR is a 32-bit register.

#### Field descriptions

The JMCR bit assignments are:




**Bits [31:0]**

Reserved, RAZ/WI.

#### Accessing the JMCR:

To access the JMCR:

```
MRC p14,7,<Rt>,c2,c0,0 ; Read JMCR into Rt
MCR p14,7,<Rt>,c2,c0,0 ; Write Rt to JMCR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 111 | 0010 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.99 JOSCR, Jazelle OS Control Register

The JOSCR characteristics are:

**Purpose**

A Jazelle register, which provides operating system control of the Jazelle Extension.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| IMP DEF | IMP DEF | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| IMP DEF | RW | RW |

For EL0 using AArch32, it is IMPLEMENTATION DEFINED whether JOSCR is RAZ/WI or UNDEFINED.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

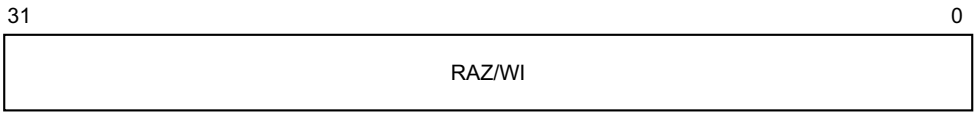
There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

JOSCR is a 32-bit register.

#### Field descriptions

The JOSCR bit assignments are:




**Bits [31:0]**

Reserved, RAZ/WI.

#### Accessing the JOSCR:

To access the JOSCR:

```
MRC p14,7,<Rt>,c1,c0,0 ; Read JOSCR into Rt
MCR p14,7,<Rt>,c1,c0,0 ; Write Rt to JOSCR
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 111 | 0001 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.100 MAIR0, Memory Attribute Indirection Register 0

The MAIR0 characteristics are:

**Purpose**

Along with MAIR1, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

MAIR0(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

MAIR0(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

MAIR0 is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

MAIR0 and PRRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in PRRR.
- When it is set to 1, the register is as described in MAIR0.

AttrIndx[2], from the translation table descriptor, selects the appropriate MAIR: setting AttrIndx[2] to 0 selects MAIR0.

In an implementation that includes EL3:

- MAIR0(S) gives the value for memory accesses from Secure state.
- MAIR0(NS) gives the value for memory accesses from Non-secure states other than Hyp mode.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If HSTR.T10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T10==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register MAIR0 is architecturally mapped to AArch64 System register MAIR_EL1[31:0].

MAIR0 and PRRR are the same register.

When EL3 is using AArch32, write access to MAIR0(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

RW fields in this register reset to architecturally UNKNOWN values.

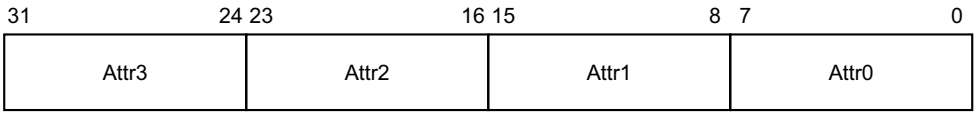
**Attributes**

MAIR0 is a 32-bit register.

## Field descriptions

The MAIR0 bit assignments are:

***When TTBCR.EAE==1:***




**Attr<n>, bits [8n+7:8n], for n = 0 to 3**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

| Attr<n>[7:4] | Meaning |
|---|---|
| `0000` | Device memory. See encoding of Attr<n>[3:0] for the type of Device memory. |
| `00RW`, RW not `00` | Normal Memory, Outer Write-through transient |
| `0100` | Normal Memory, Outer Non-cacheable |
| `01RW`, RW not `00` | Normal Memory, Outer Write-back transient |
| `10RW` | Normal Memory, Outer Write-through non-transient |
| `11RW` | Normal Memory, Outer Write-back non-transient |

R = Outer Read Allocate Policy, W = Outer Write Allocate Policy.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The meaning of bits [3:0] depends on the value of bits [7:4]:

| Attr<n>[3:0] | Meaning when Attr<n>[7:4] is 0000 | Meaning when Attr<n>[7:4] is not 0000 |
|---|---|---|
| 0000 | Device-nGnRnE memory | UNPREDICTABLE |
| 00RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through transient |
| 0100 | Device-nGnRE memory | Normal memory, Inner Non-cacheable |
| 01RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back transient |
| 1000 | Device-nGRE memory | Normal Memory, Inner Write-through non-transient (RW=00) |
| 10RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through non-transient |
| 1100 | Device-GRE memory | Normal Memory, Inner Write-back non-transient (RW=00) |
| 11RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back non-transient |

R = Inner Read Allocate Policy, W = Inner Write Allocate Policy.

The R and W bits in some Attr<n> fields have the following meanings:

| R or W | Meaning |
|---|---|
| 0 | Do not allocate |
| 1 | Allocate |

### Accessing the MAIR0:

To access the MAIR0 when TTBCR.EAE==1:

```
MRC p15,0,<Rt>,c10,c2,0 ; Read MAIR0 into Rt
MCR p15,0,<Rt>,c10,c2,0 ; Write Rt to MAIR0
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1010 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.101 MAIR1, Memory Attribute Indirection Register 1

The MAIR1 characteristics are:

**Purpose**

Along with MAIR0, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

MAIR1(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

MAIR1(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

MAIR1 is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

MAIR1 and NMRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in NMRR.
- When it is set to 1, the register is as described in MAIR1.

AttrIndx[2], from the translation table descriptor, selects the appropriate MAIR: setting AttrIndx[2] to 1 selects MAIR1.

In an implementation that includes EL3:

- MAIR1(S) gives the value for memory accesses from Secure state.
- MAIR1(NS) gives the value for memory accesses from Non-secure states other than Hyp mode.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If HSTR.T10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T10==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register MAIR1 is architecturally mapped to AArch64 System register MAIR_EL1[63:32].

MAIR1 and NMRR are the same register.

When EL3 is using AArch32, write access to MAIR1(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

RW fields in this register reset to architecturally UNKNOWN values.

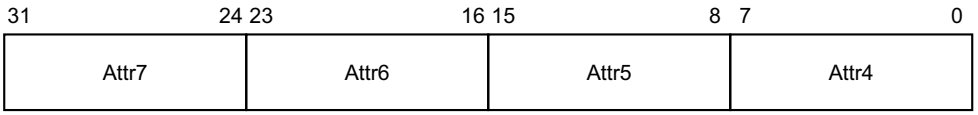
**Attributes**

MAIR1 is a 32-bit register.

## Field descriptions

The MAIR1 bit assignments are:

***When TTBCR.EAE==1:***




**Attr<n>, bits [8(n-4)+7:8(n-4)], for n = 4 to 7**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

| Attr<n>[7:4] | Meaning |
|---|---|
| `0000` | Device memory. See encoding of Attr<n>[3:0] for the type of Device memory. |
| `00RW`, RW not `00` | Normal Memory, Outer Write-through transient |
| `0100` | Normal Memory, Outer Non-cacheable |
| `01RW`, RW not `00` | Normal Memory, Outer Write-back transient |
| `10RW` | Normal Memory, Outer Write-through non-transient |
| `11RW` | Normal Memory, Outer Write-back non-transient |

R = Outer Read Allocate Policy, W = Outer Write Allocate Policy.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The meaning of bits [3:0] depends on the value of bits [7:4]:

| Attr<n>[3:0] | Meaning when Attr<n>[7:4] is 0000 | Meaning when Attr<n>[7:4] is not 0000 |
|---|---|---|
| 0000 | Device-nGnRnE memory | UNPREDICTABLE |
| 00RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through transient |
| 0100 | Device-nGnRE memory | Normal memory, Inner Non-cacheable |
| 01RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back transient |
| 1000 | Device-nGRE memory | Normal Memory, Inner Write-through non-transient (RW=00) |
| 10RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-through non-transient |
| 1100 | Device-GRE memory | Normal Memory, Inner Write-back non-transient (RW=00) |
| 11RW, RW not 00 | UNPREDICTABLE | Normal Memory, Inner Write-back non-transient |

R = Inner Read Allocate Policy, W = Inner Write Allocate Policy.

The R and W bits in some Attr<n> fields have the following meanings:

| R or W | Meaning |
|---|---|
| 0 | Do not allocate |
| 1 | Allocate |

### Accessing the MAIR1:

To access the MAIR1 when TTBCR.EAE==1:

```
MRC p15,0,<Rt>,c10,c2,1 ; Read MAIR1 into Rt
MCR p15,0,<Rt>,c10,c2,1 ; Write Rt to MAIR1
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1010 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.102 MIDR, Main ID Register

The MIDR characteristics are:

**Purpose**

Provides identification information for the PE, including an implementer code for the device and a device ID number.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register MIDR is architecturally mapped to AArch64 System register MIDR_EL1.

AArch32 System register MIDR is architecturally mapped to External register MIDR_EL1.

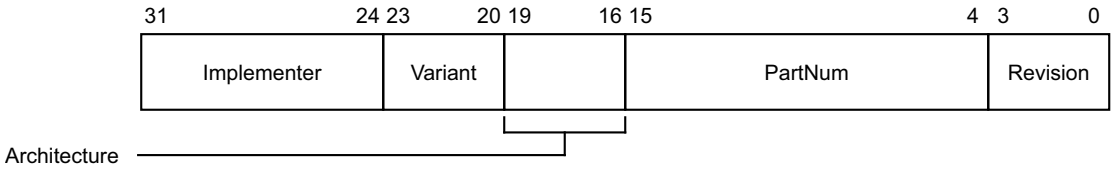
Some fields of the MIDR are IMPLEMENTATION DEFINED. For details of the values of these fields for a particular ARMv8 implementation, and any implementation-specific significance of these values, see the product documentation.

**Attributes**

MIDR is a 32-bit register.

### Field descriptions

The MIDR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by ARM. Assigned codes include the following:

| Hex representation | ASCII representation | Implementer |
|---|---|---|
| 0x41 | A | ARM Limited |
| 0x42 | B | Broadcom Corporation |
| 0x43 | C | Cavium Inc. |
| 0x44 | D | Digital Equipment Corporation |
| 0x49 | I | Infineon Technologies AG |
| 0x4D | M | Motorola or Freescale Semiconductor Inc. |
| 0x4E | N | NVIDIA Corporation |
| 0x50 | P | Applied Micro Circuits Corporation |
| 0x51 | Q | Qualcomm Inc. |
| 0x56 | V | Marvell International Ltd. |
| 0x69 | i | Intel Corporation |

ARM can assign codes that are not published in this manual. All values not assigned by ARM are reserved and must not be used.

**Variant, bits [23:20]**

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

**Architecture, bits [19:16]**

The permitted values of this field are:

0001 ARMv4

0010 ARMv4T

0011 ARMv5 (obsolete)

0100 ARMv5T

0101 ARMv5TE

0110 ARMv5TEJ

0111 ARMv6

1111 Architectural features are individually identified in the ID_* registers, see *Identification registers, functional group* on page G4-4162.

All other values are reserved.

**PartNum, bits [15:4]**

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by ARM, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

**Revision, bits [3:0]**

An IMPLEMENTATION DEFINED revision number for the device.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the MIDR:

To access the MIDR:

```
MRC p15,0,<Rt>,c0,c0,0 ; Read MIDR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.103 MPIDR, Multiprocessor Affinity Register

The MPIDR characteristics are:

**Purpose**

In a multiprocessor system, provides an additional PE identification mechanism for scheduling purposes.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register MPIDR is architecturally mapped to AArch64 System register MPIDR_EL1.

The assigned value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

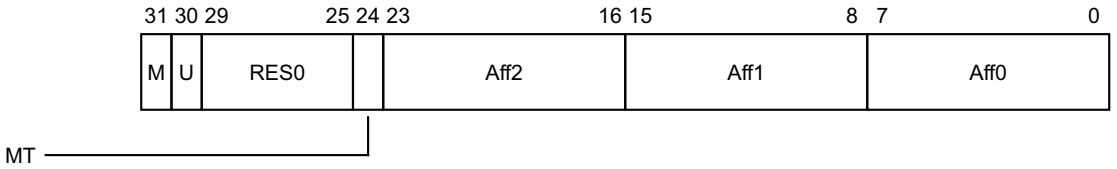
In a uniprocessor system ARM recommends that each Aff<n> field of this register returns a value of 0.

**Attributes**

MPIDR is a 32-bit register.

#### Field descriptions

The MPIDR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**M, bit [31]**

Indicates whether this implementation includes the functionality introduced by the ARMv7 Multiprocessing Extensions. The possible values of this bit are:

0 This implementation does not include the ARMv7 Multiprocessing Extensions functionality.

1 This implementation includes the ARMv7 Multiprocessing Extensions functionality.

In ARMv8 this bit is RES1.

**U, bit [30]**

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system. The possible values of this bit are:

0 Processor is part of a multiprocessor system.

1 Processor is part of a uniprocessor system.

**Bits [29:25]**

Reserved, RES0.

**MT, bit [24]**

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach. The possible values of this bit are:

0 Performance of PEs at the lowest affinity level is largely independent.

1 Performance of PEs at the lowest affinity level is very interdependent.

**Aff2, bits [23:16]**

Affinity level 2. The least significant affinity level field, for this PE in the system.

**Aff1, bits [15:8]**

Affinity level 1. The intermediate affinity level field, for this PE in the system.

**Aff0, bits [7:0]**

Affinity level 0. The most significant affinity level field, for this PE in the system.

## Accessing the MPIDR:

To access the MPIDR:

```
MRC p15,0,<Rt>,c0,c0,5 ; Read MPIDR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0000 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.104 MVBAR, Monitor Vector Base Address Register

The MVBAR characteristics are:

**Purpose**

When EL3 is implemented and can use AArch32, holds the vector base address for any exception that is taken to Monitor mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|---|
| - | - | - | RW | - | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | - |

If EL3 is implemented and is using AArch64, any read or write to MVBAR from Secure EL1 using AArch32 is trapped as an exception to EL3.

Secure software must program the MVBAR with the required initial value as part of the PE boot sequence.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T12==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T12==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

It is IMPLEMENTATION DEFINED whether MVBAR[0] has a fixed value and ignored writes, or takes the last value written to it.

When EL3 is using AArch32, write access to MVBAR(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

On a reset into EL3 using AArch32, the reset value of MVBAR is an IMPLEMENTATION DEFINED choice between:

- MVBAR[31:5] = an IMPLEMENTATION DEFINED value, which might be UNKNOWN.
- MVBAR[4:1] = RES0.
- MVBAR[0] = 0.

And:

- MVBAR[31:1] = an IMPLEMENTATION DEFINED value that is bits[31:1] of the AArch32 reset address.
- MVBAR[0] = 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

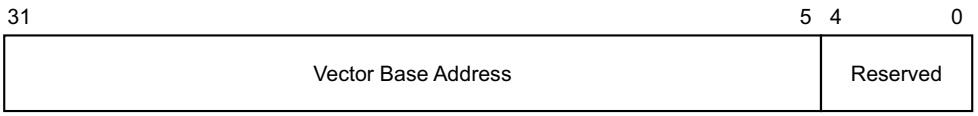
**Attributes**

MVBAR is a 32-bit register.

## Field descriptions

The MVBAR bit assignments are:

***When programmed with a vector base address:***




**Bits [31:5]**

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

**Reserved, bits [4:0]**

Reserved, see Configurations.

## Accessing the MVBAR:

To access the MVBAR:

```
MRC p15,0,<Rt>,c12,c0,1 ; Read MVBAR into Rt
MCR p15,0,<Rt>,c12,c0,1 ; Write Rt to MVBAR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1100 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.105 MVFR0, Media and VFP Feature Register 0

The MVFR0 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | Config-RO | Config-RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | Config-RO | Config-RO |

Must be interpreted with MVFR1 and MVFR2.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CPTR_EL2.TFP==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If CPTR_EL3.TFP==1, EL1 and EL2 read accesses to this register are trapped to EL3.

If HCPTR.TCP10==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If CPACR.cp10==00, PL1 read accesses to this register are UNDEFINED.

If HCPTR.TCP10==1, Non-secure EL2 read accesses to this register are UNDEFINED.

If NSACR.cp10==0, Non-secure EL1 and EL2 read accesses to this register are UNDEFINED.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register MVFR0 is architecturally mapped to AArch64 System register MVFR0_EL1.

Implemented only if the implementation includes floating-point and Advanced SIMD instructions.

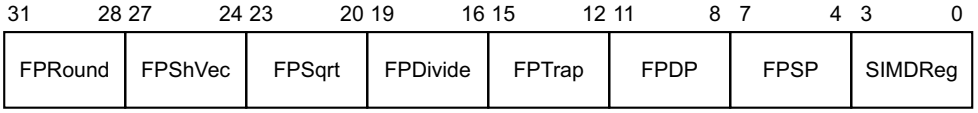
**Attributes**

MVFR0 is a 32-bit register.

#### Field descriptions

The MVFR0 bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**FPRound, bits [31:28]**

Floating-Point Rounding modes. Indicates the rounding modes supported by the VFP floating-point hardware. Defined values are:

`0000` Only Round to Nearest mode supported, except that Round towards Zero mode is supported for VCVT instructions that always use that rounding mode regardless of the FPSCR setting.

`0001` All rounding modes supported.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**FPShVec, bits [27:24]**

Short Vectors. Indicates that the floating-point instructions support the use of short vectors. Defined values are:

`0000` Short vectors not supported.

`0001` Short vector operation supported.

All other values are reserved.

In ARMv8-A the only permitted value is `0000`.

**FPSqrt, bits [23:20]**

Square Root. Indicates the hardware support for the ARMv6 VFP square root operations. Defined values are:

`0000` Not supported in hardware.

`0001` Supported.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

The VSQRT.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VSQRT.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

**FPDivide, bits [19:16]**

Indicates the hardware support for VFP divide operations. Defined values are:

`0000` Not supported in hardware.

`0001` Supported.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

The VDIV.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VDIV.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

**FPTrap, bits [15:12]**

Floating Point Exception Trapping. Indicates whether the VFP hardware implementation supports exception trapping. Defined values are:

`0000` Not supported.

`0001` Supported by the hardware. When exception trapping is supported, support code is needed to handle the trapped exceptions.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A value of 0001 does not indicate that trapped exception handling is available. Because trapped exception handling requires support code, only the support code can provide this information.

**FPDP, bits [11:8]**

Double Precision. Indicates the hardware support for VFP double-precision operations. Defined values are:

0000 Not supported in hardware.

0001 Supported, VFPv2.

0010 Supported, VFPv3, VFPv4, or ARMv8. VFPv3 and ARMv8 add an instruction to load a double-precision floating-point constant, and conversions between double-precision and fixed-point values.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

A value of 0b0001 or 0b0010 indicates support for all VFP double-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F64 is only available if the Square root field is 0001.
- VDIV.F64 is only available if the Divide field is 0001.
- Conversion between double-precision and single-precision is only available if the single-precision field is nonzero.

**FPSP, bits [7:4]**

Single Precision. Indicates the hardware support for VFP single-precision operations. Defined values are:

0000 Not supported in hardware.

0001 Supported, VFPv2.

0010 Supported, VFPv3 or VFPv4. VFPv3 adds an instruction to load a single-precision floating-point constant, and conversions between single-precision and fixed-point values.

All other values are reserved.

In ARMv8-A the only permitted value is 0010.

A value of 0b0001 or 0b0010 indicates support for all VFP single-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F32 is only available if the Square root field is 0001.
- VDIV.F32 is only available if the Divide field is 0001.
- Conversion between double-precision and single-precision is only available if the double-precision field is nonzero.

**SIMDReg, bits [3:0]**

Advanced SIMD registers. Indicates support for the Advanced SIMD register bank. Defined values are:

0000 Not supported.

0001 Supported, 16 x 64-bit registers.

0010 Supported, 32 x 64-bit registers.

All other values are reserved.

In ARMv8-A the permitted values are 0000 and 0010.

If this field is nonzero:

- All VFP LDC, STC, MCR, and MRC instructions are supported.
- If the MCRR and MRRC instructions are supported then the corresponding VFP instructions are supported.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the MVFR0:

To access the MVFR0:

```
VMRS <Rt>, MVFR0 ; Read MVFR0 into Rt
```

Register access is encoded as follows:

| spec_reg |
|---|
| 0111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.106 MVFR1, Media and VFP Feature Register 1

The MVFR1 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | Config-RO | Config-RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | Config-RO | Config-RO |

Must be interpreted with MVFR0 and MVFR2.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CPTR_EL2.TFP==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If CPTR_EL3.TFP==1, EL1 and EL2 read accesses to this register are trapped to EL3.

If HCPTR.TCP10==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If CPACR.cp10==00, PL1 read accesses to this register are UNDEFINED.

If HCPTR.TCP10==1, Non-secure EL2 read accesses to this register are UNDEFINED.

If NSACR.cp10==0, Non-secure EL1 and EL2 read accesses to this register are UNDEFINED.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register MVFR1 is architecturally mapped to AArch64 System register MVFR1_EL1.

Implemented only if the implementation includes floating-point and Advanced SIMD instructions.

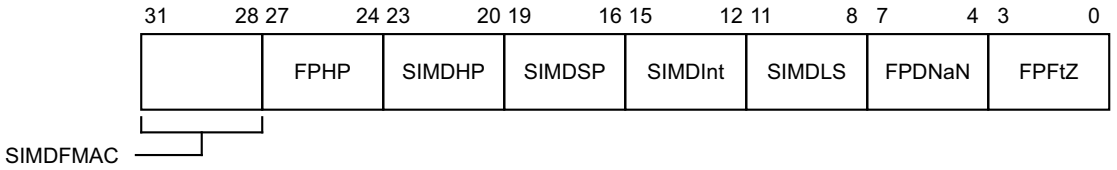
**Attributes**

MVFR1 is a 32-bit register.

#### Field descriptions

The MVFR1 bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**SIMDFMAC, bits [31:28]**

Advanced SIMD Fused Multiply-Accumulate. Indicates whether the floating-point and Advanced SIMD implementation includes the fused multiply accumulate instructions. Defined values are:

`0000` Not implemented.

`0001` Implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

The floating-point and Advanced SIMD implementations must provide the same level of support for these instructions.

**FPHP, bits [27:24]**

Floating Point Half Precision. Indicates whether the floating-point implementation provides half-precision floating-point conversion instructions. Defined values are:

`0000` Not implemented.

`0001` Instructions to convert between half-precision and single-precision implemented.

`0010` As for `0b0001`, and also instructions to convert between half-precision and double-precision implemented.

All other values are reserved.

In ARMv8-A the only permitted value is `0010`.

**SIMDHP, bits [23:20]**

Advanced SIMD Half Precision. Indicates whether the floating-point and Advanced SIMD implementation provides half-precision floating-point conversion instructions. Defined values are:

`0000` Not implemented.

`0001` Implemented. This value is permitted only if the SIMDSP field is `0001`.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**SIMDSP, bits [19:16]**

Advanced SIMD Single Precision. Indicates whether the floating-point and Advanced SIMD implementation provides single-precision floating-point instructions. Defined values are:

`0000` Not implemented.

`0001` Implemented. This value is permitted only if the SIMDInt field is `0001`.

All other values are reserved.

In ARMv8-A the only permitted value is `0001`.

**SIMDInt, bits [15:12]**

Advanced SIMD Integer. Indicates whether the floating-point and Advanced SIMD implementation provides integer instructions. Defined values are:

`0000` Not implemented.

`0001` Implemented.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In ARMv8-A the only permitted value is 0001.

**SIMDLS, bits [11:8]**

Advanced SIMD Load/Store. Indicates whether the floating-point and Advanced SIMD implementation provides load/store instructions. Defined values are:

0000 Not implemented.

0001 Implemented.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**FPDNaN, bits [7:4]**

Default NaN mode. Indicates whether the VFP hardware implementation supports only the Default NaN mode. Defined values are:

0000 Hardware supports only the Default NaN mode.

0001 Hardware supports propagation of NaN values.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

**FPFtZ, bits [3:0]**

Flush to Zero mode. Indicates whether the VFP hardware implementation supports only the Flush-to-Zero mode of operation. Defined values are:

0000 Hardware supports only the Flush-to-Zero mode of operation.

0001 Hardware supports full denormalized number arithmetic.

All other values are reserved.

In ARMv8-A the only permitted value is 0001.

## Accessing the MVFR1:

To access the MVFR1:

```
VMRS <Rt>, MVFR1 ; Read MVFR1 into Rt
```

Register access is encoded as follows:

| spec_reg |
|---|
| 0110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.107 MVFR2, Media and VFP Feature Register 2

The MVFR2 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | Config-RO | Config-RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | Config-RO | Config-RO |

Must be interpreted with MVFR0 and MVFR1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CPTR_EL2.TFP==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If CPTR_EL3.TFP==1, EL1 and EL2 read accesses to this register are trapped to EL3.

If HCPTR.TCP10==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR.TID3==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID3==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If CPACR.cp10==00, PL1 read accesses to this register are UNDEFINED.

If HCPTR.TCP10==1, Non-secure EL2 read accesses to this register are UNDEFINED.

If NSACR.cp10==0, Non-secure EL1 and EL2 read accesses to this register are UNDEFINED.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register MVFR2 is architecturally mapped to AArch64 System register MVFR2_EL1.

Implemented only if the implementation includes floating-point and Advanced SIMD instructions.

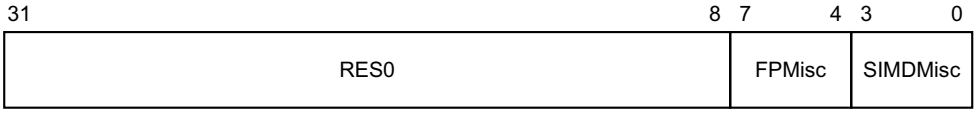
**Attributes**

MVFR2 is a 32-bit register.

#### Field descriptions

The MVFR2 bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:8]**

Reserved, RES0.

**FPMisc, bits [7:4]**

Indicates support for miscellaneous VFP features.

`0000` No support for miscellaneous features.

`0001` Support for Floating-point selection.

`0010` As `0001`, and Floating-point Conversion to Integer with Directed Rounding modes.

`0011` As `0010`, and Floating-point Round to Integral Floating-point.

`0100` As `0011`, and Floating-point MaxNum and MinNum.

All other values are reserved.

In ARMv8-A the only permitted value is `0100`.

**SIMDMisc, bits [3:0]**

Indicates support for miscellaneous Advanced SIMD features.

`0000` No support for miscellaneous features.

`0001` Floating-point Conversion to Integer with Directed Rounding modes.

`0010` As `0001`, and Floating-point Round to Integral Floating-point.

`0011` As `0010`, and Floating-point MaxNum and MinNum.

All other values are reserved.

In ARMv8-A the only permitted value is `0011`.

### Accessing the MVFR2:

To access the MVFR2:

```
VMRS <Rt>, MVFR2 ; Read MVFR2 into Rt
```

Register access is encoded as follows:

| spec_reg |
|---|
| 0101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.108 NMRR, Normal Memory Remap Register

The NMRR characteristics are:

**Purpose**

Provides additional mapping controls for memory regions that are mapped as Normal memory by their entry in the PRRR.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

NMRR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

NMRR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

NMRR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

Used in conjunction with the PRRR.

MAIR1 and NMRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in NMRR.
- When it is set to 1, the register is as described in MAIR1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T10==1, Non-secure EL1 accesses to this register are trapped to EL2.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Configurations**

AArch32 System register NMRR is architecturally mapped to AArch64 System register MAIR_EL1[63:32].

MAIR1 and NMRR are the same register.

When EL3 is using AArch32, write access to NMRR(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

NMRR is a 32-bit register.

## Field descriptions

The NMRR bit assignments are:

***When TTBCR.EAE==0:***

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OR7 | | | | OR6 | | | | OR5 | | | | OR4 | | | | OR3 | | | | OR2 | | | | OR1 | | | | OR0 | | | |
| IR7 | | | | IR6 | | | | IR5 | | | | IR4 | | | | IR3 | | | | IR2 | | | | IR1 | | | | IR0 | | | |

**OR<n>, bits [2n+17:2n+16], for n = 0 to 7**

Outer Cacheable property mapping for memory attributes n, if the region is mapped as Normal memory by the PRRR.TR<n> entry. n is the value of the TEX[0], C, and B bits concatenated. The possible values of this field are:

`00` Region is Non-cacheable.

`01` Region is Write-Back, Write-Allocate.

`10` Region is Write-Through, no Write-Allocate.

`11` Region is Write-Back, no Write-Allocate.

The meaning of the field with n = 6 is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination {TEX[0] = 1, C = 1, B = 0} is IMPLEMENTATION DEFINED.

**IR<n>, bits [2n+1:2n], for n = 0 to 7**

Inner Cacheable property mapping for memory attributes n, if the region is mapped as Normal memory by the PRRR.TR<n> entry. n is the value of the TEX[0], C, and B bits concatenated. The possible values of this field are:

`00` Region is Non-cacheable.

`01` Region is Write-Back, Write-Allocate.

`10` Region is Write-Through, no Write-Allocate.

`11` Region is Write-Back, no Write-Allocate.

The meaning of the field with n = 6 is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination {TEX[0] = 1, C = 1, B = 0} is IMPLEMENTATION DEFINED.

## Accessing the NMRR:

To access the NMRR when TTBCR.EAE==0:

```
MRC p15,0,<Rt>,c10,c2,1 ; Read NMRR into Rt
MCR p15,0,<Rt>,c10,c2,1 ; Write Rt to NMRR
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1010 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.109 NSACR, Non-Secure Access Control Register

The NSACR characteristics are:

**Purpose**

Defines the Non-secure access permissions to Trace, floating-point and Advanced SIMD functionality. Also includes additional IMPLEMENTATION DEFINED bits that can define Non-secure access permissions for IMPLEMENTATION DEFINED functionality.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 (NS) | EL2 (NS) |
|---|---|---|
| - | RO | RO |

For information about the behavior of accesses to the NSACR when EL3 is not implemented or is using AArch64, see Configurations.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

If EL3 is not implemented, then any read of the NSACR from EL2 using AArch32 or from EL1 using AArch32 returns a value of `0x00000C00`.

If EL3 is implemented and is using AArch64 then:

- Any read of the NSACR from Non-secure EL2 using AArch32 or Non-secure EL1 using AArch32 returns a value of `0x00000C00`.
- Any read or write to NSACR from Secure EL1 using AArch32 is trapped as an exception to EL3.

**Note**

In AArch64, the NSACR functionality is replaced by functionality in CPTR_EL3.

Some or all RW fields of this register have defined reset values. These apply whenever the register is accessible. This means they apply when the PE resets into EL3 using AArch32.

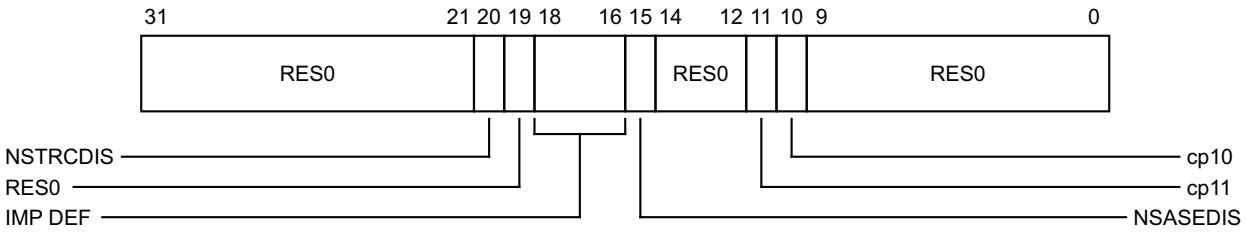
**Attributes**

NSACR is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The NSACR bit assignments are:




**Bits [31:21]**

Reserved, RES0.

**NSTRCDIS, bit [20]**

Disables Non-secure System register accesses to all implemented trace registers.

`0` This control has no effect on:

- System register access to implemented trace registers.
- The behavior of CPACR.TRCDIS and HCPTR.TTA.

`1` Non-secure System register accesses to all implemented trace registers are disabled, meaning:

- CPACR.TRCDIS behaves as RAO/WI in Non-secure state, regardless of its actual value.
- HCPTR.TTA behaves as RAO/WI, regardless of its actual value.

The implementation of this field must correspond to the implementation of the CPACR.TRCDIS field:

- If CPACR.TRCDIS is RAZ/WI, this field is RAZ/WI.
- If CPACR.TRCDIS is RW, this field is RW.

**Note**

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the implementation includes an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED.
- The architecture does not provide Non-secure access controls on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**Bit [19]**

Reserved, RES0.

**IMPLEMENTATION DEFINED, bits [18:16]**

IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**NSASEDIS, bit [15]**

Disables Non-secure access to the Advanced SIMD functionality.

`0` This control has no effect on:

- Non-secure access to Advanced SIMD functionality.
- The behavior of CPACR.ASEDIS and HCPTR.TASE.

`1` Non-secure access to the Advanced SIMD functionality is disabled, meaning:

- CPACR.ASEDIS behaves as RAO/WI in Non-secure state, regardless of its actual value.
- HCPTR.TASE behaves as RAO/WI, regardless of its actual value.

The implementation of this field must correspond to the implementation of the CPACR.ASEDIS field:

- If CPACR.ASEDIS is RES0, this field is RES0. If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.
- If CPACR.ASEDIS is RAZ/WI, this field is RAZ/WI.
- If CPACR.ASEDIS is RW, this field is RW.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**Bits [14:12]**

Reserved, RES0.

**cp11, bit [11]**

The value of this field is ignored. If this field is programmed with a different value to the cp10 field then this field is UNKNOWN on a direct read of the NSACR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to an architecturally UNKNOWN value.

**cp10, bit [10]**

Enable Non-secure access to the floating-point and Advanced SIMD features. Possible values of the fields are:

`0` Floating-point and Advanced SIMD features can be accessed only from Secure state. Any attempt to access this functionality from Non-secure state is UNDEFINED. If the PE is in Non-secure state, the CPACR.{cp11, cp10} fields ignore writes and read as `0b00`, access denied.

`1` Floating-point and Advanced SIMD features can be accessed from both Security states.

If Non-secure access to the floating-point and Advanced SIMD functionality is enabled, the CPACR must be checked to determine the level of access that is permitted.

The floating-point and Advanced SIMD features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to an architecturally UNKNOWN value.

**Bits [9:0]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the NSACR:

To access the NSACR:

```
MRC p15,0,<Rt>,c1,c1,2 ; Read NSACR into Rt
MCR p15,0,<Rt>,c1,c1,2 ; Write Rt to NSACR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0001 | 0001 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.110 PAR, Physical Address Register

The PAR characteristics are:

**Purpose**

Returns the output address (OA) from an address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

PAR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

PAR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

PAR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

On a successful conversion, the PAR can return a value that indicates the resulting attributes, rather than the values that appear in the translation table descriptors. More precisely:

- Memory attribute fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the translation table descriptors. This applies:
  - For a result returned to a 32-bit PAR, to the NOS, SH, Inner, and Outer fields.
  - For a result returned to a 64-bit PAR, to the ATTR and SH fields.
- See the NS bit description for constraints on the value it returns.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T7==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T7==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register PAR is architecturally mapped to AArch64 System register PAR_EL1.

The PAR returns a 32-bit value:

- When the PE is not in Hyp mode and is using the Short-descriptor translation table format.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- When the PE is in Hyp mode for the ATS12NSOxx instructions when the value of HCR.VM is 0 and the value of TTBCR.EAE is 0.

In these cases, PAR[63:32] is RES0.

Otherwise, the PAR returns a 64-bit value. This means it returns a 64-bit value in the following cases:

- When using the Long-descriptor translation table format.
- If the stage 1 address translation is disabled and TTBCR.EAE is set to 1.
- In an implementation that includes EL2, for the result of an ATS1Cxx instruction performed from Hyp mode.

For PL1&0 stage 1 translations, TTBCR.EAE selects the translation table format.

RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits[31:0] and do not modify bits[63:32].

The Configurations section specifies the cases where each PAR format is used.

## Field descriptions

The PAR bit assignments are:

***For all register layouts:***

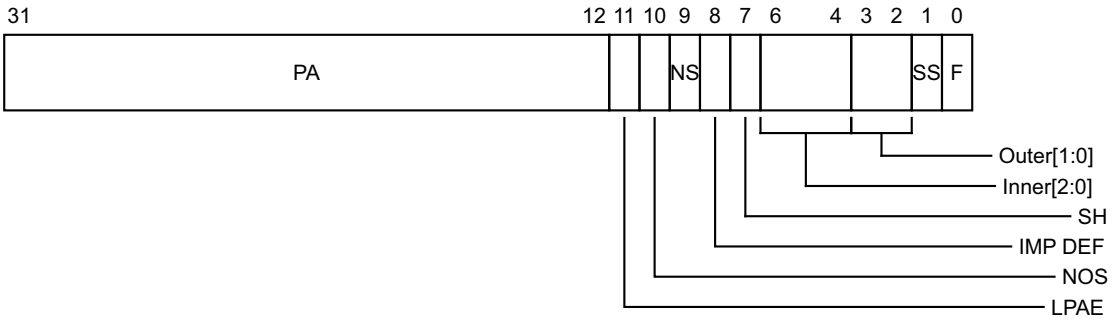
**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

`0` Address translation completed successfully.

`1` Address translation aborted.

***When the instruction returned a 32-bit value to the PAR, PAR.F==0:***




This section describes the register value returned by the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

**PA, bits [31:12]**

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[31:12].

**LPAE, bit [11]**

When updating the PAR with the result of the translation operation, this bit is set as follows:

`0` Short-descriptor translation table format used. This means the PAR returned a 32-bit value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**NOS, bit [10]**

Not Outer Shareable. When the returned value of PAR.SH is 1, indicates the Shareability attribute for the physical memory region:

0 Memory region is Outer Shareable.

1 Memory region is Inner Shareable.

When the returned value of PAR.SH is 0 the value returned to this field is UNKNOWN.

See the Usage constraints for more information about the value returned.

**NS, bit [9]**

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

**IMP DEF, bit [8]**

IMPLEMENTATION DEFINED.

**SH, bit [7]**

Shareability. Indicates whether the physical memory region is Non-shareable:

0 Memory is Non-shareable.

1 Memory is shareable, and PAR.NOS indicates whether the region is Outer Shareable or Inner Shareable.

See the Usage constraints for more information about the value returned.

**Inner[2:0], bits [6:4]**

Inner cacheability attribute for the region. Permitted values are:

000 Non-cacheable.

001 Device-nGnRnE.

011 Device-nGnRE.

101 Write-Back, Write-Allocate.

110 Write-Through.

111 Write-Back, no Write-Allocate.

The values 010 and 100 are reserved.

See the Usage constraints for more information about the value returned.

**Outer[1:0], bits [3:2]**

Outer cacheability attribute for the region. Permitted values are:

00 Non-cacheable.

01 Write-Back, Write-Allocate.

10 Write-Through, no Write-Allocate.

11 Write-Back, no Write-Allocate.

See the Usage constraints for more information about the value returned.

**SS, bit [1]**

Supersection. Used to indicate if the result is a Supersection:

0 Result is not a Supersection. PAR[31:12] contains OA[31:12].

1 Result is a Supersection, and:

- PAR[31:24] contains OA[31:24].


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- PAR[23:16] contains OA[39:32].
- PAR[15:12] contains 0b0000.

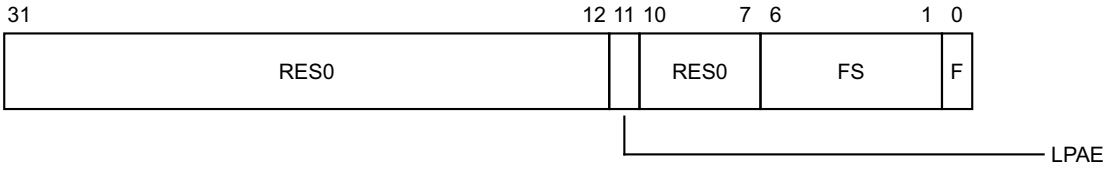
If an implementation supports less than 40 bits of physical address, the bits in the PAR field that correspond to physical address bits that are not implemented are UNKNOWN.

**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

0 Address translation completed successfully.

***When the instruction returned a 32-bit value to the PAR, PAR.F==1:***




This section describes the register value returned by the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

**Bits [31:12]**

Reserved, RES0.

**LPAE, bit [11]**

When updating the PAR with the result of the translation operation, this bit is set as follows:

0 Short-descriptor translation table format used. This means the PAR returned a 32-bit value.

**Bits [10:7]**

Reserved, RES0.

**FS, bits [6:1]**

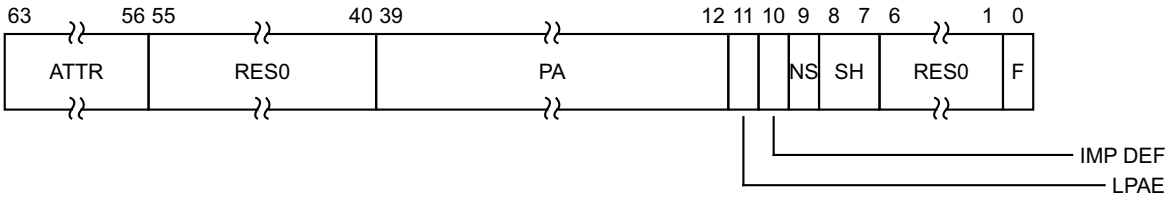
Fault status bits. Bits [12,10,3:0] from the DFSR, indicating the source of the abort.

**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

1 Address translation aborted.

***When the instruction returned a 64-bit value to the PAR, PAR.F==0:***




This section describes the register value returned by the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

**ATTR, bits [63:56]**

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR0 and MAIR1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See the Usage constraints for more information about the value returned.

**Bits [55:40]**

Reserved, RES0.

**PA, bits [39:12]**

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[39:12].

**LPAE, bit [11]**

When updating the PAR with the result of the translation operation, this bit is set as follows:

`1` Long-descriptor translation table format used. This means the PAR returned a 64-bit value.

**IMP DEF, bit [10]**

IMPLEMENTATION DEFINED.

**NS, bit [9]**

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

**SH, bits [8:7]**

Shareability attribute, for the returned output address. Permitted values are:

`00` Non-shareable.

`10` Outer Shareable.

`11` Inner Shareable.

The value `01` is reserved.

**Note**

This field returns the value `10` for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

See the Usage constraints for more information about the value returned.

**Bits [6:1]**

Reserved, RES0.

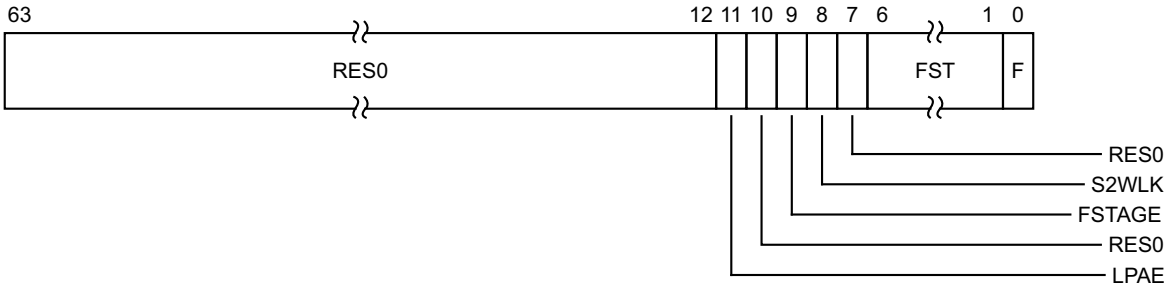
**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

`0` Address translation completed successfully.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

***When the instruction returned a 64-bit value to the PAR, PAR.F==1:***




This section describes the register value returned by the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

**Bits [63:12]**

Reserved, RES0.

**LPAE, bit [11]**

When updating the PAR with the result of the translation operation, this bit is set as follows:

1 Long-descriptor translation table format used. This means the PAR returned a 64-bit value.

**Bit [10]**

Reserved, RES0.

**FSTAGE, bit [9]**

Indicates the translation stage at which the translation aborted:

0 Translation aborted because of a fault in the stage 1 translation.

1 Translation aborted because of a fault in the stage 2 translation.

**S2WLK, bit [8]**

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

**Bit [7]**

Reserved, RES0.

**FST, bits [6:1]**

Fault status field. Values are as in the DFSR.STATUS and IFSR.STATUS fields when using the Long-descriptor translation table format.

**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

1 Address translation aborted.

## Accessing the PAR:

To access the PAR when accessing as a 32-bit register:

```
MRC p15,0,<Rt>,c7,c4,0 ; Read PAR[31:0] into Rt
MCR p15,0,<Rt>,c7,c4,0 ; Write Rt to PAR[31:0]. PAR[63:32] are unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0111 | 0100 | 000 |

To access the PAR when accessing as a 64-bit register:

```
MRRC p15,0,<Rt>,<Rt2>,c7 ; Read PAR[31:0] into Rt and PAR[63:32] into Rt2
MCRR p15,0,<Rt>,<Rt2>,c7 ; Write Rt to PAR[31:0] and Rt2 to PAR[63:32]
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0000 | 0111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.111 PRRR, Primary Region Remap Register

The PRRR characteristics are:

**Purpose**

Controls the top level mapping of the TEX[0], C, and B memory region attributes.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

PRRR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

PRRR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

PRRR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

MAIR0 and PRRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in PRRR.
- When it is set to 1, the register is as described in MAIR0.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T10==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T10==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register PRRR is architecturally mapped to AArch64 System register MAIR_EL1[31:0].

MAIR0 and PRRR are the same register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When EL3 is using AArch32, write access to PRRR(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

RW fields in this register reset to architecturally UNKNOWN values.

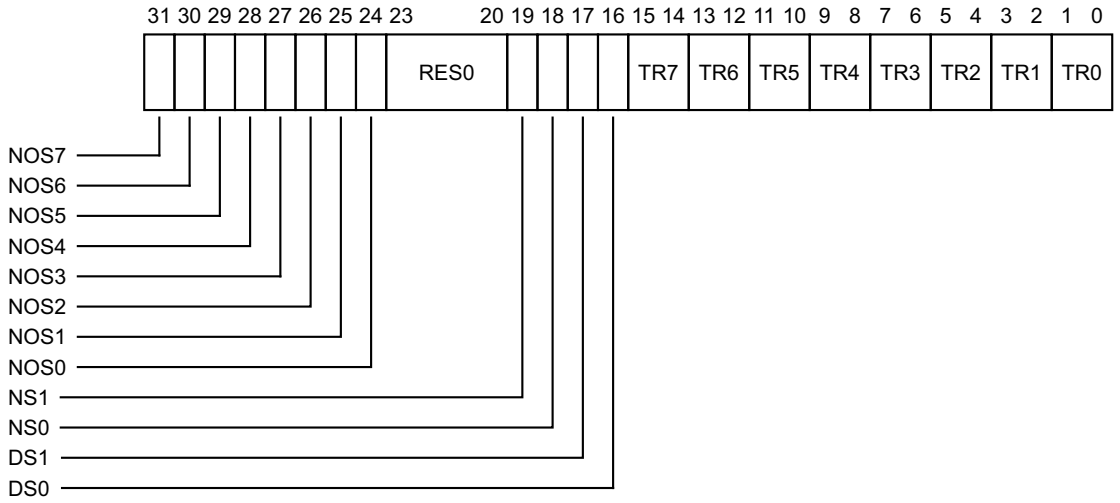
**Attributes**

PRRR is a 32-bit register.

## Field descriptions

The PRRR bit assignments are:

***When TTBCR.EAE==0:***




**NOS<n>, bit [n+24], for n = 0 to 7**

Not Outer Shareable. NOS<n> is the Outer Shareable property for memory attributes n, if the region is mapped as Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, and the appropriate PRRR.{NS0, NS1} field identifies the region as shareable. n is the value of the concatenation of the {TEX[0], C, B} bits from the translation table descriptor. The possible values of each NOS<n> field other than NOS6 are:

`0` Memory region is Outer Shareable.

`1` Memory region is Inner Shareable.

The value of this bit is ignored if the region is:

- Device memory
- Normal memory that is at least one of:
  - — Inner Non-cacheable, Outer Non-cacheable.
  - — Identified by the appropriate PRRR.{NS0, NS1} field as Non-shareable.

The meaning of the NOS6 field is IMPLEMENTATION DEFINED.

**Bits [23:20]**

Reserved, RES0.

**NS1, bit [19]**

Mapping of S = 1 attribute for Normal memory regions. This field is used in determining the Shareability of a memory region that is mapped to Normal memory and both:

- Is not Inner Non-cacheable, Outer Non-cacheable.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Has the S bit in the translation table descriptor set to 1.

The possible values of this bit are:

0 Region is Non-shareable.

1 Region is shareable. The value of the appropriate PRRR.NOS<n> field determines whether the region is Inner Shareable or Outer Shareable.

**NS0, bit [18]**

Mapping of S = 0 attribute for Normal memory regions. This field is used in determining the Shareability of a memory region that is mapped to Normal memory and both:

- Is not Inner Non-cacheable, Outer Non-cacheable.
- Has the S bit in the translation table descriptor set to 0.

The possible values of this bit are:

0 Region is Non-shareable.

1 Region is shareable. The value of the appropriate PRRR.NOS<n> field determines whether the region is Inner Shareable or Outer Shareable.

**DS1, bit [17]**

Mapping of S = 1 attribute for Device memory. In ARMv8, all types of Device memory are Outer Shareable, and therefore this bit is RES1.

**DS0, bit [16]**

Mapping of S = 0 attribute for Device memory. In ARMv8, all types of Device memory are Outer Shareable, and therefore this bit is RES1.

**TR<n>, bits [2n+1:2n], for n = 0 to 7**

TR<n> is the primary TEX mapping for memory attributes n, and defines the mapped memory type for a region with attributes n. n is the value of the concatenation of the {TEX[0], C, B} bits from the translation table descriptor. The possible values for each field other than TR6 are:

00 Device-nGnRnE memory

01 Device-nGnRE memory

10 Normal memory

The value 11 is reserved. The effect of programming a field to 11 is CONSTRAINED UNPREDICTABLE, see *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

The meaning of the TR6 field is IMPLEMENTATION DEFINED.

## Accessing the PRRR:

To access the PRRR when TTBCR.EAE==0:

```
MRC p15,0,<Rt>,c10,c2,0 ; Read PRRR into Rt
MCR p15,0,<Rt>,c10,c2,0 ; Write Rt to PRRR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1010 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.112 REVIDR, Revision ID Register

The REVIDR characteristics are:

**Purpose**

Provides implementation-specific minor revision information.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID1==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID1==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register REVIDR is architecturally mapped to AArch64 System register REVIDR_EL1.

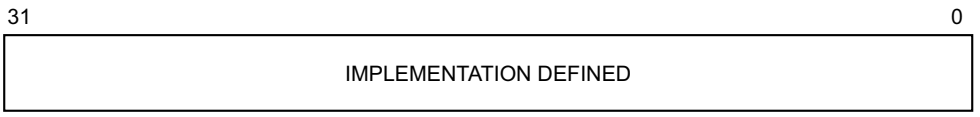
If REVIDR has the same value as MIDR, then its contents have no significance.

**Attributes**

REVIDR is a 32-bit register.

#### Field descriptions

The REVIDR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the REVIDR:

To access the REVIDR:

```
MRC p15,0,<Rt>,c0,c0,6 ; Read REVIDR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0000 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.113 RMR (at EL1), Reset Management Register

The RMR (at EL1) characteristics are:

**Purpose**

If this register's Exception level is the highest implemented Exception level:

- A write to the register can request a Warm reset.
- If the highest implemented Exception level can use AArch32 and AArch64, specifies the Execution state that the PE boots into on a Warm reset.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 |
|---|---|
| - | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register RMR (at EL1) is architecturally mapped to AArch64 System register RMR_EL1.

Only implemented if this register's Exception level is the highest implemented Exception level. In this case:

- If the highest implemented Exception level can use AArch32 and AArch64 then this register must be implemented.
- If the highest implemented Exception level cannot use AArch64 then it is IMPLEMENTATION DEFINED whether the register is implemented.

When this register is not implemented its encoding is UNDEFINED at EL1.

See the field descriptions for the reset values. These apply whenever the register is implemented.

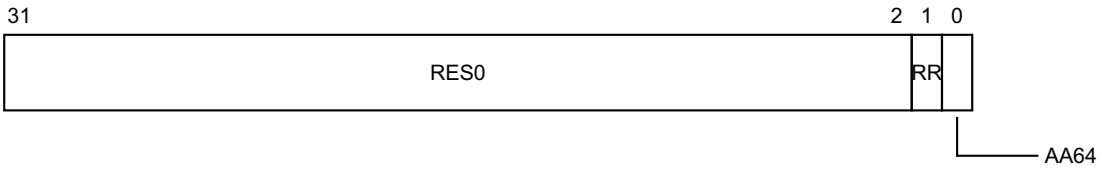
**Attributes**

RMR (at EL1) is a 32-bit register.

#### Field descriptions

The RMR (at EL1) bit assignments are:

***When EL2 and EL3 not implemented:***




**Bits [31:2]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**RR, bit [1]**

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to `0`. This reset value applies on a Cold reset and on a Warm reset.

**AA64, bit [0]**

When the highest implemented Exception level can use AArch32, determines which Execution state the PE boots into after a Warm reset:

`0` AArch32.

`1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If the highest implemented Exception level cannot use AArch64 this bit is RAZ/WI.

If this field is implemented as an RW field, this field resets to `0`. This reset value applies on a Cold reset.

This field resets to its defined reset value on Cold reset.

### Accessing the RMR (at EL1):

To access the RMR (at EL1) when EL2 and EL3 not implemented:

```
MRC p15,0,<Rt>,c12,c0,2 ; Read RMR into Rt
MCR p15,0,<Rt>,c12,c0,2 ; Write Rt to RMR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1100 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.114 RMR (at EL3), Reset Management Register

The RMR (at EL3) characteristics are:

**Purpose**

If this register's Exception level is the highest implemented Exception level:

- A write to the register can request a Warm reset.
- If the highest implemented Exception level can use AArch32 and AArch64, specifies the Execution state that the PE boots into on a Warm reset.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | - |

If EL3 is implemented, the AArch32 view of the RMR register is subject to CP15SDISABLE, which prevents writing to this register when the CP15SDISABLE signal is asserted.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

This register is only accessible in Secure state.

AArch32 System register RMR (at EL3) is architecturally mapped to AArch64 System register RMR_EL3.

Only implemented if this register's Exception level is the highest implemented Exception level. In this case:

- If the highest implemented Exception level can use AArch32 and AArch64 then this register must be implemented.
- If the highest implemented Exception level cannot use AArch64 then it is IMPLEMENTATION DEFINED whether the register is implemented.

When this register is not implemented its encoding is UNDEFINED at EL3.

See the field descriptions for the reset values. These apply whenever the register is implemented.

**Attributes**

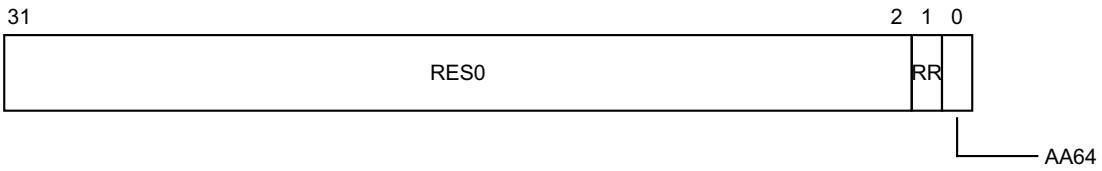
RMR (at EL3) is a 32-bit register.

#### Field descriptions

The RMR (at EL3) bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

***When EL3 implemented:***




**Bits [31:2]**

Reserved, RES0.

**RR, bit [1]**

Reset Request. Setting this bit to 1 requests a Warm reset.

This field resets to `0`. This reset value applies on a Cold reset and on a Warm reset.

**AA64, bit [0]**

When the highest implemented Exception level can use AArch32, determines which Execution state the PE boots into after a Warm reset:

`0` AArch32.

`1` AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If the highest implemented Exception level cannot use AArch64 this bit is RAZ/WI.

If this field is implemented as an RW field, this field resets to `0`. This reset value applies on a Cold reset.

## Accessing the RMR (at EL3):

To access the RMR (at EL3) when EL3 implemented:

```
MRC p15,0,<Rt>,c12,c0,2 ; Read RMR into Rt
MCR p15,0,<Rt>,c12,c0,2 ; Write Rt to RMR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1100 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.115 RVBAR, Reset Vector Base Address Register

The RVBAR characteristics are:

**Purpose**

If EL3 is not implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch32 state.

**Usage constraints**

If EL1 is the highest exception level implemented and is using AArch32, this register is accessible as follows:

| EL0 | EL1 |
|---|---|
| - | RO |

If EL2 is the highest exception level implemented and is using AArch32, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RO |

This register can only be read at the highest Exception level implemented. It is UNDEFINED at all other Exception levels.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T12==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T12==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

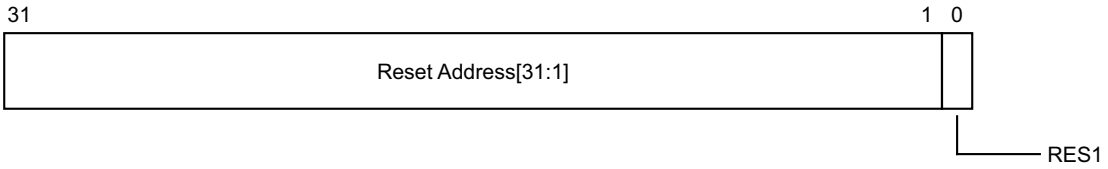
This register is only implemented if the highest Exception level implemented is capable of using AArch32, and is not EL3.

**Attributes**

RVBAR is a 32-bit register.

#### Field descriptions

The RVBAR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:1]**

Reset Address[31:1]. Bits [31:1] of the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 32-bit state.

**Bit [0]**

Reserved, RES1.

### Accessing the RVBAR:

To access the RVBAR:

```
MRC p15,0,<Rt>,c12,c0,1 ; Read RVBAR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1100 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.116 SCR, Secure Configuration Register

The SCR characteristics are:

**Purpose**

When EL3 is implemented and can use AArch32, defines the configuration of the current Security state. It specifies:

- The Security state, either Secure or Non-secure.
- What mode the PE branches to if an IRQ, FIQ, or External Abort occurs.
- Whether the CPSR.F or CPSR.A bits can be modified when SCR.NS==1.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|---|
| - | - | - | RW | - | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | - |

If EL3 is implemented and is using AArch64, any read or write to SCR from Secure EL1 using AArch32 is trapped as an exception to EL3.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

This register is only accessible in Secure state.

AArch32 System register SCR can be mapped to AArch64 System register SCR_EL3, but this is not architecturally mandated.

Some or all RW fields of this register have defined reset values. These apply whenever the register is accessible. This means they apply when the PE resets into EL3 using AArch32.

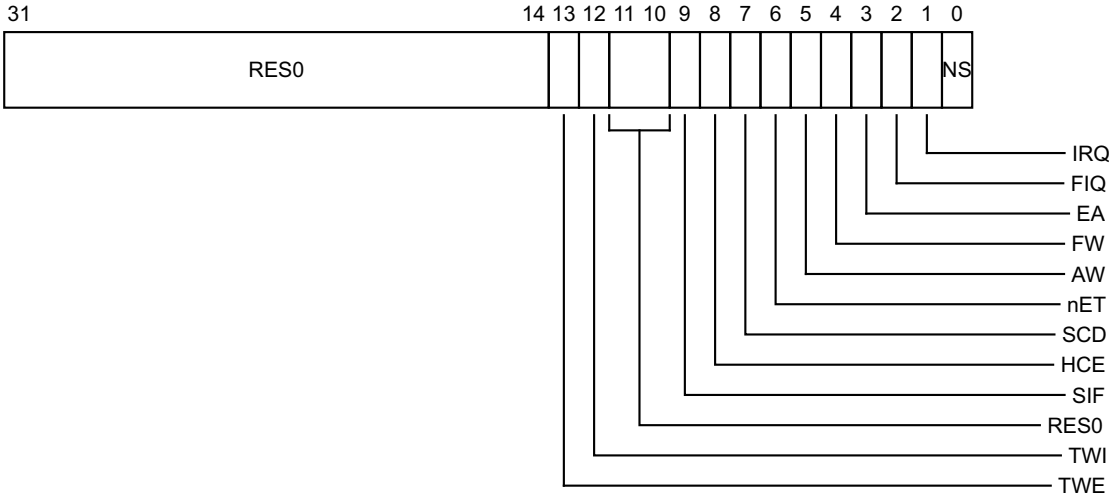
**Attributes**

SCR is a 32-bit register.

#### Field descriptions

The SCR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:14]**

Reserved, RES0.

**TWE, bit [13]**

Traps WFE instructions to Monitor mode.

`0` This control has no effect on the execution of WFE instructions.

`1` Any attempt to execute a WFE instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state and the attempted execution does not generate an exception that is taken to EL1 or EL2.

Any exception that is taken to EL1 or to EL2 has priority over this trap.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to `0`.

**TWI, bit [12]**

Traps WFI instructions to Monitor mode.

`0` This control has no effect on the execution of WFI instructions.

`1` Any attempt to execute a WFI instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state and the attempted execution does not generate an exception that is taken to EL1 or EL2.

Any exception that is taken to EL1 or to EL2 has priority over this trap.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to `0`.

**Bits [11:10]**

Reserved, RES0.

**SIF, bit [9]**

Secure instruction fetch. When the PE is in Secure state, this bit disables instruction fetch from Non-secure memory. The possible values for this bit are:

`0` Secure state instruction fetches from Non-secure memory are permitted.

`1` Secure state instruction fetches from Non-secure memory are not permitted.

This bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to `0`.

**HCE, bit [8]**

Hypervisor Call instruction enable. Enables EL2 and Non-secure EL1 execution of HVC instructions.

`0` HVC instructions are:

- UNDEFINED at Non-secure EL1. The Undefined Instruction exception is taken from PL1 to PL1.
- UNPREDICTABLE at EL2. Behavior is one of the following:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.

`1` HVC instructions are enabled at EL2 and Non-secure EL1.

**Note**

HVC instructions are always UNDEFINED at EL0 and in Secure state.

If EL2 is not implemented, this bit is RES0 and HVC is UNDEFINED.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**SCD, bit [7]**

Secure Monitor Call disable. Disables SMC instructions.

`0` SMC instructions are enabled.

`1` In Non-secure state, SMC instructions are UNDEFINED. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

In Secure state, behavior is one of the following:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.

**Note**

SMC instructions are always UNDEFINED at PL0.

When this register has an architecturally-defined reset value, this field resets to `0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**nET, bit [6]**

Not Early Termination. This bit disables early termination. The possible values of this bit are:

`0` Early termination permitted. Execution time of data operations can depend on the data values.

`1` Disable early termination. The number of cycles required for data operations is forced to be independent of the data values.

This IMPLEMENTATION DEFINED mechanism can disable data dependent timing optimizations from multiplies and data operations. It can provide system support against information leakage that might be exploited by timing correlation types of attack.

On implementations that do not support early termination or do not support disabling early termination, this bit is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**AW, bit [5]**

When the value of SCR.EA is 1 and the value of HCR.AMO is 0, this bit controls whether CPSR.A masks an external abort taken from Non-secure state, and the possible values of this bit are:

`0` External aborts taken from Non-secure state are not masked by CPSR.A, and are taken to EL3.

External aborts taken from Secure state are masked by CPSR.A.

`1` External aborts taken from either Security state are masked by CPSR.A. When CPSR.A is 0, the abort is taken to EL3.

When SCR.EA is 0 or HCR.AMO is 1, this bit has no effect.

When this register has an architecturally-defined reset value, this field resets to `0`.

**FW, bit [4]**

When the value of SCR.FIQ is 1 and the value of HCR.FMO is 0, this bit controls whether CPSR.F masks an FIQ interrupt taken from Non-secure state, and the possible values of this bit are:

`0` An FIQ taken from Non-secure state is not masked by CPSR.F, and is taken to EL3.

An FIQ taken from Secure state is masked by CPSR.F.

`1` An FIQ taken from either Security state is masked by CPSR.F. When CPSR.F is 0, the FIQ is taken to EL3.

When SCR.FIQ is 0 or HCR.FMO is 1, this bit has no effect.

When this register has an architecturally-defined reset value, this field resets to `0`.

**EA, bit [3]**

External Abort handler. This bit controls which mode takes external aborts. The possible values of this bit are:

`0` External aborts taken to Abort mode.

`1` External aborts taken to Monitor mode.

When this register has an architecturally-defined reset value, this field resets to `0`.

**FIQ, bit [2]**

FIQ handler. This bit controls which mode takes FIQ exceptions. The possible values of this bit are:

`0` FIQs taken to FIQ mode.

`1` FIQs taken to Monitor mode.

When this register has an architecturally-defined reset value, this field resets to `0`.

**IRQ, bit [1]**

IRQ handler. This bit controls which mode takes IRQ exceptions. The possible values of this bit are:

`0` IRQs taken to IRQ mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 IRQs taken to Monitor mode.

When this register has an architecturally-defined reset value, this field resets to 0.

**NS, bit [0]**

Non-secure bit. Except when the PE is in Monitor mode, this bit determines the Security state of the PE:

0 PE is in Secure state.

1 PE is in Non-secure state.

If the HCR.TGE bit is set, an attempt to change from a Secure PL1 mode to a Non-secure EL1 mode by changing the SCR.NS bit from 0 to 1 results in the SCR.NS bit remaining as 0.

When this register has an architecturally-defined reset value, this field resets to 0.

### Accessing the SCR:

To access the SCR:

```
MRC p15,0,<Rt>,c1,c1,0 ; Read SCR into Rt
MCR p15,0,<Rt>,c1,c1,0 ; Write Rt to SCR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0001 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.117 SCTLR, System Control Register

The SCTLR characteristics are:

**Purpose**

Provides the top level control of the system, including its memory system.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

SCTLR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

SCTLR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

SCTLR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

Some bits in the register are read-only. These bits relate to non-configurable features of an implementation, and are provided for compatibility with previous versions of the architecture.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register SCTLR is architecturally mapped to AArch64 System register SCTLR_EL1.

When EL3 is using AArch32, write access to SCTLR(S) is disabled when the CP15SDISABLE signal is asserted HIGH.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

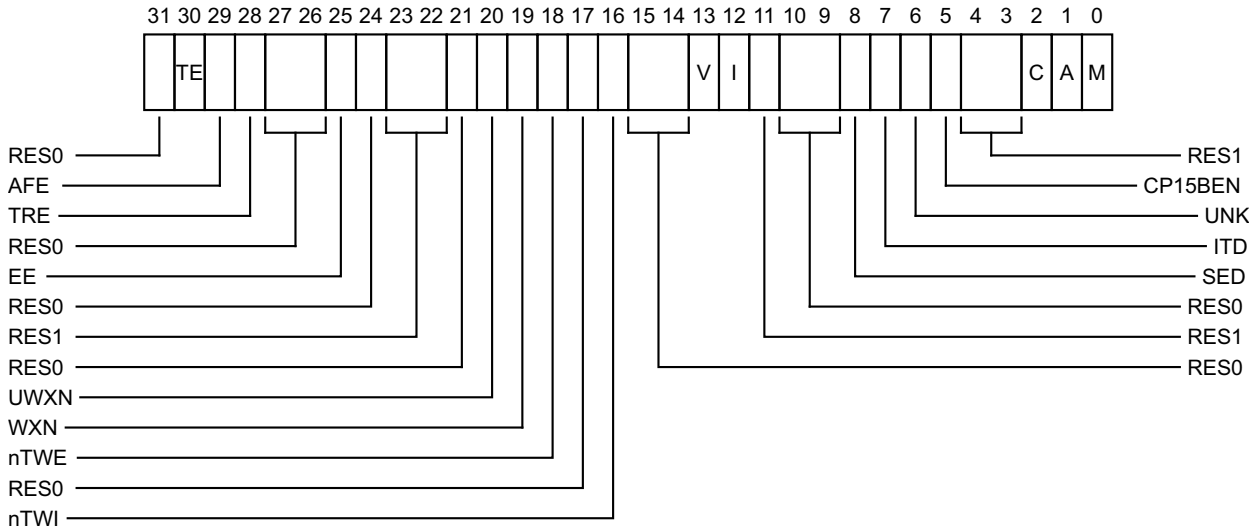
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. If the PE resets into EL3 using AArch32 they apply only to the Secure instance of the register. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

SCTLR is a 32-bit register.

## Field descriptions

The SCTLR bit assignments are:




**Bit [31]**

Reserved, RES0.

**TE, bit [30]**

T32 Exception Enable. This bit controls whether exceptions to an Exception Level that is executing at PL1 are taken to A32 or T32 state:

0 Exceptions, including reset, taken to A32 state.

1 Exceptions, including reset, taken to T32 state.

When this register has an architecturally-defined reset value, this field resets to an IMPLEMENTATION DEFINED choice between:

- 0.
- A value determined by an input configuration signal.

**AFE, bit [29]**

Access Flag Enable. When using the Short-descriptor translation table format for the PL1&0 translation regime, this bit enables use of the AP[0] bit in the translation descriptors as the Access flag, and restricts access permissions in the translation descriptors to the simplified model. The possible values of this bit are:

0 In the translation table descriptors, AP[0] is an access permissions bit. The full range of access permissions is supported. No Access flag is implemented.

1 In the translation table descriptors, AP[0] is the Access flag. Only the simplified model for access permissions is supported.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When using the Long-descriptor translation table format, the VMSA behaves as if this bit is set to 1, regardless of the value of this bit.

The AFE bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to `0`.

**TRE, bit [28]**

TEX remap enable. This bit enables remapping of the TEX[2:1] bits in the PL1&0 translation regime for use as two translation table bits that can be managed by the operating system. Enabling this remapping also changes the scheme used to describe the memory region attributes in the VMSA. The possible values of this bit are:

`0` TEX remap disabled. TEX[2:0] are used, with the C and B bits, to describe the memory region attributes.

`1` TEX remap enabled. TEX[2:1] are reassigned for use as bits managed by the operating system. The TEX[0], C, and B bits are used to describe the memory region attributes, with the MMU remap registers.

When the value of TTBCR.EAE is 1, this bit is RES1.

The TRE bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to `0`.

**Bits [27:26]**

Reserved, RES0.

**EE, bit [25]**

The value of the PSTATE.E bit on branch to an exception vector or coming out of reset, and the endianness of stage 1 translation table walks in the PL1&0 translation regime.

The possible values of this bit are:

`0` Little-endian. PSTATE.E is cleared to 0 on taking an exception or coming out of reset. Stage 1 translation table walks in the PL1&0 translation regime are little-endian.

`1` Big-endian. PSTATE.E is cleared to 0 on taking an exception or coming out of reset. Stage 1 translation table walks in the PL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support for data accesses at Exception Levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support for data accesses at Exception Levels higher than EL0, this bit is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to an IMPLEMENTATION DEFINED choice between:

- `0`.
- A value determined by an input configuration signal.

**Bit [24]**

Reserved, RES0.

**Bits [23:22]**

Reserved, RES1.

**Bit [21]**

Reserved, RES0.

**UWXN, bit [20]**

Unprivileged write permission implies PL1 XN (Execute-never). This bit can force all memory regions that are writeable at PL0 to be treated as XN for accesses from software executing at PL1. The possible values of this bit are:

`0` This control has no effect on memory access permissions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 Any region that is writeable at PL0 forced to XN for accesses from software executing at PL1.

The UWXN bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to 0.

**WXN, bit [19]**

Write permission implies XN (Execute-never). For the PL1&0 translation regime, this bit can force all memory regions that are writeable to be treated as XN. The possible values of this bit are:

0 This control has no effect on memory access permissions.

1 Any region that is writeable in the PL1&0 translation regime is forced to XN for accesses from software executing at PL1 or PL0.

The WXN bit is permitted to be cached in a TLB.

When this register has an architecturally-defined reset value, this field resets to 0.

**nTWE, bit [18]**

Traps PL0 execution of WFE instructions to Undefined mode.

0 Any attempt to execute a WFE instruction at PL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

1 This control has no effect on the PL0 execution of WFE instructions.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to 1.

**Bit [17]**

Reserved, RES0.

**nTWI, bit [16]**

Traps PL0 execution of WFI instructions to Undefined mode.

0 Any attempt to execute a WFI instruction at PL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

1 This control has no effect on the PL0 execution of WFI instructions.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When this register has an architecturally-defined reset value, this field resets to 1.

**Bits [15:14]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**V, bit [13]**

Vectors bit. This bit selects the base address of the exception vectors for exceptions taken to a PE mode other than Monitor mode or Hyp mode:

`0` Normal exception vectors. Base address is held in VBAR.

`1` High exception vectors (Hivecs), base address `0xFFFF0000`. This base address cannot be remapped.

When this register has an architecturally-defined reset value, this field resets to an IMPLEMENTATION DEFINED choice between:

- `0`.
- A value determined by an input configuration signal.

**I, bit [12]**

Instruction cache enable. This is a global enable bit for instruction caches:

`0` All instruction access to Normal memory from PL1 and PL0 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR.M is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

`1` All instruction access to Normal memory from PL1 and PL0 can be cached at all levels of instruction and unified cache.

If the value of SCTLR.M is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

Instruction accesses to Normal memory from Non-secure EL1 and Non-secure EL0 are Cacheable regardless of the value of the SCTLR.I bit if either:

- EL2 is using AArch32 and the value of HCR.DC is 1.
- EL2 is using AArch64 and the value of HCR_EL2.DC is 1.

When this register has an architecturally-defined reset value, this field resets to `0`.

**Bit [11]**

Reserved, RES1.

**Bits [10:9]**

Reserved, RES0.

**SED, bit [8]**

SETEND instruction disable. Disables SETEND instructions at PL0 and PL1.

`0` SETEND instruction execution is enabled at PL0 and PL1.

`1` SETEND instructions are UNDEFINED at PL0 and PL1.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**ITD, bit [7]**

IT Disable. Disables some uses of IT instructions at PL1 and PL0.

`0` All IT instruction functionality is enabled at PL1 and PL0.

`1` Any attempt at PL1 or PL0 to execute any of the following is UNDEFINED:

- All encodings of the IT instruction with hw1[3:0]!=1000.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- All encodings of the subsequent instruction with the following values for hw1:

  `11xxxxxxxxxxxxx`
  : All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.

  `1011xxxxxxxxxxx`
  : All instructions in *Miscellaneous 16-bit instructions* on page F3-2471.

  `10100xxxxxxxxxx`
  : ADD Rd, PC, #imm

  `01001xxxxxxxxxx`
  : LDR Rd, [PC, #imm]

  `0100x1xxx1111xxx`
  : ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.

  `010001xx1xxxx111`
  : ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn.

  These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

  It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

  - A 16-bit instruction, that can only be followed by another 16-bit instruction.
  - The first half of a 32-bit instruction.

  This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

  An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

ITD is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAZ/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `0`.

**UNK, bit [6]**

Writes to this bit are IGNORED. Reads of this bit return an UNKNOWN value.

**CP15BEN, bit [5]**

CP15 memory barrier enable. Enables accesses to the CP15 DMB, DSB, and ISB barrier operations from PL1 and PL0:

`0` PL0 and PL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.

`1` PL0 and PL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1. If it is not implemented then this bit is RAO/WI.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to `1`.

**Bits [4:3]**

Reserved, RES1.

**C, bit [2]**

Cache enable, for data caching.

`0` All data access to Normal memory from PL1 and PL0, and all accesses to the PL1&0 stage 1 translation tables, are Non-cacheable for all levels of data and unified cache.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 All data access to Normal memory from PL1 and PL0, and all accesses to the PL1&0 stage 1 translation tables, can be cached at all levels of data and unified cache.

The PE ignores SCLTR.C for Non-secure state and data accesses to Normal memory from EL1 and EL0 are Cacheable if either:

- EL2 is using AArch32 and the value of HCR.DC is 1.
- EL2 is using AArch64 and the value of HCR_EL2.DC is 1.

When this register has an architecturally-defined reset value, this field resets to 0.

**A, bit [1]**

Alignment check enable. This is the enable bit for Alignment fault checking at PL1 and PL0:

0 Alignment fault checking disabled when executing at PL1 or PL0.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

1 Alignment fault checking enabled when executing at PL1 or PL0.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

When this register has an architecturally-defined reset value, this field resets to 0.

**M, bit [0]**

MMU enable for EL1 and EL0 stage 1 address translation. Possible values of this bit are:

0 EL1 and EL0 stage 1 address translation disabled.

See the SCTLR.I field for the behavior of instruction accesses to Normal memory.

1 EL1 and EL0 stage 1 address translation enabled.

In the Non-secure state the PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of the field if either:

- EL2 is using AArch32 and the value of HCR.{DC, TGE} is not {0, 0}.
- EL2 is using AArch64 and the value of HCR_EL2.{DC, TGE} is not {0, 0}.

When this register has an architecturally-defined reset value, this field resets to 0.

## Accessing the SCTLR:

To access the SCTLR:

```
MRC p15,0,<Rt>,c1,c0,0 ; Read SCTLR into Rt
MCR p15,0,<Rt>,c1,c0,0 ; Write Rt to SCTLR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0001 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.118 SPSR, Saved Program Status Register

The SPSR characteristics are:

**Purpose**

Holds the saved process state for the current mode.

**Usage constraints**

The SPSR can be read using the MRS instruction and written using the MSR (immediate) or MSR (register) instructions. For more details on the instruction syntax, see *PSTATE and banked register access instructions* on page F1-2420.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

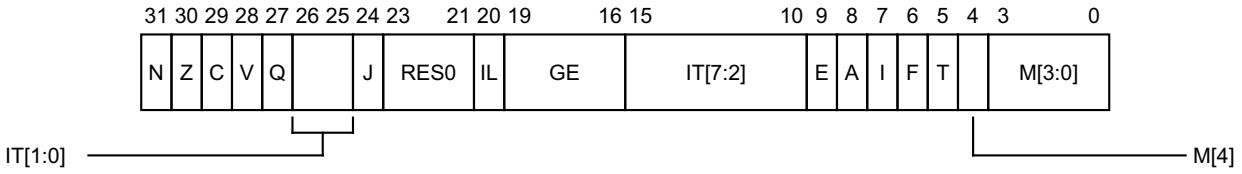
There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

SPSR is a 32-bit register.

### Field descriptions

The SPSR bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to the current mode, and copied to CPSR.N on executing an exception return operation in the current mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to the current mode, and copied to CPSR.Z on executing an exception return operation in the current mode.

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to the current mode, and copied to CPSR.C on executing an exception return operation in the current mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to the current mode, and copied to CPSR.V on executing an exception return operation in the current mode.

**Q, bit [27]**

Set to the value of CPSR.Q on taking an exception to the current mode, and copied to CPSR.Q on executing an exception return operation in the current mode.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

`0` Taken from A32 state.

`1` Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

`1` Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| `0b0000` | User |
| `0b0001` | FIQ |
| `0b0010` | IRQ |
| `0b0011` | Supervisor |
| `0b0110` | Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32) |
| `0b0111` | Abort |
| `0b1010` | Hyp |
| `0b1011` | Undefined |
| `0b1111` | System |

Other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.119 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Abort mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS, !ABT) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

This register is only accessible at EL1 in modes other than Abort mode. In Abort mode, it is accessible as the current SPSR.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

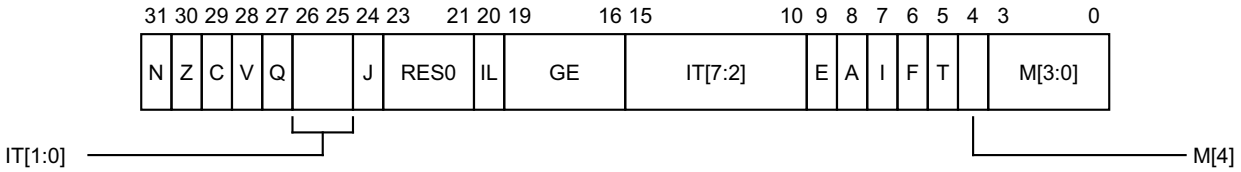
AArch32 System register SPSR_abt is architecturally mapped to AArch64 System register SPSR_abt.

**Attributes**

SPSR_abt is a 32-bit register.

## Field descriptions

The SPSR_abt bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to Abort mode, and copied to CPSR.N on executing an exception return operation in Abort mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to Abort mode, and copied to CPSR.Z on executing an exception return operation in Abort mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to Abort mode, and copied to CPSR.C on executing an exception return operation in Abort mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to Abort mode, and copied to CPSR.V on executing an exception return operation in Abort mode.

**Q, bit [27]**

Set to the value of CPSR.Q on taking an exception to Abort mode, and copied to CPSR.Q on executing an exception return operation in Abort mode.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32) |
| 0b0111 | Abort |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

## Accessing the SPSR_abt:

To access the SPSR_abt:

```
MRS <Rd>, SPSR_abt ; Read SPSR_abt into Rd
MSR SPSR_abt, <Rd> ; Write Rd to SPSR_abt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| m | m1 | R |
|---|---|---|
| 1 | 0100 | 1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.120 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to FIQ mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS, !FIQ) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

This register is only accessible at EL1 in modes other than FIQ mode. In FIQ mode, it is accessible as the current SPSR.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

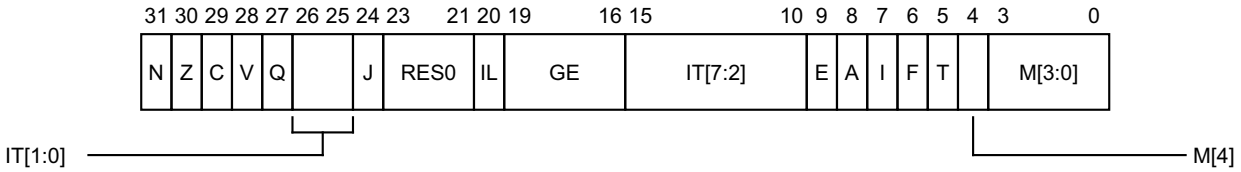
AArch32 System register SPSR_fiq is architecturally mapped to AArch64 System register SPSR_fiq.

**Attributes**

SPSR_fiq is a 32-bit register.

### Field descriptions

The SPSR_fiq bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to FIQ mode, and copied to CPSR.N on executing an exception return operation in FIQ mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to FIQ mode, and copied to CPSR.Z on executing an exception return operation in FIQ mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to FIQ mode, and copied to CPSR.C on executing an exception return operation in FIQ mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to FIQ mode, and copied to CPSR.V on executing an exception return operation in FIQ mode.

**Q, bit [27]**

Set to the value of CPSR.Q on taking an exception to FIQ mode, and copied to CPSR.Q on executing an exception return operation in FIQ mode.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32) |
| 0b0111 | Abort |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

## Accessing the SPSR_fiq:

To access the SPSR_fiq:

```
MRS <Rd>, SPSR_fiq ; Read SPSR_fiq into Rd
MSR SPSR_fiq, <Rd> ; Write Rd to SPSR_fiq
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| m | m1 | R |
|---|---|---|
| 0 | 1110 | 1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.121 SPSR_hyp, Saved Program Status Register (Hyp mode)

The SPSR_hyp characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | - |

In Hyp mode, this register is accessible as the current SPSR.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

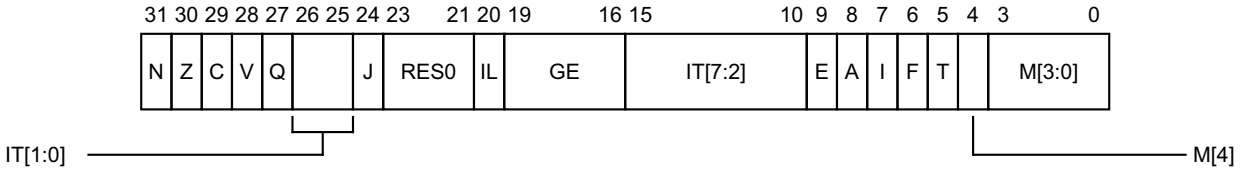
AArch32 System register SPSR_hyp is architecturally mapped to AArch64 System register SPSR_EL2.

**Attributes**

SPSR_hyp is a 32-bit register.

## Field descriptions

The SPSR_hyp bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to Hyp mode, and copied to CPSR.N on executing an exception return operation in Hyp mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to Hyp mode, and copied to CPSR.Z on executing an exception return operation in Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to Hyp mode, and copied to CPSR.C on executing an exception return operation in Hyp mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to Hyp mode, and copied to CPSR.V on executing an exception return operation in Hyp mode.

**Q, bit [27]**

Set to the value of CPSR.Q on taking an exception to Hyp mode, and copied to CPSR.Q on executing an exception return operation in Hyp mode.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0111 | Abort |
| 0b1010 | Hyp |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

## Accessing the SPSR_hyp:

To access the SPSR_hyp:

```
MRS <Rd>, SPSR_hyp ; Read SPSR_hyp into Rd
MSR SPSR_hyp, <Rd> ; Write Rd to SPSR_hyp
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| m | m1 | R |
|---|---|---|
| 1 | 1110 | 1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.122 SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to IRQ mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS, !IRQ) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

This register is only accessible at EL1 in modes other than IRQ mode. In IRQ mode, it is accessible as the current SPSR.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

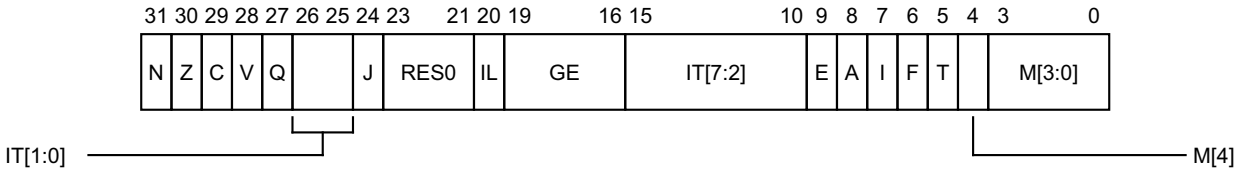
AArch32 System register SPSR_irq is architecturally mapped to AArch64 System register SPSR_irq.

**Attributes**

SPSR_irq is a 32-bit register.

### Field descriptions

The SPSR_irq bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to IRQ mode, and copied to CPSR.N on executing an exception return operation in IRQ mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to IRQ mode, and copied to CPSR.Z on executing an exception return operation in IRQ mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to IRQ mode, and copied to CPSR.C on executing an exception return operation in IRQ mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to IRQ mode, and copied to CPSR.V on executing an exception return operation in IRQ mode.

**Q, bit [27]**

Set to the value of CPSR.Q on taking an exception to IRQ mode, and copied to CPSR.Q on executing an exception return operation in IRQ mode.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32) |
| 0b0111 | Abort |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

## Accessing the SPSR_irq:

To access the SPSR_irq:

```
MRS <Rd>, SPSR_irq ; Read SPSR_irq into Rd
MSR SPSR_irq, <Rd> ; Write Rd to SPSR_irq
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| m | m1 | R |
|---|---|---|
| 1 | 0000 | 1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.123 SPSR_mon, Saved Program Status Register (Monitor mode)

The SPSR_mon characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Monitor mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1, !Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | - |

This register is only accessible at EL3 in modes other than Monitor mode. In Monitor mode, it is accessible as the current SPSR.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

This register is only accessible in Secure state.

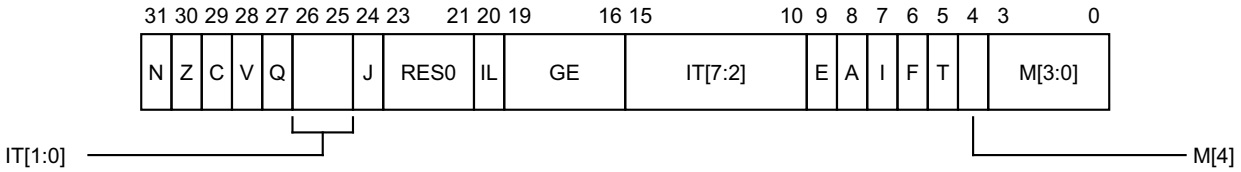
AArch32 System register SPSR_mon can be mapped to AArch64 System register SPSR_EL3, but this is not architecturally mandated.

**Attributes**

SPSR_mon is a 32-bit register.

## Field descriptions

The SPSR_mon bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to Monitor mode, and copied to CPSR.N on executing an exception return operation in Monitor mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to Monitor mode, and copied to CPSR.Z on executing an exception return operation in Monitor mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to Monitor mode, and copied to CPSR.C on executing an exception return operation in Monitor mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to Monitor mode, and copied to CPSR.V on executing an exception return operation in Monitor mode.

**Q, bit [27]**

Set to the value of CPSR.Q on taking an exception to Monitor mode, and copied to CPSR.Q on executing an exception return operation in Monitor mode.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32) |
| 0b0111 | Abort |
| 0b1010 | Hyp |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

### Accessing the SPSR_mon:

To access the SPSR_mon:

```
MRS <Rd>, SPSR_mon ; Read SPSR_mon into Rd
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MSR SPSR_mon, <Rd> ; Write Rd to SPSR_mon
```

Register access is encoded as follows:

| m | m1 | R |
|---|---|---|
| 1 | 1100 | 1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.124 SPSR_svc, Saved Program Status Register (Sup. Call mode)

The SPSR_svc characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Supervisor mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS, !SVC) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

This register is only accessible at EL1 in modes other than Supervisor mode. In Supervisor mode, it is accessible as the current SPSR.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

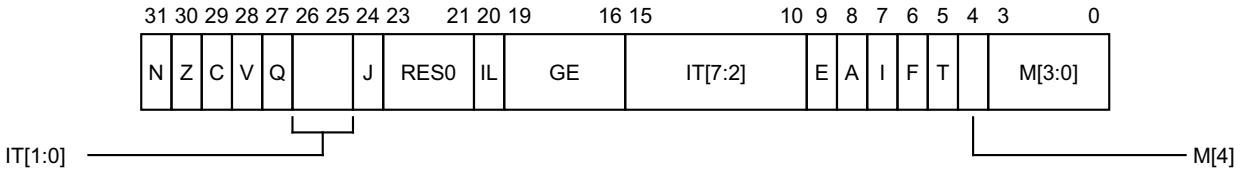
AArch32 System register SPSR_svc is architecturally mapped to AArch64 System register SPSR_EL1.

**Attributes**

SPSR_svc is a 32-bit register.

## Field descriptions

The SPSR_svc bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to Supervisor mode, and copied to CPSR.N on executing an exception return operation in Supervisor mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to Supervisor mode, and copied to CPSR.Z on executing an exception return operation in Supervisor mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to Supervisor mode, and copied to CPSR.C on executing an exception return operation in Supervisor mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to Supervisor mode, and copied to CPSR.V on executing an exception return operation in Supervisor mode.

**Q, bit [27]**

Set to the value of CPSR.Q on taking an exception to Supervisor mode, and copied to CPSR.Q on executing an exception return operation in Supervisor mode.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32) |
| 0b0111 | Abort |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

### Accessing the SPSR_svc:

To access the SPSR_svc:

```
MRS <Rd>, SPSR_svc ; Read SPSR_svc into Rd
MSR SPSR_svc, <Rd> ; Write Rd to SPSR_svc
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| m | m1 | R |
|---|---|---|
| 1 | 0010 | 1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.125 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Undefined mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS, !UND) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

This register is only accessible at EL1 in modes other than Undefined mode. In Undefined mode, it is accessible as the current SPSR.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

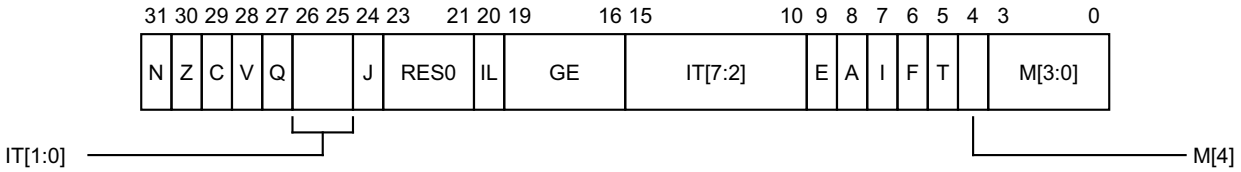
AArch32 System register SPSR_und is architecturally mapped to AArch64 System register SPSR_und.

**Attributes**

SPSR_und is a 32-bit register.

### Field descriptions

The SPSR_und bit assignments are:




**N, bit [31]**

Set to the value of CPSR.N on taking an exception to Undefined mode, and copied to CPSR.N on executing an exception return operation in Undefined mode.

**Z, bit [30]**

Set to the value of CPSR.Z on taking an exception to Undefined mode, and copied to CPSR.Z on executing an exception return operation in Undefined mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**C, bit [29]**

Set to the value of CPSR.C on taking an exception to Undefined mode, and copied to CPSR.C on executing an exception return operation in Undefined mode.

**V, bit [28]**

Set to the value of CPSR.V on taking an exception to Undefined mode, and copied to CPSR.V on executing an exception return operation in Undefined mode.

**Q, bit [27]**

Set to the value of CPSR.Q on taking an exception to Undefined mode, and copied to CPSR.Q on executing an exception return operation in Undefined mode.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:21]**

Reserved, RES0.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before the exception was taken.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that an exception was taken from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32) |
| 0b0111 | Abort |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

### Accessing the SPSR_und:

To access the SPSR_und:

```
MRS <Rd>, SPSR_und ; Read SPSR_und into Rd
MSR SPSR_und, <Rd> ; Write Rd to SPSR_und
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| m | m1 | R |
|---|---|---|
| 1 | 0110 | 1 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.126 TCMTR, TCM Type Register

The TCMTR characteristics are:

**Purpose**

Provides information about the implementation of the TCM.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID1==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID1==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

TCMTR is a 32-bit register.

### Field descriptions

The TCMTR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

### Accessing the TCMTR:

To access the TCMTR:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRC p15,0,<Rt>,c0,c0,2 ; Read TCMTR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.127 TLBIALL, TLB Invalidate All

The TLBIALL characteristics are:

**Purpose**

Invalidate all TLB entries for the PL1&0 translation regime, subject to the Privilege level and Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIALL.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

TLBIALL is a 32-bit system operation.

#### Field descriptions

The TLBIALL operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBIALL operation:

To perform the TLBIALL operation:

```
MCR p15,0,<Rt>,c8,c7,0 ; TLBIALL operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0111 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.128 TLBIALLH, TLB Invalidate All, Hyp mode

The TLBIALLH characteristics are:

**Purpose**

Invalidate all TLB entries for the PL2 translation regime.

For details of the scope of this instruction see TLBIALLH.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0, Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|---|
| - | - | - | WO | WO | WO | WO-UNPREDICTABLE |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If this operation is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE. For more information see *Hyp mode TLB maintenance instructions* on page K1-5499.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

TLBIALLH is a 32-bit system operation.

#### Field descriptions

The TLBIALLH operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBIALLH operation:

To perform the TLBIALLH operation:

```
MCR p15,4,<Rt>,c8,c7,0 ; TLBIALLH operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0111 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.129 TLBIALLHIS, TLB Invalidate All, Hyp mode, Inner Shareable

The TLBIALLHIS characteristics are:

**Purpose**

Invalidate all TLB entries for the PL2 translation regime, on all PEs in the same Inner Shareable domain.

For details of the scope of this instruction see TLBIALLH.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0, Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|---|
| - | - | - | WO | WO | WO | WO-UNPREDICTABLE |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If this operation is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE. For more information see *Hyp mode TLB maintenance instructions* on page K1-5499.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

TLBIALLHIS is a 32-bit system operation.

#### Field descriptions

The TLBIALLHIS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBIALLHIS operation:

To perform the TLBIALLHIS operation:

```
MCR p15,4,<Rt>,c8,c3,0 ; TLBIALLHIS operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.130 TLBIALLIS, TLB Invalidate All, Inner Shareable

The TLBIALLIS characteristics are:

**Purpose**

Invalidate all TLB entries for the PL1&0 translation regime, on all PEs in the same Inner Shareable domain, subject to the Privilege level and Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIALL.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

TLBIALLIS is a 32-bit system operation.

#### Field descriptions

The TLBIALLIS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBIALLIS operation:

To perform the TLBIALLIS operation:

```
MCR p15,0,<Rt>,c8,c3,0 ; TLBIALLIS operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.131 TLBIALLNSNH, TLB Invalidate All, Non-Secure Non-Hyp

The TLBIALLNSNH characteristics are:

**Purpose**

Invalidate all TLB entries for the Non-secure PL1&0 translation regime.

For details of the scope of this instruction see TLBIALLNSNH.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0, Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|---|
| - | - | - | WO | WO | WO | WO-UNPREDICTABLE |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If this operation is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE. For more information see *Hyp mode TLB maintenance instructions* on page K1-5499.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

TLBIALLNSNH is a 32-bit system operation.

#### Field descriptions

The TLBIALLNSNH operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBIALLNSNH operation:

To perform the TLBIALLNSNH operation:

```
MCR p15,4,<Rt>,c8,c7,4 ; TLBIALLNSNH operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0111 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.132 TLBIALLNSNHIS, TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable

The TLBIALLNSNHIS characteristics are:

**Purpose**

Invalidate all TLB entries for the Non-secure PL1&0 translation regime, on all PEs in the same Inner Shareable domain.

For details of the scope of this instruction see TLBIALLNSNH.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0, Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|---|
| - | - | - | WO | WO | WO | WO-UNPREDICTABLE |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If this operation is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE. For more information see *Hyp mode TLB maintenance instructions* on page K1-5499.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

There are no configuration notes.

**Attributes**

TLBIALLNSNHIS is a 32-bit system operation.

#### Field descriptions

The TLBIALLNSNHIS operation ignores the value in the register specified by the instruction used to perform this operation. Software does not have to write a value to the register before issuing this instruction.

#### Performing the TLBIALLNSNHIS operation:

To perform the TLBIALLNSNHIS operation:

```
MCR p15,4,<Rt>,c8,c3,4 ; TLBIALLNSNHIS operation, ignoring the value in Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0011 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.133 TLBIASID, TLB Invalidate by ASID match

The TLBIASID characteristics are:

**Purpose**

Invalidate TLB entries for stage 1 of the PL1&0 translation regime that match the given ASID, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIASID.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

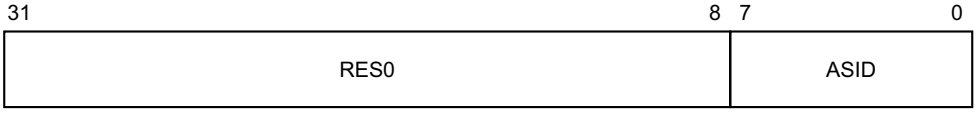
There are no configuration notes.

**Attributes**

TLBIASID is a 32-bit system operation.

#### Field descriptions

The TLBIASID input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this operation.

### Performing the TLBIASID operation:

To perform the TLBIASID operation:

```
MCR p15,0,<Rt>,c8,c7,2 ; TLBIASID operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0111 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.134 TLBIASIDIS, TLB Invalidate by ASID match, Inner Shareable

The TLBIASIDIS characteristics are:

**Purpose**

Invalidate TLB entries for stage 1 of the PL1&0 translation regime that match the given ASID, on all PEs in the same Inner Shareable domain, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIASID.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

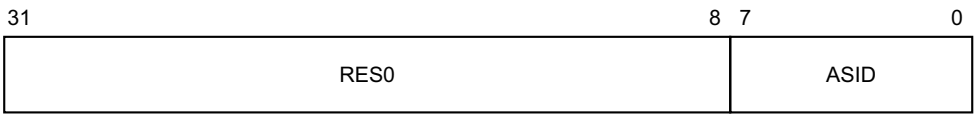
There are no configuration notes.

**Attributes**

TLBIASIDIS is a 32-bit system operation.

#### Field descriptions

The TLBIASIDIS input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this operation.

### Performing the TLBIASIDIS operation:

To perform the TLBIASIDIS operation:

```
MCR p15,0,<Rt>,c8,c3,2 ; TLBIASIDIS operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0011 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.135 TLBIIPAS2, TLB Invalidate by Intermediate Physical Address, Stage 2

The TLBIIPAS2 characteristics are:

**Purpose**

Invalidate TLB entries for stage 2 of the Non-secure PL1&0 translation regime that match the given IPA and the current VMID.

For details of the scope of this instruction see TLBIIPAS2.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

This instruction is a NOP when executed in Monitor mode with SCR.NS==0, and is UNPREDICTABLE when executed in any AArch32 Secure privileged mode other than Monitor mode.

This instruction must apply to structures that contain only stage 2 translation information, but does not need to apply to structures that contain combined stage 1 and stage 2 translation information.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

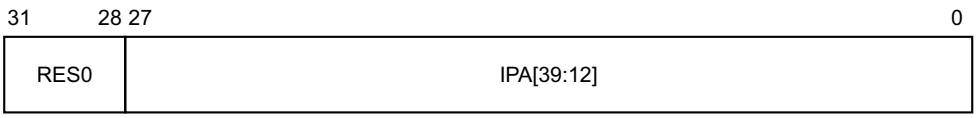
This operation is not implemented in architecture versions before ARMv8.

**Attributes**

TLBIIPAS2 is a 32-bit system operation.

## Field descriptions

The TLBIIPAS2 input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:28]**

Reserved, RES0.

**IPA[39:12], bits [27:0]**

Bits[39:12] of the intermediate physical address to match.

### Performing the TLBIIPAS2 operation:

To perform the TLBIIPAS2 operation:

```
MCR p15,4,<Rt>,c8,c4,1 ; TLBIIPAS2 operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0100 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.136 TLBIIPAS2IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable

The TLBIIPAS2IS characteristics are:

**Purpose**

Invalidate TLB entries for stage 2 of the Non-secure PL1&0 translation regime that match the given IPA and the current VMID, on all PEs in the same Inner Shareable domain.

For details of the scope of this instruction see TLBIIPAS2.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

This instruction is a NOP when executed in Monitor mode with SCR.NS==0, and is UNPREDICTABLE when executed in any AArch32 Secure privileged mode other than Monitor mode.

This instruction must apply to structures that contain only stage 2 translation information, but does not need to apply to structures that contain combined stage 1 and stage 2 translation information.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

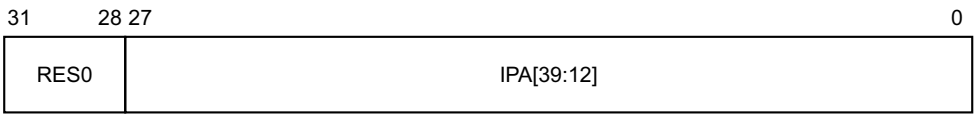
This operation is not implemented in architecture versions before ARMv8.

**Attributes**

TLBIIPAS2IS is a 32-bit system operation.

## Field descriptions

The TLBIIPAS2IS input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:28]**

Reserved, RES0.

**IPA[39:12], bits [27:0]**

Bits[39:12] of the intermediate physical address to match.

### Performing the TLBIIPAS2IS operation:

To perform the TLBIIPAS2IS operation:

```
MCR p15,4,<Rt>,c8,c0,1 ; TLBIIPAS2IS operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0000 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.137 TLBIIPAS2L, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level

The TLBIIPAS2L characteristics are:

**Purpose**

Invalidate TLB entries for stage 2 of the Non-secure PL1&0 translation regime, for the last level of translation, that match the given IPA and the current VMID.

For details of the scope of this instruction see TLBIIPAS2.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

This instruction is a NOP when executed in Monitor mode with SCR.NS==0, and is UNPREDICTABLE when executed in any AArch32 Secure privileged mode other than Monitor mode.

This instruction must apply to structures that contain only stage 2 translation information, but does not need to apply to structures that contain combined stage 1 and stage 2 translation information.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

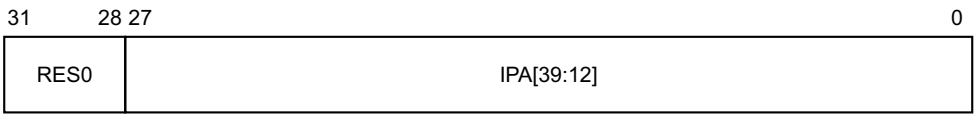
This operation is not implemented in architecture versions before ARMv8.

**Attributes**

TLBIIPAS2L is a 32-bit system operation.

## Field descriptions

The TLBIIPAS2L input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:28]**

Reserved, RES0.

**IPA[39:12], bits [27:0]**

Bits[39:12] of the intermediate physical address to match.

### Performing the TLBIIPAS2L operation:

To perform the TLBIIPAS2L operation:

```
MCR p15,4,<Rt>,c8,c4,5 ; TLBIIPAS2L operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0100 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.138 TLBIIPAS2LIS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable

The TLBIIPAS2LIS characteristics are:

**Purpose**

Invalidate TLB entries for stage 2 of the Non-secure PL1&0 translation regime, for the last level of translation, that match the given IPA and the current VMID, on all PEs in the same Inner Shareable domain.

For details of the scope of this instruction see TLBIIPAS2L.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

This instruction is a NOP when executed in Monitor mode with SCR.NS==0, and is UNPREDICTABLE when executed in any AArch32 Secure privileged mode other than Monitor mode.

This instruction must apply to structures that contain only stage 2 translation information, but does not need to apply to structures that contain combined stage 1 and stage 2 translation information.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

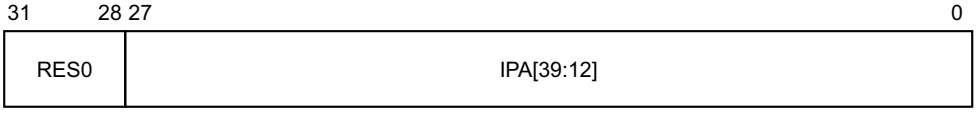
This operation is not implemented in architecture versions before ARMv8.

**Attributes**

TLBIIPAS2LIS is a 32-bit system operation.

## Field descriptions

The TLBIIPAS2LIS input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:28]**

Reserved, RES0.

**IPA[39:12], bits [27:0]**

Bits[39:12] of the intermediate physical address to match.

### Performing the TLBIIPAS2LIS operation:

To perform the TLBIIPAS2LIS operation:

```
MCR p15,4,<Rt>,c8,c0,5 ; TLBIIPAS2LIS operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0000 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.139 TLBIMVA, TLB Invalidate by VA

The TLBIMVA characteristics are:

**Purpose**

Invalidate PL1&0 regime stage 1 and 2 TLB entries for the given VA and ASID, the current VMID, and the current Security state.

For details of the scope of this instruction see TLBIMVA.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

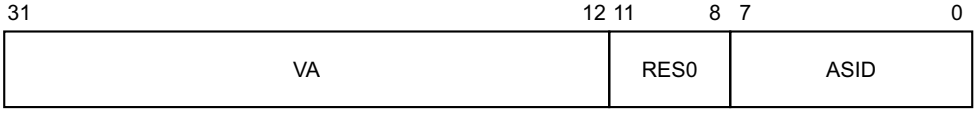
There are no configuration notes.

**Attributes**

TLBIMVA is a 32-bit system operation.

#### Field descriptions

The TLBIMVA input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

**Bits [11:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

### Performing the TLBIMVA operation:

To perform the TLBIMVA operation:

```
MCR p15,0,<Rt>,c8,c7,1 ; TLBIMVA operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0111 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.140 TLBIMVAA, TLB Invalidate by VA, All ASID

The TLBIMVAA characteristics are:

**Purpose**

Invalidate TLB entries for stage 1 of the PL1&0 translation regime, for any ASID, that match the given VA, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIMVAA.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

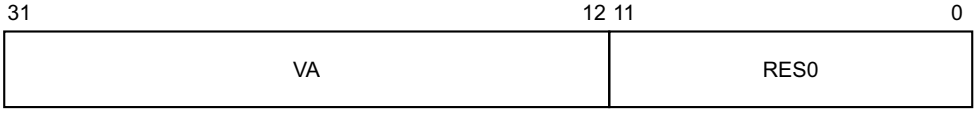
There are no configuration notes.

**Attributes**

TLBIMVAA is a 32-bit system operation.

#### Field descriptions

The TLBIMVAA input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this operation, regardless of the ASID.

**Bits [11:0]**

Reserved, RES0.

### Performing the TLBIMVAA operation:

To perform the TLBIMVAA operation:

```
MCR p15,0,<Rt>,c8,c7,3 ; TLBIMVAA operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0111 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.141 TLBIMVAAIS, TLB Invalidate by VA, All ASID, Inner Shareable

The TLBIMVAAIS characteristics are:

**Purpose**

Invalidate TLB entries for stage 1 of the PL1&0 translation regime, for any ASID, that match the given VA, on all PEs in the same Inner Shareable domain, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIMVAA.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

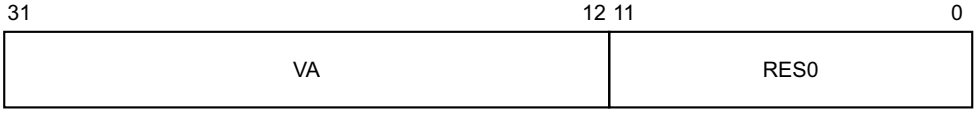
There are no configuration notes.

**Attributes**

TLBIMVAAIS is a 32-bit system operation.

#### Field descriptions

The TLBIMVAAIS input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this operation, regardless of the ASID.

**Bits [11:0]**

Reserved, RES0.

### Performing the TLBIMVAAIS operation:

To perform the TLBIMVAAIS operation:

```
MCR p15,0,<Rt>,c8,c3,3 ; TLBIMVAAIS operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0011 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.142 TLBIMVAAL, TLB Invalidate by VA, All ASID, Last level

The TLBIMVAAL characteristics are:

**Purpose**

Invalidate TLB entries for stage 1 of the PL1&0 translation regime, for the last level of translation, for any ASID, that match the given VA, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIMVAAL.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

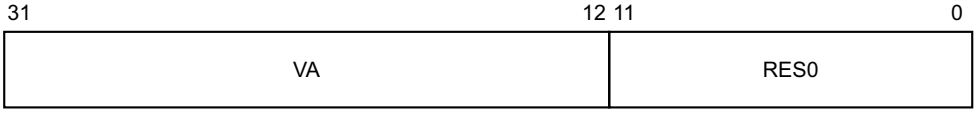
This operation is not implemented in architecture versions before ARMv8.

**Attributes**

TLBIMVAAL is a 32-bit system operation.

#### Field descriptions

The TLBIMVAAL input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this operation, regardless of the ASID.

**Bits [11:0]**

Reserved, RES0.

### Performing the TLBIMVAAL operation:

To perform the TLBIMVAAL operation:

```
MCR p15,0,<Rt>,c8,c7,7 ; TLBIMVAAL operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0111 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.143 TLBIMVAALIS, TLB Invalidate by VA, All ASID, Last level, Inner Shareable

The TLBIMVAALIS characteristics are:

**Purpose**

Invalidate TLB entries for stage 1 of the PL1&0 translation regime, for the last level of translation, for any ASID, that match the given VA, on all PEs in the same Inner Shareable domain, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIMVAAL.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

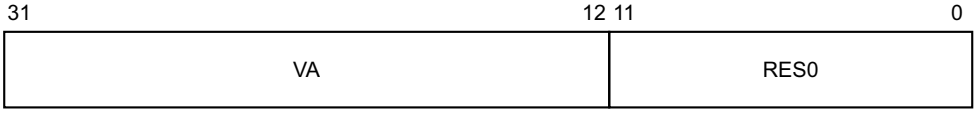
This operation is not implemented in architecture versions before ARMv8.

**Attributes**

TLBIMVAALIS is a 32-bit system operation.

#### Field descriptions

The TLBIMVAALIS input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this operation, regardless of the ASID.

**Bits [11:0]**

Reserved, RES0.

### Performing the TLBIMVAALIS operation:

To perform the TLBIMVAALIS operation:

```
MCR p15,0,<Rt>,c8,c3,7 ; TLBIMVAALIS operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0011 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.144 TLBIMVAH, TLB Invalidate by VA, Hyp mode

The TLBIMVAH characteristics are:

**Purpose**

Invalidate TLB entries for the Non-secure PL2 translation regime that match the given VA.

For details of the scope of this instruction see TLBIMVAH.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0, Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|---|
| - | - | - | WO | WO | WO | WO-UNPREDICTABLE |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If this operation is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE. For more information see *Hyp mode TLB maintenance instructions* on page K1-5499.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

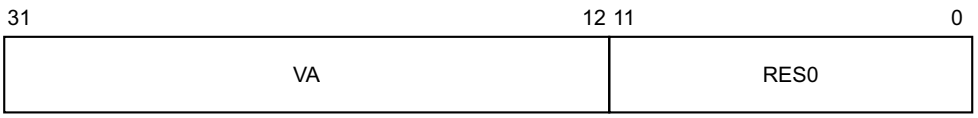
There are no configuration notes.

**Attributes**

TLBIMVAH is a 32-bit system operation.

#### Field descriptions

The TLBIMVAH input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

**Bits [11:0]**

Reserved, RES0.

### Performing the TLBIMVAH operation:

To perform the TLBIMVAH operation:

```
MCR p15,4,<Rt>,c8,c7,1 ; TLBIMVAH operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0111 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.145 TLBIMVAHIS, TLB Invalidate by VA, Hyp mode, Inner Shareable

The TLBIMVAHIS characteristics are:

**Purpose**

Invalidate TLB entries for the Non-secure PL2 translation regime that match the given VA, on all PEs in the same Inner Shareable domain.

For details of the scope of this instruction see TLBIMVAH.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0, Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|---|
| - | - | - | WO | WO | WO | WO-UNPREDICTABLE |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If this operation is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE. For more information see *Hyp mode TLB maintenance instructions* on page K1-5499.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

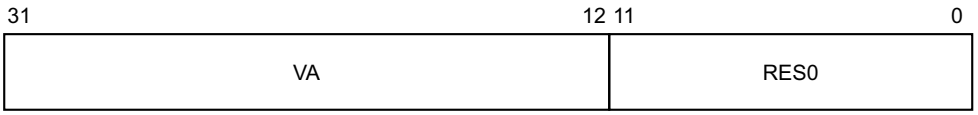
There are no configuration notes.

**Attributes**

TLBIMVAHIS is a 32-bit system operation.

#### Field descriptions

The TLBIMVAHIS input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

**Bits [11:0]**

Reserved, RES0.

### Performing the TLBIMVAHIS operation:

To perform the TLBIMVAHIS operation:

```
MCR p15,4,<Rt>,c8,c3,1 ; TLBIMVAHIS operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.146 TLBIMVAIS, TLB Invalidate by VA, Inner Shareable

The TLBIMVAIS characteristics are:

**Purpose**

Invalidate TLB entries for stage 1 of the PL1&0 translation regime that match the given VA and ASID, on all PEs in the same Inner Shareable domain, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIMVA.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

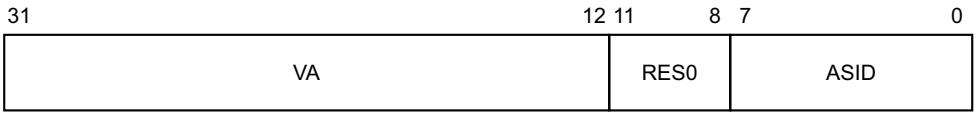
There are no configuration notes.

**Attributes**

TLBIMVAIS is a 32-bit system operation.

#### Field descriptions

The TLBIMVAIS input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

**Bits [11:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

### Performing the TLBIMVAIS operation:

To perform the TLBIMVAIS operation:

```
MCR p15,0,<Rt>,c8,c3,1 ; TLBIMVAIS operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.147 TLBIMVAL, TLB Invalidate by VA, Last level

The TLBIMVAL characteristics are:

**Purpose**

Invalidate TLB entries for the stage 1 PL1&0 translation regime, for the last level of translation, that match the given VA and ASID, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIMVAL.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

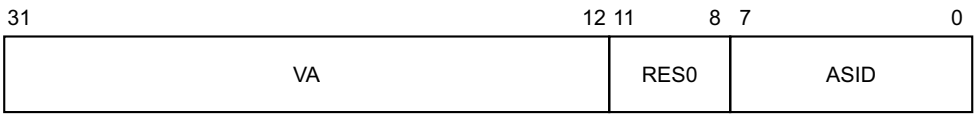
This operation is not implemented in architecture versions before ARMv8.

**Attributes**

TLBIMVAL is a 32-bit system operation.

#### Field descriptions

The TLBIMVAL input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

**Bits [11:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

### Performing the TLBIMVAL operation:

To perform the TLBIMVAL operation:

```
MCR p15,0,<Rt>,c8,c7,5 ; TLBIMVAL operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0111 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.148 TLBIMVALH, TLB Invalidate by VA, Last level, Hyp mode

The TLBIMVALH characteristics are:

**Purpose**

Invalidate TLB entries for the Non-secure PL2 translation regime, for the last level of translation, that match the given VA.

For details of the scope of this instruction see TLBIMVALH.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0, Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|---|
| - | - | - | WO | WO | WO | WO-UNPREDICTABLE |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If this operation is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE. For more information see *Hyp mode TLB maintenance instructions* on page K1-5499.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

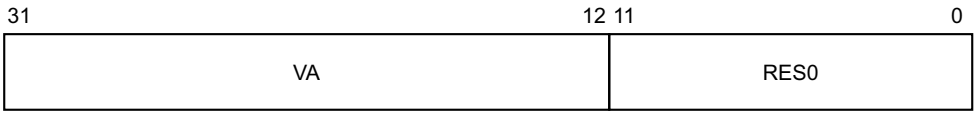
This operation is not implemented in architecture versions before ARMv8.

**Attributes**

TLBIMVALH is a 32-bit system operation.

## Field descriptions

The TLBIMVALH input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

**Bits [11:0]**

Reserved, RES0.

### Performing the TLBIMVALH operation:

To perform the TLBIMVALH operation:

```
MCR p15,4,<Rt>,c8,c7,5 ; TLBIMVALH operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0111 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.149 TLBIMVALHIS, TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable

The TLBIMVALHIS characteristics are:

**Purpose**

Invalidate TLB entries for the Non-secure PL2 translation regime, for the last level of translation, that match the given VA, on all PEs in the same Inner Shareable domain.

For details of the scope of this instruction see TLBIMVALH.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0, Mon) | EL3 (SCR.NS=0, !Mon) |
|---|---|---|---|---|---|---|
| - | - | - | WO | WO | WO | WO-UNPREDICTABLE |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | WO |

If this operation is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE. For more information see *Hyp mode TLB maintenance instructions* on page K1-5499.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

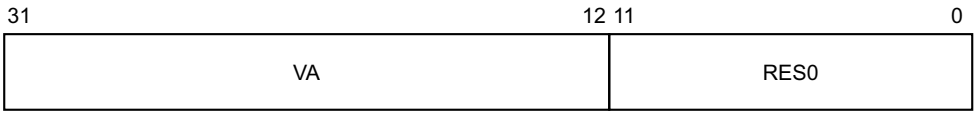
This operation is not implemented in architecture versions before ARMv8.

**Attributes**

TLBIMVALHIS is a 32-bit system operation.

## Field descriptions

The TLBIMVALHIS input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

**Bits [11:0]**

Reserved, RES0.

### Performing the TLBIMVALHIS operation:

To perform the TLBIMVALHIS operation:

```
MCR p15,4,<Rt>,c8,c3,5 ; TLBIMVALHIS operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1000 | 0011 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.150 TLBIMVALIS, TLB Invalidate by VA, Last level, Inner Shareable

The TLBIMVALIS characteristics are:

**Purpose**

Invalidate TLB entries for stage 1 of the PL1&0 translation regime, for the last level of translation, that match the given VA and ASID, on all PEs in the same Inner Shareable domain, subject to the Security state at which the instruction is executed.

For details of the scope of this instruction see TLBIMVAL.

**Usage constraints**

If EL3 is implemented and is using AArch32, this operation can be performed at the following exception levels:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

If this operation is executed at Secure EL1 in AArch32 when EL3 is using AArch64, it only affects TLB entries related to the Secure EL1 translation regime.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TTLB==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HCR_EL2.TTLB==1, Non-secure EL1 execution of this instruction is trapped to EL2.

If HSTR.T8==1, Non-secure EL1 execution of this instruction is trapped to Hyp mode.

If HSTR_EL2.T8==1, Non-secure EL1 execution of this instruction is trapped to EL2.

**Configurations**

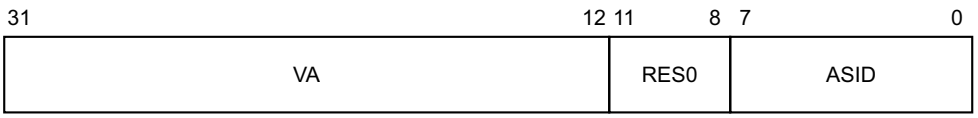
This operation is not implemented in architecture versions before ARMv8.

**Attributes**

TLBIMVALIS is a 32-bit system operation.

#### Field descriptions

The TLBIMVALIS input value bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

**Bits [11:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this operation.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

### Performing the TLBIMVALIS operation:

To perform the TLBIMVALIS operation:

```
MCR p15,0,<Rt>,c8,c3,5 ; TLBIMVALIS operation
```

The operation is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1000 | 0011 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.2.151 TLBTR, TLB Type Register

The TLBTR characteristics are:

**Purpose**

Provides information about the TLB implementation. The register must define whether the implementation provides separate instruction and data TLBs, or a unified TLB. Normally, the IMPLEMENTATION DEFINED information in this register includes the number of lockable entries in the TLB.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TID1==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TID1==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HSTR.T0==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 read accesses to this register are trapped to EL2.

**Configurations**

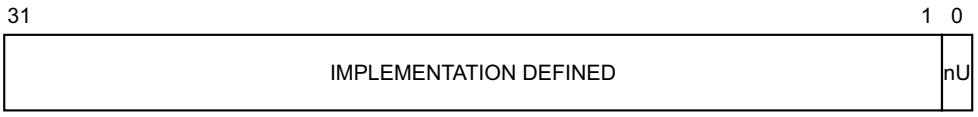
There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

TLBTR is a 32-bit register.

### Field descriptions

The TLBTR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:1]**

IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**nU, bit [0]**

Not Unified TLB. Indicates whether the implementation has a unified TLB:

0 Unified TLB.

1 Separate Instruction and Data TLBs.

### Accessing the TLBTR:

To access the TLBTR:

```
MRC p15,0,<Rt>,c0,c0,3 ; Read TLBTR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0000 | 0000 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.152 TPIDRPRW, PL1 Software Thread ID Register

The TPIDRPRW characteristics are:

**Purpose**

Provides a location where software executing at EL1 or higher can store thread identifying information that is not visible to software executing at EL0, for OS management purposes.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

TPIDRPRW(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

TPIDRPRW(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

TPIDRPRW is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

The PE never updates this register. This means the register is always UNKNOWN on reset.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T13==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T13==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register TPIDRPRW is architecturally mapped to AArch64 System register TPIDR_EL1[31:0].

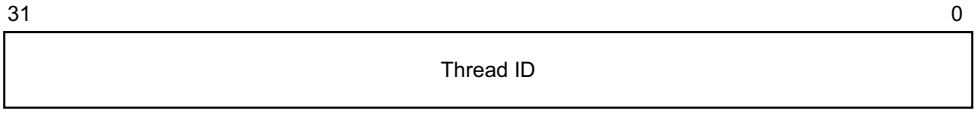
**Attributes**

TPIDRPRW is a 32-bit register.

#### Field descriptions

The TPIDRPRW bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

### Accessing the TPIDRPRW:

To access the TPIDRPRW:

```
MRC p15,0,<Rt>,c13,c0,4 ; Read TPIDRPRW into Rt
MCR p15,0,<Rt>,c13,c0,4 ; Write Rt to TPIDRPRW
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1101 | 0000 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.153 TPIDRURO, PL0 Read-Only Software Thread ID Register

The TPIDRURO characteristics are:

**Purpose**

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

TPIDRURO(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RO | - | - | - | RW |

TPIDRURO(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RO | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

TPIDRURO is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| RO | RW | RW |

The PE never updates this register. This means the register is always UNKNOWN on reset.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T13==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T13==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register TPIDRURO is architecturally mapped to AArch64 System register TPIDRRO_EL0[31:0].

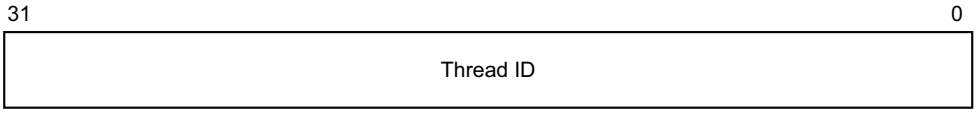
**Attributes**

TPIDRURO is a 32-bit register.

#### Field descriptions

The TPIDRURO bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

### Accessing the TPIDRURO:

To access the TPIDRURO:

```
MRC p15,0,<Rt>,c13,c0,3 ; Read TPIDRURO into Rt
MCR p15,0,<Rt>,c13,c0,3 ; Write Rt to TPIDRURO
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1101 | 0000 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.154 TPIDRURW, PL0 Read/Write Software Thread ID Register

The TPIDRURW characteristics are:

**Purpose**

Provides a location where software executing at EL0 can store thread identifying information, for OS management purposes.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

TPIDRURW(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | RW | - | - | - | RW |

TPIDRURW(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

TPIDRURW is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| RW | RW | RW |

The PE never updates this register. This means the register is always UNKNOWN on reset.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T13==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T13==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register TPIDRURW is architecturally mapped to AArch64 System register TPIDR_EL0[31:0].

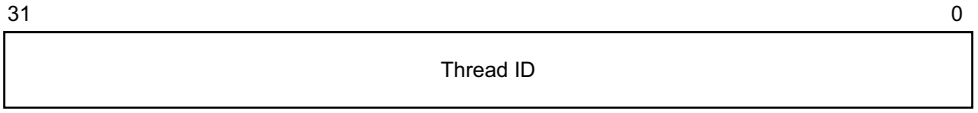
**Attributes**

TPIDRURW is a 32-bit register.

#### Field descriptions

The TPIDRURW bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

### Accessing the TPIDRURW:

To access the TPIDRURW:

```
MRC p15,0,<Rt>,c13,c0,2 ; Read TPIDRURW into Rt
MCR p15,0,<Rt>,c13,c0,2 ; Write Rt to TPIDRURW
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1101 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.155 TTBCR, Translation Table Base Control Register

The TTBCR characteristics are:

**Purpose**

Determines which of the Translation Table Base Registers defined the base address for a translation table walk required for the stage 1 translation of a memory access from any mode other than Hyp mode. Also controls the translation table format and, when using the Long-descriptor translation table format, holds cacheability and shareability information.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

TTBCR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

TTBCR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

TTBCR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T2==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T2==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register TTBCR is architecturally mapped to AArch64 System register TCR_EL1[31:0].

The current translation table format determines which format of the register is used.

When EL3 is using AArch32, write access to TTBCR(S) is disabled when the CP15SDISABLE signal is asserted HIGH.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Some RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. If the PE resets into EL3 using AArch32 then:

- The EAE bit resets to 0 in both the Secure and the Non-secure instances of the register.
- Other reset values apply only to the Secure instance of the register.

**Attributes**

TTBCR is a 32-bit register.

## Field descriptions

The TTBCR bit assignments are:

***For all register layouts:***

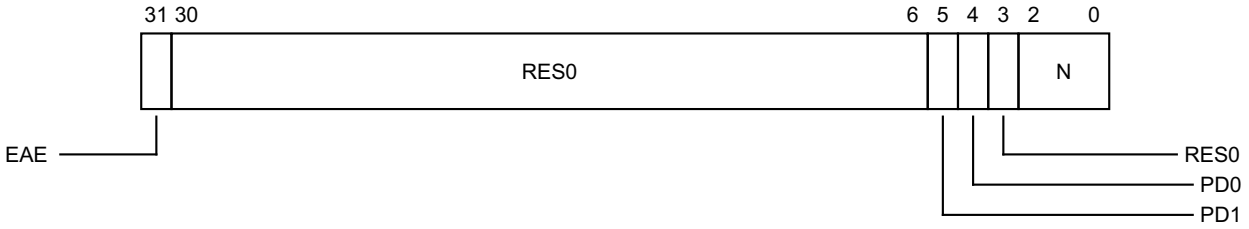
**EAE, bit [31]**

Extended Address Enable. The meanings of the possible values of this bit are:

`0` Use the VMSAv8-32 translation system with the Short-descriptor translation table format.

`1` Use the VMSAv8-32 translation system with the Long-descriptor translation table format.

***When TTBCR.EAE==0:***




**EAE, bit [31]**

Extended Address Enable. The meanings of the possible values of this bit are:

`0` Use the VMSAv8-32 translation system with the Short-descriptor translation table format.

When this register has an architecturally-defined reset value, this field resets to `0`.

**Bits [30:6]**

Reserved, RES0.

**PD1, bit [5]**

Translation table walk disable for translations using TTBR1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1. The encoding of this bit is:

`0` Perform translation table walks using TTBR1.

`1` A TLB miss on an address that is translated using TTBR1 generates a Translation fault. No translation table walk is performed.

When this register has an architecturally-defined reset value, this field resets to `0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**PD0, bit [4]**

Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss for an address that is translated using TTBR0. The encoding of this bit is:

`0` Perform translation table walks using TTBR0.

`1` A TLB miss on an address that is translated using TTBR0 generates a Translation fault. No translation table walk is performed.

When this register has an architecturally-defined reset value, this field resets to `0`.

**Bit [3]**

Reserved, RES0.

**N, bits [2:0]**

Indicate the width of the base address held in TTBR0. In TTBR0, the base address field is bits[31:14-N]. The value of N also determines:

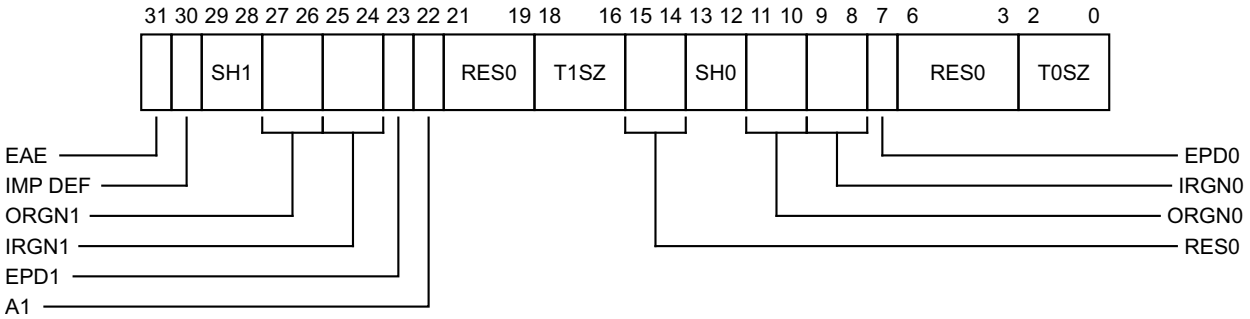
- Whether TTBR0 or TTBR1 is used as the base address for translation table walks.
- The size of the translation table pointed to by TTBR0.

N can take any value from 0 to 7, that is, from `0b000` to `0b111`.

When N has its reset value of 0, the translation table base is compatible with ARMv5 and ARMv6.

When this register has an architecturally-defined reset value, this field resets to `0`.

***When TTBCR.EAE==1:***




**EAE, bit [31]**

Extended Address Enable. The meanings of the possible values of this bit are:

`1` Use the VMSAv8-32 translation system with the Long-descriptor translation table format.

When this register has an architecturally-defined reset value, this field resets to `0`.

**IMP DEF, bit [30]**

IMPLEMENTATION DEFINED.

When this register has an architecturally-defined reset value, this field resets to `0`.

**SH1, bits [29:28]**

Shareability attribute for memory associated with translation table walks using TTBR1.

`00` Non-shareable

`10` Outer Shareable

`11` Inner Shareable


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

When this register has an architecturally-defined reset value, this field resets to 0.

**ORGN1, bits [27:26]**

Outer cacheability attribute for memory associated with translation table walks using TTBR1.

00 Normal memory, Outer Non-cacheable

01 Normal memory, Outer Write-Back Write-Allocate Cacheable

10 Normal memory, Outer Write-Through Cacheable

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable

When this register has an architecturally-defined reset value, this field resets to 0.

**IRGN1, bits [25:24]**

Inner cacheability attribute for memory associated with translation table walks using TTBR1.

00 Normal memory, Inner Non-cacheable

01 Normal memory, Inner Write-Back Write-Allocate Cacheable

10 Normal memory, Inner Write-Through Cacheable

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable

When this register has an architecturally-defined reset value, this field resets to 0.

**EPD1, bit [23]**

Translation table walk disable for translations using TTBR1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1. The encoding of this bit is:

0 Perform translation table walks using TTBR1.

1 A TLB miss on an address that is translated using TTBR1 generates a Translation fault. No translation table walk is performed.

When this register has an architecturally-defined reset value, this field resets to 0.

**A1, bit [22]**

Selects whether TTBR0 or TTBR1 defines the ASID. The encoding of this bit is:

0 TTBR0.ASID defines the ASID.

1 TTBR1.ASID defines the ASID.

When this register has an architecturally-defined reset value, this field resets to 0.

**Bits [21:19]**

Reserved, RES0.

**T1SZ, bits [18:16]**

See *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G4-4026 for how TTBCR.{T1SZ, T0SZ} determine the input address ranges and memory region sizes translated using TTBR0 and TTBR1.

When this register has an architecturally-defined reset value, this field resets to 0.

**Bits [15:14]**

Reserved, RES0.

**SH0, bits [13:12]**

Shareability attribute for memory associated with translation table walks using TTBR0.

00 Non-shareable


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

10 Outer Shareable

11 Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

When this register has an architecturally-defined reset value, this field resets to 0.

**ORGN0, bits [11:10]**

Outer cacheability attribute for memory associated with translation table walks using TTBR0.

00 Normal memory, Outer Non-cacheable

01 Normal memory, Outer Write-Back Write-Allocate Cacheable

10 Normal memory, Outer Write-Through Cacheable

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable

When this register has an architecturally-defined reset value, this field resets to 0.

**IRGN0, bits [9:8]**

Inner cacheability attribute for memory associated with translation table walks using TTBR0.

00 Normal memory, Inner Non-cacheable

01 Normal memory, Inner Write-Back Write-Allocate Cacheable

10 Normal memory, Inner Write-Through Cacheable

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable

When this register has an architecturally-defined reset value, this field resets to 0.

**EPD0, bit [7]**

Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0. The encoding of this bit is:

0 Perform translation table walks using TTBR0.

1 A TLB miss on an address that is translated using TTBR0 generates a Translation fault. No translation table walk is performed.

When this register has an architecturally-defined reset value, this field resets to 0.

**Bits [6:3]**

Reserved, RES0.

**T0SZ, bits [2:0]**

See *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G4-4026 for how TTBCR.{T1SZ, T0SZ} determine the input address ranges and memory region sizes translated using TTBR0 and TTBR1.

When this register has an architecturally-defined reset value, this field resets to 0.

## Accessing the TTBCR:

To access the TTBCR:

```
MRC p15,0,<Rt>,c2,c0,2 ; Read TTBCR into Rt
MCR p15,0,<Rt>,c2,c0,2 ; Write Rt to TTBCR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0010 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.156 TTBR0, Translation Table Base Register 0

The TTBR0 characteristics are:

**Purpose**

Holds the base address of translation table 0, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses from modes other than Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

TTBR0(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

TTBR0(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

TTBR0 is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

Used in conjunction with the TTBCR. When the 64-bit TTBR0 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR0.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T2==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T2==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register TTBR0 is architecturally mapped to AArch64 System register TTBR0_EL1.

TTBCR.EAE determines which TTBR0 format is used:

`EAE==0` 32-bit format is used. TTBR0[63:32] are ignored.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EAE==1 64-bit format is used.

When EL3 is using AArch32, write access to TTBR0(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

RW fields in this register reset to architecturally UNKNOWN values.

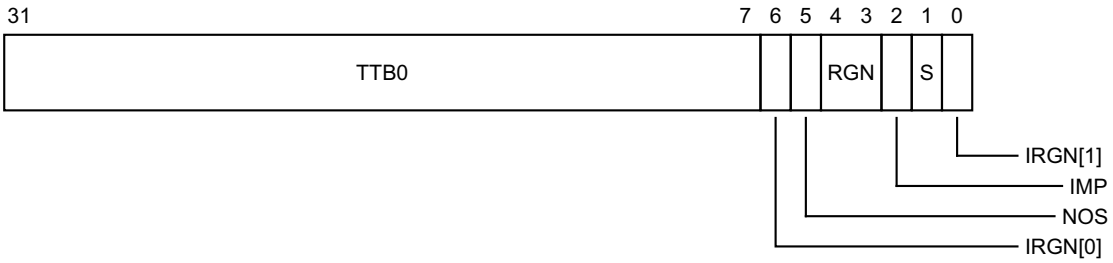
**Attributes**

TTBR0 is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

## Field descriptions

The TTBR0 bit assignments are:

***When TTBCR.EAE==0:***




**TTB0, bits [31:7]**

Translation table base address, bits[31:x], where x is 14-(TTBCR.N). Register bits [x-1:7] are RES0, with the additional requirement that if these bits are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:7] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

**IRGN[0], bit [6]**

See IRGN[1] below for description of the IRGN field.

**NOS, bit [5]**

Not Outer Shareable. When the value of TTBR0.S is 1, indicates whether the memory associated with a translation table walk is Inner Shareable or Outer Shareable:

0 Memory is Outer Shareable.

1 Memory is Inner Shareable.

This bit is ignored when the value of TTBR0.S is 0.

**RGN, bits [4:3]**

Region bits. Indicates the Outer cacheability attributes for the memory associated with the translation table walks:

00 Normal memory, Outer Non-cacheable.

01 Normal memory, Outer Write-Back Write-Allocate Cacheable.

10 Normal memory, Outer Write-Through Cacheable.

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IMP, bit [2]**

The effect of this bit is IMPLEMENTATION DEFINED. If the translation table implementation does not include any IMPLEMENTATION DEFINED features this bit is UNK/SBZP.

**S, bit [1]**

Shareable. Indicates whether the memory associated with the translation table walks is Non-shareable:

0 Memory is Non-shareable.

1 Memory is shareable. The TTBR0.NOS field indicates whether the memory is Inner Shareable or Outer Shareable.

**IRGN[1], bit [0]**

Inner region bits. Indicates the Inner Cacheability attributes for the memory associated with the translation table walks. The possible values of IRGN[1:0] are:

00 Normal memory, Inner Non-cacheable.

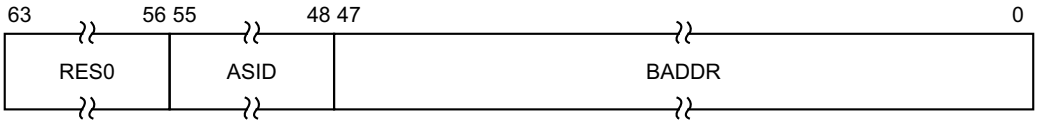
01 Normal memory, Inner Write-Back Write-Allocate Cacheable.

10 Normal memory, Inner Write-Through Cacheable.

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable.

The encoding of the IRGN bits is counter-intuitive, with register bit[6] being IRGN[0] and register bit[0] being IRGN[1]. This encoding is chosen to give a consistent encoding of memory region types and to ensure that software written for ARMv7 without the Multiprocessing Extensions can run unmodified on an implementation that includes the functionality introduced by the ARMv7 Multiprocessing Extensions.

***When TTBCR.EAE==1:***




**Bits [63:56]**

Reserved, RES0.

**ASID, bits [55:48]**

An ASID for the translation table base address. The TTBCR.A1 field selects either TTBR0.ASID or TTBR1.ASID.

**BADDR, bits [47:0]**

Translation table base address, bits[47:x], Bits [x-1:0] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of TTBCR.T0SZ as follows:

- If TTBCR.T0SZ is 0 or 1, x = 5 - TTBCR.T0SZ.
- If TTBCR.T0SZ is greater than 1, x = 14 - TTBCR.T0SZ.

If bits[47:40] of the Translation table base address are not zero, an Address size fault is generated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the TTBR0:

To access the TTBR0 when accessing as a 32-bit register:

```
MRC p15,0,<Rt>,c2,c0,0 ; Read TTBR0[31:0] into Rt
MCR p15,0,<Rt>,c2,c0,0 ; Write Rt to TTBR0[31:0]. TTBR0[63:32] are unchanged
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0010 | 0000 | 000 |

To access the TTBR0 when accessing as a 64-bit register:

```
MRRC p15,0,<Rt>,<Rt2>,c2 ; Read TTBR0[31:0] into Rt and TTBR0[63:32] into Rt2
MCRR p15,0,<Rt>,<Rt2>,c2 ; Write Rt to TTBR0[31:0] and Rt2 to TTBR0[63:32]
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0000 | 0010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.157 TTBR1, Translation Table Base Register 1

The TTBR1 characteristics are:

**Purpose**

Holds the base address of translation table 1, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses from modes other than Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

TTBR1(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

TTBR1(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

TTBR1 is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

Used in conjunction with the TTBCR. When the 64-bit TTBR1 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR1.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HCR.TVM==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If HCR.TRVM==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If HCR_EL2.TRVM==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If HCR_EL2.TVM==1, Non-secure EL1 write accesses to this register are trapped to EL2.

If HSTR.T2==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T2==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register TTBR1 is architecturally mapped to AArch64 System register TTBR1_EL1.

TTBCR.EAE determines which TTBR1 format is used:

`EAE==0` 32-bit format is used. TTBR1[63:32] are ignored.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

EAE==1 64-bit format is used.

RW fields in this register reset to architecturally UNKNOWN values.

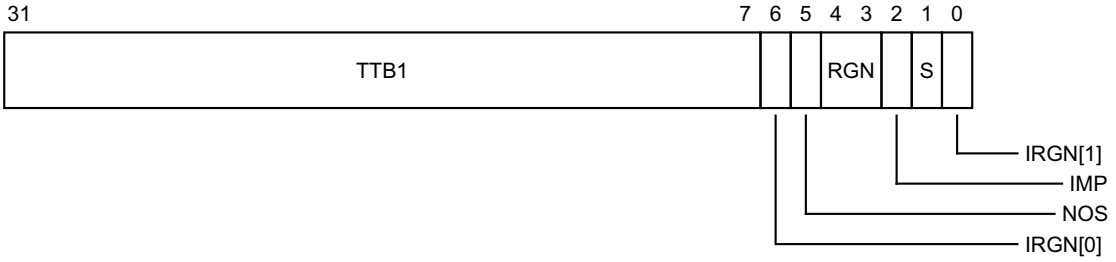
**Attributes**

TTBR1 is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

## Field descriptions

The TTBR1 bit assignments are:

***When TTBCR.EAE==0:***




**TTB1, bits [31:7]**

Translation table base address, bits[31:14]. Register bits [13:7] are RES0, with the additional requirement that if these bits are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [13:7] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

**IRGN[0], bit [6]**

See IRGN[1] below for description of the IRGN field.

**NOS, bit [5]**

Not Outer Shareable. When the value of TTBR1.S is 1, indicates whether the memory associated with a translation table walk is Inner Shareable or Outer Shareable:

0 Memory is Outer Shareable.

1 Memory is Inner Shareable.

This bit is ignored when the value of TTBR1.S is 0.

**RGN, bits [4:3]**

Region bits. Indicates the Outer cacheability attributes for the memory associated with the translation table walks:

00 Normal memory, Outer Non-cacheable.

01 Normal memory, Outer Write-Back Write-Allocate Cacheable.

10 Normal memory, Outer Write-Through Cacheable.

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable.

**IMP, bit [2]**

The effect of this bit is IMPLEMENTATION DEFINED. If the translation table implementation does not include any IMPLEMENTATION DEFINED features this bit is UNK/SBZP.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**S, bit [1]**

Shareable. Indicates whether the memory associated with the translation table walks is Non-shareable:

0 Memory is Non-shareable.

1 Memory is shareable. The TTBR1.NOS field indicates whether the memory is Inner Shareable or Outer Shareable.

**IRGN[1], bit [0]**

Inner region bits. Indicates the Inner Cacheability attributes for the memory associated with the translation table walks. The possible values of IRGN[1:0] are:

00 Normal memory, Inner Non-cacheable.

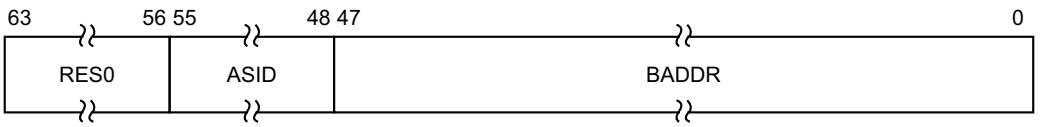
01 Normal memory, Inner Write-Back Write-Allocate Cacheable.

10 Normal memory, Inner Write-Through Cacheable.

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable.

The encoding of the IRGN bits is counter-intuitive, with register bit[6] being IRGN[0] and register bit[0] being IRGN[1]. This encoding is chosen to give a consistent encoding of memory region types and to ensure that software written for ARMv7 without the Multiprocessing Extensions can run unmodified on an implementation that includes the functionality introduced by the ARMv7 Multiprocessing Extensions.

***When TTBCR.EAE==1:***




**Bits [63:56]**

Reserved, RES0.

**ASID, bits [55:48]**

An ASID for the translation table base address. The TTBCR.A1 field selects either TTBR0.ASID or TTBR1.ASID.

**BADDR, bits [47:0]**

Translation table base address, bits[47:x], Bits [x-1:0] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of TTBCR.T1SZ as follows:

- If TTBCR.T1SZ is 0 or 1, x = 5 - TTBCR.T1SZ.
- If TTBCR.T1SZ is greater than 1, x = 14 - TTBCR.T1SZ.

If bits[47:40] of the Translation table base address are not zero, an Address size fault is generated.

## Accessing the TTBR1:

To access the TTBR1 when accessing as a 32-bit register:

```
MRC p15,0,<Rt>,c2,c0,1 ; Read TTBR1[31:0] into Rt
MCR p15,0,<Rt>,c2,c0,1 ; Write Rt to TTBR1[31:0]. TTBR1[63:32] are unchanged
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0010 | 0000 | 001 |

To access the TTBR1 when accessing as a 64-bit register:

```
MRRC p15,1,<Rt>,<Rt2>,c2 ; Read TTBR1[31:0] into Rt and TTBR1[63:32] into Rt2
MCRR p15,1,<Rt>,<Rt2>,c2 ; Write Rt to TTBR1[31:0] and Rt2 to TTBR1[63:32]
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0001 | 0010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.158 VBAR, Vector Base Address Register

The VBAR characteristics are:

**Purpose**

When high exception vectors are not selected, holds the vector base address for exceptions that are not taken to Monitor mode or to Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

VBAR(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | - | RW |

VBAR(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

VBAR is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

Software must program VBAR(NS) with the required initial value as part of the PE boot sequence.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T12==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T12==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register VBAR is architecturally mapped to AArch64 System register VBAR_EL1[31:0].

When EL3 is using AArch32, write access to VBAR(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

Some or all RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. If the PE resets into EL3 using AArch32 they apply only to the Secure instance of the register. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

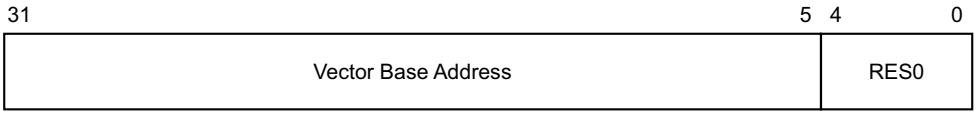
**Attributes**

VBAR is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The VBAR bit assignments are:




**Bits [31:5]**

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

When this register has an architecturally-defined reset value, this field resets to an IMPLEMENTATION DEFINED value.

**Bits [4:0]**

Reserved, RES0.

### Accessing the VBAR:

To access the VBAR:

```
MRC p15,0,<Rt>,c12,c0,0 ; Read VBAR into Rt
MCR p15,0,<Rt>,c12,c0,0 ; Write Rt to VBAR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1100 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.159 VMPIDR, Virtualization Multiprocessor ID Register

The VMPIDR characteristics are:

**Purpose**

Holds the value of the Virtualization Multiprocessor ID. This is the value returned by Non-secure EL1 reads of MPIDR.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T0==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register VMPIDR is architecturally mapped to AArch64 System register VMPIDR_EL2[31:0].

If EL2 is not implemented but EL3 is implemented, this register takes the value of the MPIDR.

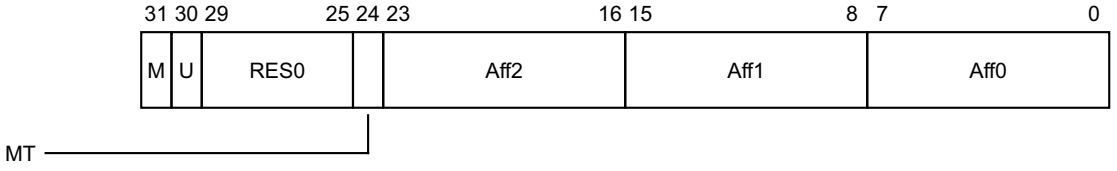
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 with EL2 using AArch32, or into EL3 with EL3 using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

VMPIDR is a 32-bit register.

#### Field descriptions

The VMPIDR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**M, bit [31]**

Indicates whether this implementation includes the functionality introduced by the ARMv7 Multiprocessing Extensions. The possible values of this bit are:

0 This implementation does not include the ARMv7 Multiprocessing Extensions functionality.

1 This implementation includes the ARMv7 Multiprocessing Extensions functionality.

In ARMv8 this bit is RES1.

**U, bit [30]**

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system. The possible values of this bit are:

0 Processor is part of a multiprocessor system.

1 Processor is part of a uniprocessor system.

When this register has an architecturally-defined reset value, this field resets to the value of MPIDR.U.

**Bits [29:25]**

Reserved, RES0.

**MT, bit [24]**

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach. The possible values of this bit are:

0 Performance of PEs at the lowest affinity level is largely independent.

1 Performance of PEs at the lowest affinity level is very interdependent.

When this register has an architecturally-defined reset value, this field resets to the value of MPIDR.MT.

**Aff2, bits [23:16]**

Affinity level 2. The least significant affinity level field, for this PE in the system.

When this register has an architecturally-defined reset value, this field resets to the value of MPIDR.Aff2.

**Aff1, bits [15:8]**

Affinity level 1. The intermediate affinity level field, for this PE in the system.

When this register has an architecturally-defined reset value, this field resets to the value of MPIDR.Aff1.

**Aff0, bits [7:0]**

Affinity level 0. The most significant affinity level field, for this PE in the system.

When this register has an architecturally-defined reset value, this field resets to the value of MPIDR.Aff0.

## Accessing the VMPIDR:

To access the VMPIDR:

```
MRC p15,4,<Rt>,c0,c0,5 ; Read VMPIDR into Rt
MCR p15,4,<Rt>,c0,c0,5 ; Write Rt to VMPIDR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0000 | 0000 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.160 VPIDR, Virtualization Processor ID Register

The VPIDR characteristics are:

**Purpose**

Holds the value of the Virtualization Processor ID. This is the value returned by Non-secure EL1 reads of MIDR.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T0==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T0==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register VPIDR is architecturally mapped to AArch64 System register VPIDR_EL2.

If EL2 is not implemented but EL3 is implemented, this register takes the value of the MIDR.

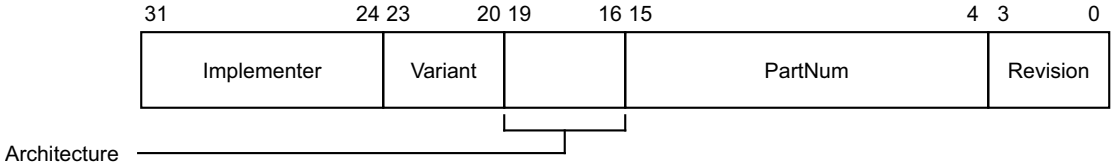
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 with EL2 using AArch32, or into EL3 with EL3 using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

VPIDR is a 32-bit register.

#### Field descriptions

The VPIDR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by ARM. Assigned codes include the following:

| Hex representation | ASCII representation | Implementer |
|---|---|---|
| 0x41 | A | ARM Limited |
| 0x42 | B | Broadcom Corporation |
| 0x43 | C | Cavium Inc. |
| 0x44 | D | Digital Equipment Corporation |
| 0x49 | I | Infineon Technologies AG |
| 0x4D | M | Motorola or Freescale Semiconductor Inc. |
| 0x4E | N | NVIDIA Corporation |
| 0x50 | P | Applied Micro Circuits Corporation |
| 0x51 | Q | Qualcomm Inc. |
| 0x56 | V | Marvell International Ltd. |
| 0x69 | i | Intel Corporation |

ARM can assign codes that are not published in this manual. All values not assigned by ARM are reserved and must not be used.

When this register has an architecturally-defined reset value, this field resets to the value of MIDR.Implementer.

**Variant, bits [23:20]**

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

When this register has an architecturally-defined reset value, this field resets to the value of MIDR.Variant.

**Architecture, bits [19:16]**

The permitted values of this field are:

0001 ARMv4

0010 ARMv4T

0011 ARMv5 (obsolete)

0100 ARMv5T

0101 ARMv5TE

0110 ARMv5TEJ

0111 ARMv6

1111 Architectural features are individually identified in the ID_* registers, see *Identification registers, functional group* on page G4-4162.

All other values are reserved.

When this register has an architecturally-defined reset value, this field resets to the value of MIDR.Architecture.

**PartNum, bits [15:4]**

An IMPLEMENTATION DEFINED primary part number for the device.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

On processors implemented by ARM, if the top four bits of the primary part number are `0x0` or `0x7`, the variant and architecture are encoded differently.

When this register has an architecturally-defined reset value, this field resets to the value of MIDR.PartNum.

**Revision, bits [3:0]**

An IMPLEMENTATION DEFINED revision number for the device.

When this register has an architecturally-defined reset value, this field resets to the value of MIDR.Revision.

## Accessing the VPIDR:

To access the VPIDR:

```
MRC p15,4,<Rt>,c0,c0,0 ; Read VPIDR into Rt
MCR p15,4,<Rt>,c0,c0,0 ; Write Rt to VPIDR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0000 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.161 VTCR, Virtualization Translation Control Register

The VTCR characteristics are:

**Purpose**

Controls the translation table walks required for the stage 2 translation of memory accesses from Non-secure modes other than Hyp mode, and holds cacheability and shareability information for the accesses.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

Used in conjunction with VTTBR, that defines the translation table base address for the translations.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T2==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T2==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register VTCR is architecturally mapped to AArch64 System register VTCR_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

RW fields in this register reset to architecturally UNKNOWN values.

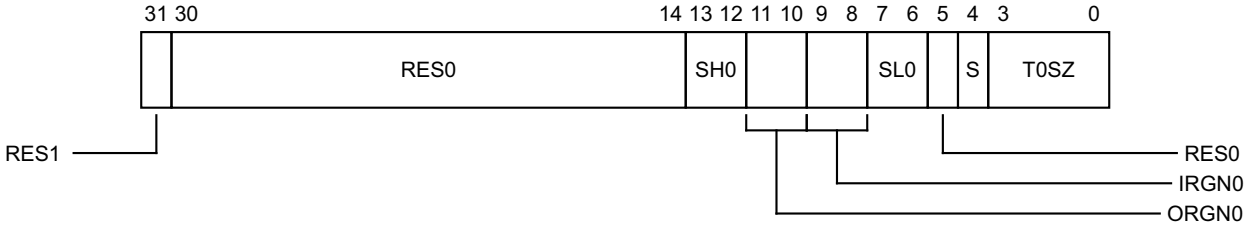
**Attributes**

VTCR is a 32-bit register.

#### Field descriptions

The VTCR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bit [31]**

Reserved, RES1.

**Bits [30:14]**

Reserved, RES0.

**SH0, bits [13:12]**

Shareability attribute for memory associated with translation table walks using VTTBR.

00 Non-shareable

10 Outer Shareable

11 Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

**ORGN0, bits [11:10]**

Outer cacheability attribute for memory associated with translation table walks using VTTBR.

00 Normal memory, Outer Non-cacheable

01 Normal memory, Outer Write-Back Write-Allocate Cacheable

10 Normal memory, Outer Write-Through Cacheable

11 Normal memory, Outer Write-Back no Write-Allocate Cacheable

**IRGN0, bits [9:8]**

Inner cacheability attribute for memory associated with translation table walks using VTTBR.

00 Normal memory, Inner Non-cacheable

01 Normal memory, Inner Write-Back Write-Allocate Cacheable

10 Normal memory, Inner Write-Through Cacheable

11 Normal memory, Inner Write-Back no Write-Allocate Cacheable

**SL0, bits [7:6]**

Starting level for translation table walks using VTTBR.

00 Start at level 2

01 Start at level 1

All other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

If the stage 2 input address size, as programmed in VTCR.T0SZ, is out of range with respect to the starting level at the time of a translation walk that uses the stage 2 translation, then a second stage level 1 translation fault is generated.

**Bit [5]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**S, bit [4]**

Sign extension bit. This bit must be programmed to the value of T0SZ[3]. If it is not, then the stage 2 T0SZ value is treated as an UNKNOWN value.

**T0SZ, bits [3:0]**

The size offset of the memory region addressed by VTTBR. The region size is $2^{(32-T0SZ)}$ bytes.

This field holds a four-bit signed integer value, meaning it supports values from -8 to 7. This is different from the other translation control registers, where TnSZ holds a three-bit unsigned integer, supporting values from 0 to 7.

## Accessing the VTCR:

To access the VTCR:

```
MRC p15,4,<Rt>,c2,c1,2 ; Read VTCR into Rt
MCR p15,4,<Rt>,c2,c1,2 ; Write Rt to VTCR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0010 | 0001 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.2.162 VTTBR, Virtualization Translation Table Base Register

The VTTBR characteristics are:

**Purpose**

Holds the base address of the translation table for the stage 2 translation of memory accesses from Non-secure modes other than Hyp mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

Used in conjunction with the VTCR.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T2==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T2==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

AArch32 System register VTTBR is architecturally mapped to AArch64 System register VTTBR_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

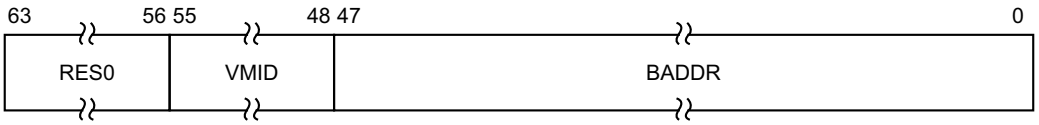
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 with EL2 using AArch32, or into EL3 with EL3 using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

VTTBR is a 64-bit register.

#### Field descriptions

The VTTBR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [63:56]**

Reserved, RES0.

**VMID, bits [55:48]**

The VMID for the translation table.

When this register has an architecturally-defined reset value, this field resets to `0`.

**BADDR, bits [47:0]**

Translation table base address, bits[47:x], Bits [x-1:0] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned Translation Table Base Address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of VTCR.T0SZ as follows:

- If VTCR.T0SZ is 0 or 1, x = 5 - VTCR.T0SZ.
- If VTCR.T0SZ is greater than 1, x = 14 - VTCR.T0SZ.

If bits[47:40] of the Translation table base address are not zero, an Address size fault is generated.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

## Accessing the VTTBR:

To access the VTTBR:

```
MRRC p15,6,<Rt>,<Rt2>,c2 ; Read VTTBR[31:0] into Rt and VTTBR[63:32] into Rt2
MCRR p15,6,<Rt>,<Rt2>,c2 ; Write Rt to VTTBR[31:0] and Rt2 to VTTBR[63:32]
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0110 | 0010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.3 Debug registers

This section lists the Debug System registers in AArch32 state, in alphabetic order.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.1 DBGAUTHSTATUS, Debug Authentication Status register

The DBGAUTHSTATUS characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 read accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGAUTHSTATUS is architecturally mapped to AArch64 System register DBGAUTHSTATUS_EL1.

AArch32 System register DBGAUTHSTATUS is architecturally mapped to External register DBGAUTHSTATUS_EL1.

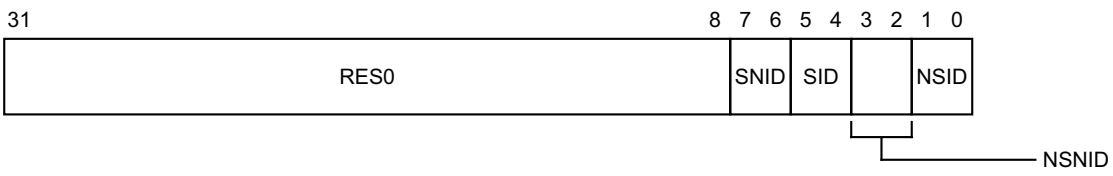
This register is required in all implementations.

**Attributes**

DBGAUTHSTATUS is a 32-bit register.

#### Field descriptions

The DBGAUTHSTATUS bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:8]**

Reserved, RES0.

**SNID, bits [7:6]**

Secure non-invasive debug. Possible values of this field are:

00 Not implemented. EL3 is not implemented and the implemented Security state is Non-Secure state.

10 Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.

11 Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

Other values are reserved.

**SID, bits [5:4]**

Secure invasive debug. Possible values of this field are:

00 Not implemented. EL3 is not implemented and the implemented Security state is Non-Secure state.

10 Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.

11 Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.

Other values are reserved.

**NSNID, bits [3:2]**

Non-secure non-invasive debug. Possible values of this field are:

00 Not implemented. EL3 is not implemented and the implemented Security state is Secure state.

10 Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.

11 Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

Other values are reserved.

**NSID, bits [1:0]**

Non-secure invasive debug. Possible values of this field are:

00 Not implemented. EL3 is not implemented and the implemented Security state is Secure state.

10 Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.

11 Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.

Other values are reserved.

### Accessing the DBGAUTHSTATUS:

To access the DBGAUTHSTATUS:

```
MRC p14,0,<Rt>,c7,c14,6 ; Read DBGAUTHSTATUS into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0111 | 1110 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.2 DBGBCR<n>, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n> characteristics are:

**Purpose**

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>. If EL2 is implemented and this breakpoint supports Context matching, DBGBVR<n> can be associated with a Breakpoint Extended Value Register DBGBXVR<n> for VMID matching.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

When the E field is zero, all the other fields in the register are ignored.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see:

- *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state.
- *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.
- *Software Access debug event* on page H3-4881 for accesses to this register taken to Debug state.

Subject to the prioritization rules:

If EDSCR.TDA==1, OSLSR.OSLK==0, and halting is allowed, EL1 and EL2 accesses to this register generate a Software Access debug event.

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGBCR<n> is architecturally mapped to AArch64 System register DBGBCR<n>_EL1.

AArch32 System register DBGBCR<n> is architecturally mapped to External register DBGBCR<n>_EL1.

If breakpoint n is not implemented then this register is unallocated.

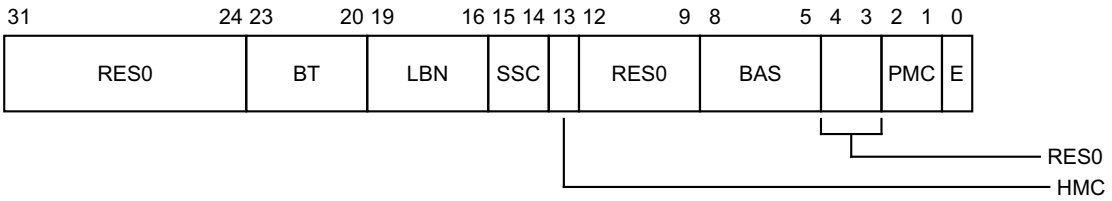
**Attributes**

DBGBCR<n> is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The DBGBCR<n> bit assignments are:




**Bits [31:24]**

Reserved, RES0.

**BT, bits [23:20]**

Breakpoint Type. Possible values are:

`0000` Unlinked instruction address match.

`0001` Linked instruction address match.

`0010` Unlinked context ID match.

`0011` Linked context ID match

`0100` Unlinked instruction address mismatch.

`0101` Linked instruction address mismatch.

`1000` Unlinked VMID match.

`1001` Linked VMID match.

`1010` Unlinked VMID and context ID match.

`1011` Linked VMID and context ID match.

The field breaks down as follows:

- BT[3:1]: Base type.

  `000` Match address. DBGBVR<n> is the address of an instruction.

  `010` Mismatch address. Behaves as type `000` if in an AArch64 translation, or if Halting debug is enabled and halting is allowed. Otherwise, DBGBVR<n> is the address of an instruction to be stepped.

  `001` Match context ID. DBGBVR<n> is a context ID.

  `100` Match VMID. DBGBXVR<n>.VMID is a VMID.

  `101` Match VMID and context ID. DBGBVR<n> is a context ID, and DBGBXVR<n>.VMID is a VMID.

- BT[0]: Enable linking.

Values `0b011x` and `0b11xx` are always reserved. Constraints on breakpoint programming mean other values are reserved under some conditions. For more information, including the effect of programming this field to a reserved value, see *Reserved DBGBCR<n>.BT values* on page G2-3915.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**LBN, bits [19:16]**

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of DBGBCR<n>.E is 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**SSC, bits [15:14]**

Security state control. Determines the Security states under which a breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see *Usage constraints* on page G2-3914.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**HMC, bit [13]**

Higher mode control. Determines the debug perspective for deciding when a breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the SSC, bits [15:14] description.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [12:9]**

Reserved, RES0.

**BAS, bits [8:5]**

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state. In an AArch64-only implementation, this field is reserved, RES1. Otherwise:

- BAS[2] and BAS[0] are read/write.
- BAS[3] and BAS[1] are read-only copies of BAS[2] and BAS[0] respectively.

The values `0011` and `1100` are only supported if AArch32 is supported at any Exception level.

The permitted values depend on the breakpoint type.

For Address match breakpoints, the permitted values are:

| BAS | Match instruction at | Constraint for debuggers |
|---|---|---|
| `0011` | DBGBVR<n> | Use for T32 instructions. |
| `1100` | DBGBVR<n>+2 | Use for T32 instructions. |
| `1111` | DBGBVR<n> | Use for A32 instructions. |

`0000` is reserved and must behave as if the breakpoint is disabled or map to a permitted value.

For more information, see *Using the BAS field in Address Match breakpoints* on page G2-3909.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

| BAS | Step instruction at | Constraint for debuggers |
|---|---|---|
| 0000 | - | Use for a match anywhere breakpoint. |
| 0011 | DBGBVR<n> | Use for stepping T32 instructions. |
| 1100 | DBGBVR<n>+2 | Use for stepping T32 instructions. |
| 1111 | DBGBVR<n> | Use for stepping A32 instructions. |

For more information, see *Using the BAS field in Address Match breakpoints* on page G2-3909.

For Context matching breakpoints, this field is RES1 and ignored.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [4:3]**

Reserved, RES0.

**PMC, bits [2:1]**

Privilege mode control. Determines the Exception level or levels at which a breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>.SSC description.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**E, bit [0]**

Enable breakpoint DBGBVR<n>. Possible values are:

0 Breakpoint disabled.

1 Breakpoint enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

### Accessing the DBGBCR<n>:

To access the DBGBCR<n>:

```
MRC p14,0,<Rt>,c0,<CRm>,5 ; Read DBGBCR<n> into Rt, where n is in the range 0 to 15
MCR p14,0,<Rt>,c0,<CRm>,5 ; Write Rt to DBGBCR<n>, where n is in the range 0 to 15
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | n<3:0> | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.3 DBGBVR<n>, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n> characteristics are:

**Purpose**

Holds a value for use in breakpoint matching, either the virtual address of an instruction or a context ID. Forms breakpoint n together with control register DBGBCR<n>. If EL2 is implemented and this breakpoint supports Context matching, DBGBVR<n> can be associated with a Breakpoint Extended Value Register DBGBXVR<n> for VMID matching.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

Some breakpoints might not support Context ID comparison. For more information, see the description of the DBGDIDR.CTX_CMPs field.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see:

- *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state.
- *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.
- *Software Access debug event* on page H3-4881 for accesses to this register taken to Debug state.

Subject to the prioritization rules:

If EDSCR.TDA==1, OSLSR.OSLK==0, and halting is allowed, EL1 and EL2 accesses to this register generate a Software Access debug event.

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGBVR<n> is architecturally mapped to AArch64 System register DBGBVR<n>_EL1[31:0].

AArch32 System register DBGBVR<n> is architecturally mapped to External register DBGBVR<n>_EL1[31:0].

If breakpoint n is not implemented then this register is unallocated.

**Attributes**

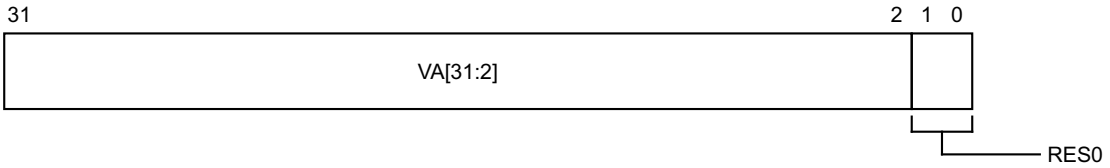
DBGBVR<n> is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The DBGBVR<n> bit assignments are:

***When DBGBCR<n>.BT==0b0x0x:***




**VA[31:2], bits [31:2]**

Bits[31:2] of the address value for comparison.

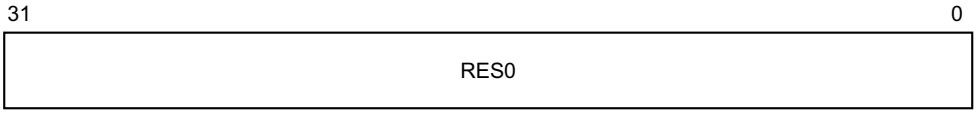
When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [1:0]**

Reserved, RES0.

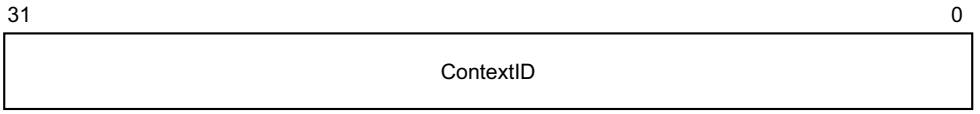
***When DBGBCR<n>.BT==0b1x0x:***




**Bits [31:0]**

Reserved, RES0.

***When DBGBCR<n>.BT==0xxx1x:***




**ContextID, bits [31:0]**

Context ID value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

### Accessing the DBGBVR<n>:

To access the DBGBVR<n>:

```
MRC p14,0,<Rt>,c0,<CRm>,4 ; Read DBGBVR<n> into Rt, where n is in the range 0 to 15
MCR p14,0,<Rt>,c0,<CRm>,4 ; Write Rt to DBGBVR<n>, where n is in the range 0 to 15
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | n<3:0> | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.4 DBGBXVR<n>, Debug Breakpoint Extended Value Registers, n = 0 - 15

The DBGBXVR<n> characteristics are:

**Purpose**

Holds a value for use in breakpoint matching, to support VMID matching. Used in conjunction with a control register DBGBCR<n> and a value register DBGBVR<n>, where EL2 is implemented and breakpoint n supports Context matching.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see:

- *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state.
- *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.
- *Software Access debug event* on page H3-4881 for accesses to this register taken to Debug state.

Subject to the prioritization rules:

If EDSCR.TDA==1, OSLSR.OSLK==0, and halting is allowed, EL1 and EL2 accesses to this register generate a Software Access debug event.

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGBXVR<n> is architecturally mapped to AArch64 System register DBGBVR<n>_EL1[63:32].

AArch32 System register DBGBXVR<n> is architecturally mapped to External register DBGBVR<n>_EL1[63:32].

This register is unallocated in any of the following cases:

- Breakpoint n is not implemented.
- Breakpoint n does not support Context matching.
- EL2 is not implemented.

**Attributes**

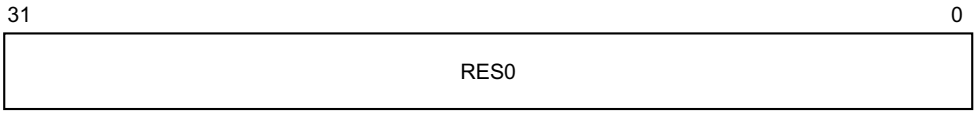
DBGBXVR<n> is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The DBGBXVR<n> bit assignments are:

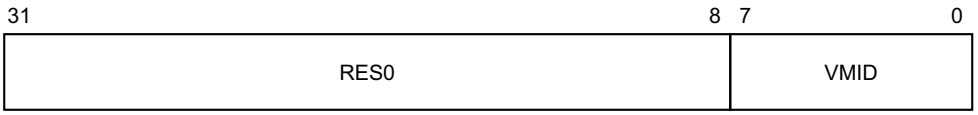
***When DBGBCR<n>.BT==0b0xxx:***




**Bits [31:0]**

Reserved, RES0.

***When DBGBCR<n>.BT==0b1xxx and EL2 implemented:***




**Bits [31:8]**

Reserved, RES0.

**VMID, bits [7:0]**

VMID value for comparison.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

### Accessing the DBGBXVR<n>:

To access the DBGBXVR<n>:

```
MRC p14,0,<Rt>,c1,<CRm>,1 ; Read DBGBXVR<n> into Rt, where n is in the range 0 to 15
MCR p14,0,<Rt>,c1,<CRm>,1 ; Write Rt to DBGBXVR<n>, where n is in the range 0 to 15
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0001 | n<3:0> | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.5 DBGCLAIMCLR, Debug CLAIM Tag Clear register

The DBGCLAIMCLR characteristics are:

**Purpose**

Used by software to read the values of the CLAIM tag bits, and to clear these bits to 0.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

The architecture does not define any functionality for the CLAIM tag bits.

Used in conjunction with the DBGCLAIMSET register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGCLAIMCLR is architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1.

AArch32 System register DBGCLAIMCLR is architecturally mapped to External register DBGCLAIMCLR_EL1.

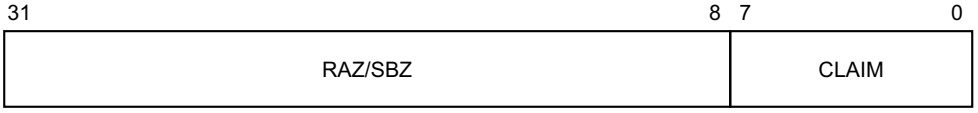
An implementation must include 8 CLAIM tag bits.

**Attributes**

DBGCLAIMCLR is a 32-bit register.

#### Field descriptions

The DBGCLAIMCLR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:8]**

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

**CLAIM, bits [7:0]**

Read or clear CLAIM tag bits. Reading this field returns the current value of the CLAIM tag bits.

Writing a 1 to one of these bits clears the corresponding CLAIM tag bit to 0. This is an indirect write to the CLAIM tag bits. A single write operation can clear multiple CLAIM tag bits to 0.

Writing 0 to one of these bits has no effect.

A cold reset clears the CLAIM tag bits to 0.

### Accessing the DBGCLAIMCLR:

To access the DBGCLAIMCLR:

```
MRC p14,0,<Rt>,c7,c9,6 ; Read DBGCLAIMCLR into Rt
MCR p14,0,<Rt>,c7,c9,6 ; Write Rt to DBGCLAIMCLR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0111 | 1001 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.6 DBGCLAIMSET, Debug CLAIM Tag Set register

The DBGCLAIMSET characteristics are:

**Purpose**

Used by software to set the CLAIM tag bits to 1.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

The architecture does not define any functionality for the CLAIM tag bits.

Used in conjunction with the DBGCLAIMCLR register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGCLAIMSET is architecturally mapped to AArch64 System register DBGCLAIMSET_EL1.

AArch32 System register DBGCLAIMSET is architecturally mapped to External register DBGCLAIMSET_EL1.

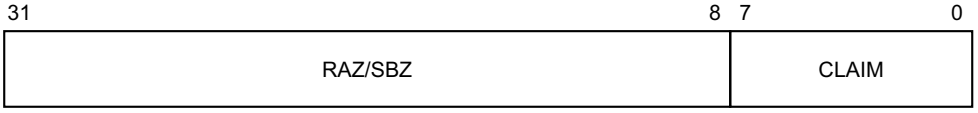
An implementation must include 8 CLAIM tag bits.

**Attributes**

DBGCLAIMSET is a 32-bit register.

#### Field descriptions

The DBGCLAIMSET bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:8]**

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

**CLAIM, bits [7:0]**

Set CLAIM tag bits. RAO.

Writing a 1 to one of these bits sets the corresponding CLAIM tag bit to 1. This is an indirect write to the CLAIM tag bits. A single write operation can set multiple CLAIM tag bits to 1.

Writing 0 to one of these bits has no effect.

A cold reset clears the CLAIM tag bits to 0.

### Accessing the DBGCLAIMSET:

To access the DBGCLAIMSET:

```
MRC p14,0,<Rt>,c7,c8,6 ; Read DBGCLAIMSET into Rt
MCR p14,0,<Rt>,c7,c8,6 ; Write Rt to DBGCLAIMSET
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0111 | 1000 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.7 DBGDCCINT, DCC Interrupt Enable Register

The DBGDCCINT characteristics are:

**Purpose**

Enables interrupt requests to be signaled based on the DCC status flags.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

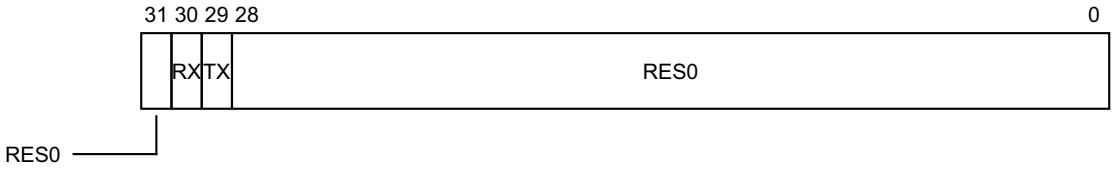
AArch32 System register DBGDCCINT is architecturally mapped to AArch64 System register MDCCINT_EL1.

**Attributes**

DBGDCCINT is a 32-bit register.

## Field descriptions

The DBGDCCINT bit assignments are:




**Bit [31]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**RX, bit [30]**

DCC interrupt request enable control for DTRRX. Enables a common COMMIRQ interrupt request to be signaled based on the DCC status flags.

`0` No interrupt request generated by DTRRX.

`1` Interrupt request will be generated on RXfull == 1.

If legacy COMMRX and COMMTX signals are implemented, then these are not affected by the value of this bit.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

**TX, bit [29]**

DCC interrupt request enable control for DTRTX. Enables a common COMMIRQ interrupt request to be signaled based on the DCC status flags.

`0` No interrupt request generated by DTRTX.

`1` Interrupt request will be generated on TXfull == 0.

If legacy COMMRX and COMMTX signals are implemented, then these are not affected by the value of this bit.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

**Bits [28:0]**

Reserved, RES0.

## Accessing the DBGDCCINT:

To access the DBGDCCINT:

```
MRC p14,0,<Rt>,c0,c2,0 ; Read DBGDCCINT into Rt
MCR p14,0,<Rt>,c0,c2,0 ; Write Rt to DBGDCCINT
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.8 DBGDEVID, Debug Device ID register 0

The DBGDEVID characteristics are:

**Purpose**

Adds to the information given by the DBGDIDR by describing other features of the debug implementation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 read accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

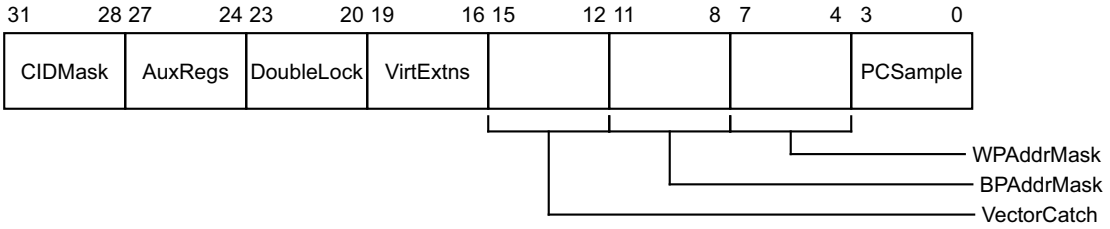
This register is required in all implementations.

**Attributes**

DBGDEVID is a 32-bit register.

## Field descriptions

The DBGDEVID bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**CIDMask, bits [31:28]**

Indicates the level of support for the Context ID matching breakpoint masking capability. Permitted values of this field are:

0000 Context ID masking is not implemented.

0001 Context ID masking is implemented.

All other values are reserved. The value of this for ARMv8 is 0000.

**AuxRegs, bits [27:24]**

Indicates support for Auxiliary registers. Permitted values for this field are:

0000 None supported.

0001 Support for External Debug Auxiliary Control Register, EDACR.

All other values are reserved.

**DoubleLock, bits [23:20]**

Indicates the presence of the DBGOSDLR, OS Double Lock Register. Permitted values of this field are:

0000 The DBGOSDLR is not present.

0001 The DBGOSDLR is present.

All other values are reserved. The value of this for ARMv8 is 0001.

**VirtExtns, bits [19:16]**

Indicates whether EL2 is implemented. Permitted values of this field are:

0000 EL2 is not implemented.

0001 EL2 is implemented.

All other values are reserved.

**VectorCatch, bits [15:12]**

Defines the form of Vector catch debug event implemented. Permitted values of this field are:

0000 Address matching vector catch debug event implemented.

0001 Exception matching vector catch debug event implemented.

All other values are reserved.

**BPAddrMask, bits [11:8]**

Indicates the level of support for the IVA matching breakpoint masking capability. Permitted values of this field are:

0000 Breakpoint address masking may be implemented. If not implemented, DBGBCR<n>[28:24] is RAZ/WI.

0001 Breakpoint address masking is implemented.

1111 Breakpoint address masking is not implemented. DBGBCR<n>[28:24] is RES0.

All other values are reserved. The value of this for ARMv8 is 1111.

**WPAddrMask, bits [7:4]**

Indicates the level of support for the data VA matching watchpoint masking capability. Permitted values of this field are:

0000 Watchpoint address masking may be implemented. If not implemented, DBGWCR<n>.MASK (Address mask) is RAZ/WI.

0001 Watchpoint address masking is implemented.

1111 Watchpoint address masking is not implemented. DBGWCR<n>.MASK (Address mask) is RES0.

All other values are reserved. The value of this for ARMv8 is 0001.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**PCSample, bits [3:0]**

Indicates the level of Sample-based profiling support using external debug registers 40 through 43. Permitted values of this field in ARMv8 are:

`0000` Architecture-defined form of Sample-based profiling not implemented.

`0010` EDPCSR and EDCIDSR are implemented (only permitted if EL3 and EL2 are not implemented).

`0011` EDPCSR, EDCIDSR, and EDVIDSR are implemented.

All other values are reserved.

## Accessing the DBGDEVID:

To access the DBGDEVID:

```
MRC p14,0,<Rt>,c7,c2,7 ; Read DBGDEVID into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0111 | 0010 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.9 DBGDEVID1, Debug Device ID register 1

The DBGDEVID1 characteristics are:

**Purpose**

Adds to the information given by the DBGDIDR by describing other features of the debug implementation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 read accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

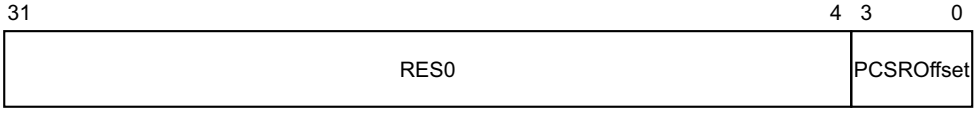
This register is required in all implementations.

**Attributes**

DBGDEVID1 is a 32-bit register.

## Field descriptions

The DBGDEVID1 bit assignments are:




**Bits [31:4]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**PCSROffset, bits [3:0]**

This field indicates the offset applied to PC samples returned by reads of EDPCSR. Permitted values of this field in ARMv8 are:

`0000` EDPCSR not implemented.

`0010` EDPCSR implemented. Samples have no offset applied and do not sample the instruction set state in AArch32 state.

**Note**

In ARMv7, a PCSROffset value of `0000` has an alternative meaning that EDPCSR is implemented and returns values that have an offset applied and indicate the Instruction set state. This implementation option is not permitted in ARMv8.

## Accessing the DBGDEVID1:

To access the DBGDEVID1:

```
MRC p14,0,<Rt>,c7,c1,7 ; Read DBGDEVID1 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0111 | 0001 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.10 DBGDEVID2, Debug Device ID register 2

The DBGDEVID2 characteristics are:

**Purpose**

Reserved for future descriptions of features of the debug implementation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 read accesses to this register are trapped to EL3.

**Configurations**

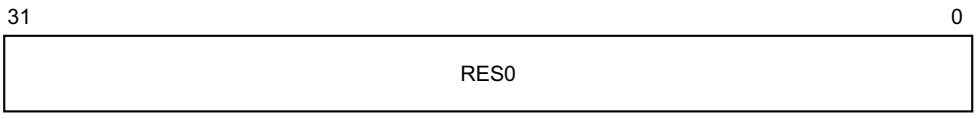
There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

DBGDEVID2 is a 32-bit register.

#### Field descriptions

The DBGDEVID2 bit assignments are:




**Bits [31:0]**

Reserved, RES0.

#### Accessing the DBGDEVID2:

To access the DBGDEVID2:

```
MRC p14,0,<Rt>,c7,c0,7 ; Read DBGDEVID2 into Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0111 | 0000 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.11 DBGDIDR, Debug ID Register

The DBGDIDR characteristics are:

**Purpose**

Specifies which version of the Debug architecture is implemented, and some features of the debug implementation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RO |

ARM deprecates any access to this register from EL0.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If DBGDSCRext.UDCCdis==1, PL0 read accesses to this register are trapped to Undefined mode.

If HDCR.TDA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL0, PL1, and PL2 read accesses to this register are trapped to EL3.

If MDSCR_EL1.TDCC==1, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

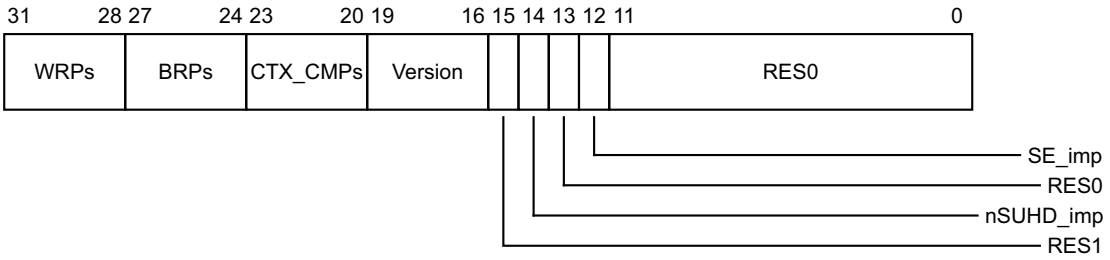
**Attributes**

DBGDIDR is a 32-bit register.

#### Field descriptions

The DBGDIDR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**WRPs, bits [31:28]**

The number of watchpoints implemented, minus 1.

Permitted values of this field are from `0b0001` for 2 implemented watchpoints, to `0b1111` for 16 implemented watchpoints.

The value of `0b0000` is reserved.

If AArch64 is implemented, this field has the same value as ID_AA64DFR0_EL1.WRPs.

**BRPs, bits [27:24]**

The number of breakpoints implemented, minus 1.

Permitted values of this field are from `0b0001` for 2 implemented breakpoint, to `0b1111` for 16 implemented breakpoints.

The value of `0b0000` is reserved.

If AArch64 is implemented, this field has the same value as ID_AA64DFR0_EL1.BRPs.

**CTX_CMPs, bits [23:20]**

The number of breakpoints that can be used for Context matching, minus 1.

Permitted values of this field are from `0b0000` for 1 Context matching breakpoint, to `0b1111` for 16 Context matching breakpoints.

The Context matching breakpoints must be the highest addressed breakpoints. For example, if six breakpoints are implemented and two are Context matching breakpoints, they must be breakpoints 4 and 5.

If AArch64 is implemented, this field has the same value as ID_AA64DFR0_EL1.CTX_CMPs.

**Version, bits [19:16]**

The Debug architecture version. Defined values are:

`0001` ARMv6, v6 Debug architecture

`0010` ARMv6, v6.1 Debug architecture

`0011` ARMv7, v7 Debug architecture, with baseline CP14 registers implemented

`0100` ARMv7, v7 Debug architecture, with all CP14 registers implemented

`0101` ARMv7, v7.1 Debug architecture.

`0110` ARMv8, v8 Debug architecture.

All other values are reserved.

**Bit [15]**

Reserved, RES1.

**nSUHD_imp, bit [14]**

In ARMv7-A, was Secure User Halting Debug not implemented.

The value of this bit must match the value of the SE_imp bit.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bit [13]**

Reserved, RES0.

**SE_imp, bit [12]**

EL3 implemented. The meanings of the values of this bit are:

0 EL3 not implemented.

1 EL3 implemented.

The value of this bit must match the value of the nSUHD_imp bit.

**Bits [11:0]**

Reserved, RES0.

## Accessing the DBGDIDR:

To access the DBGDIDR:

```
MRC p14,0,<Rt>,c0,c0,0 ; Read DBGDIDR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.12 DBGDRAR, Debug ROM Address Register

The DBGDRAR characteristics are:

**Purpose**

Defines the base physical address of a 4KB-aligned memory-mapped debug component, usually a ROM table that locates and describes the memory-mapped debug components in the system. ARMv8 deprecates any use of this register.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RO |

ARMv8 deprecates any use of this register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If DBGDSCRext.UDCCdis==1, PL0 read accesses to this register are trapped to Undefined mode.

If HDCR.TDRA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDRA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDSCR_EL1.TDCC==1, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGDRAR is architecturally mapped to AArch64 System register MDRAR_EL1.

If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

**Attributes**

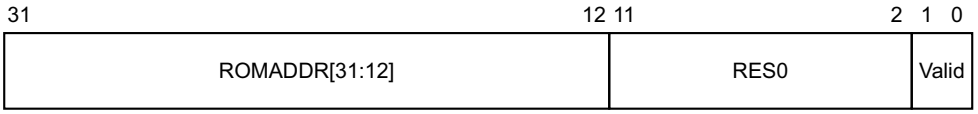
DBGDRAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, bits [31:0] are read.

#### Field descriptions

The DBGDRAR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

***When accessing as a 32-bit register:***




**ROMADDR[31:12], bits [31:12]**

Bits[31:12] of the ROM table physical address. Bits [11:0] of the address are zero.

**Bits [11:2]**

Reserved, RES0.

**Valid, bits [1:0]**

This field indicates whether the ROM Table address is valid. The permitted values of this field are:

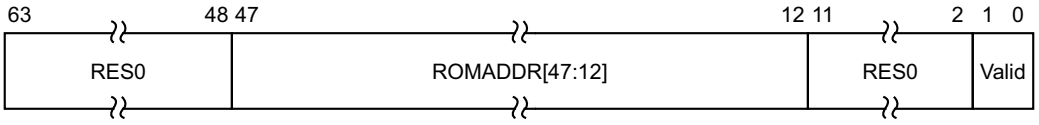
`00` ROM Table address is not valid

`11` ROM Table address is valid.

Other values are reserved.

If no memory-mapped debug components are implemented, this field is RES0.

***When accessing as a 64-bit register:***




**Bits [63:48]**

Reserved, RES0.

**ROMADDR[47:12], bits [47:12]**

Bits[47:12] of the ROM table physical address.

If the physical address size in bits (PAsize) is less than 48 then the register bits corresponding to ROMADDR [47:PAsize] are RES0.

Bits [11:0] of the ROM table physical address are zero.

ARM strongly recommends that bits ROMADDR[(PAsize-1):32] are zero in any system that supports AArch32 at the highest implemented Exception level.

**Bits [11:2]**

Reserved, RES0.

**Valid, bits [1:0]**

This field indicates whether the ROM Table address is valid. The permitted values of this field are:

`00` ROM Table address is not valid

`11` ROM Table address is valid.

Other values are reserved.

If no memory-mapped debug components are implemented, this field is RES0.

## Accessing the DBGDRAR:

To access the DBGDRAR when accessing as a 32-bit register:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRC p14,0,<Rt>,c1,c0,0 ; Read DBGDRAR[31:0] into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0001 | 0000 | 000 |

To access the DBGDRAR when accessing as a 64-bit register:

```
MRRC p14,0,<Rt>,<Rt2>,c1 ; Read DBGDRAR[31:0] into Rt and DBGDRAR[63:32] into Rt2
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1110 | 0000 | 0001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.13 DBGDSAR, Debug Self Address Register

The DBGDSAR characteristics are:

**Purpose**

In earlier versions of the ARM Architecture, this register defines the offset from the base address defined in DBGDRAR of the physical base address of the debug registers for the PE. ARMv8 deprecates any use of this register.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RO |

ARMv8 deprecates any use of this register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If DBGDSCRext.UDCCdis==1, PL0 read accesses to this register are trapped to Undefined mode.

If HDCR.TDRA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDRA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL0, PL1, and PL2 read accesses to this register are trapped to EL3.

If MDSCR_EL1.TDCC==1, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

**Attributes**

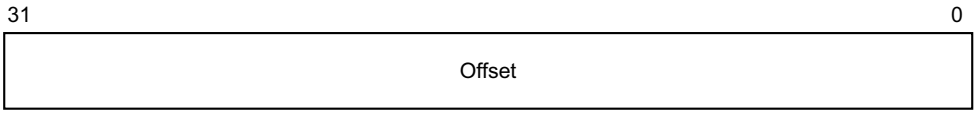
DBGDSAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, bits [31:0] are read.

#### Field descriptions

The DBGDSAR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

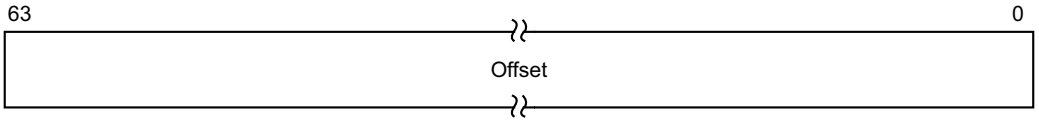
***When accessing as a 32-bit register:***




**Offset, bits [31:0]**

This register value is RAZ.

***When accessing as a 64-bit register:***




**Offset, bits [63:0]**

This register value is RAZ.

## Accessing the DBGDSAR:

To access the DBGDSAR when accessing as a 32-bit register:

```
MRC p14,0,<Rt>,c2,c0,0 ; Read DBGDSAR[31:0] into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0010 | 0000 | 000 |

To access the DBGDSAR when accessing as a 64-bit register:

```
MRRC p14,0,<Rt>,<Rt2>,c2 ; Read DBGDSAR[31:0] into Rt and DBGDSAR[63:32] into Rt2
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1110 | 0000 | 0010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.14 DBGDSCRext, Debug Status and Control Register, External View

The DBGDSCRext characteristics are:

**Purpose**

Main control register for the debug implementation.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGDSCRext is architecturally mapped to AArch64 System register MDSCR_EL1.

This register is required in all implementations.

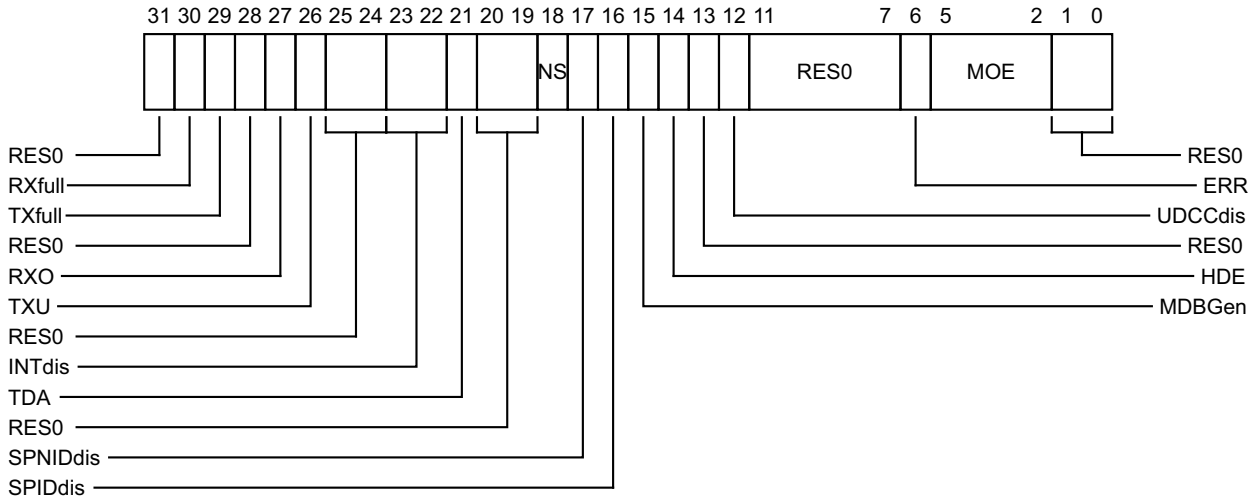
**Attributes**

DBGDSCRext is a 32-bit register.

#### Field descriptions

The DBGDSCRext bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bit [31]**

Reserved, RES0.

**RXfull, bit [30]**

DTRRX full. Used for save/restore of EDSCR.RXfull.

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW.

ARM deprecates use of this bit other than for save/restore. Use DBGDSCRint to access the DTRRX full status.

**TXfull, bit [29]**

DTRTX full. Used for save/restore of EDSCR.TXfull.

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW.

ARM deprecates use of this bit other than for save/restore. Use DBGDSCRint to access the DTRTX full status.

**Bit [28]**

Reserved, RES0.

**RXO, bit [27]**

Used for save/restore of EDSCR.RXO.

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW.

**TXU, bit [26]**

Used for save/restore of EDSCR.TXU.

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW.

**Bits [25:24]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**INTdis, bits [23:22]**

Used for save/restore of EDSCR.INTdis.

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this field is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this field is RW.

**TDA, bit [21]**

Used for save/restore of EDSCR.TDA.

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW.

**Bits [20:19]**

Reserved, RES0.

**NS, bit [18]**

Non-secure status. Returns the inverse of IsSecure(). This bit is RO.

ARM deprecates use of this field.

**SPNIDdis, bit [17]**

Secure privileged profiling disabled status bit. This bit is RO and reflects the value of ProfilingProhibited(TRUE,EL1). Permitted values are:

`0` Profiling allowed in Secure privileged modes.

`1` Profiling prohibited in Secure privileged modes.

ARM deprecates use of this field.

**SPIDdis, bit [16]**

Secure privileged AArch32 invasive self-hosted debug disabled status bit. This bit is RO and depends on the value of SDCR.SPD and the pseudocode function AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled(). Permitted values are:

`0` Self-hosted debug enabled in Secure privileged AArch32 modes.

`1` Self-hosted debug disabled in Secure privileged AArch32 modes.

This bit reads as 1 if any of the following is true and reads as 0 otherwise:

- SDCR.SPD has the value `10`.
- SDCR.SPD has the value `00` and AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled() returns FALSE.

ARM deprecates use of this field.

**MDBGen, bit [15]**

Monitor debug events enable. Enable Breakpoint, Watchpoint, and Vector catch debug exceptions.

`0` Breakpoint, Watchpoint, and Vector catch debug exceptions disabled.

`1` Breakpoint, Watchpoint, and Vector catch debug exceptions enabled.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

**HDE, bit [14]**

Used for save/restore of EDSCR.HDE.

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bit [13]**

Reserved, RES0.

**UDCCdis, bit [12]**

Traps PL0 accesses to the DCC registers to Undefined mode.

0 PL0 accesses to the DCC registers are not trapped to Undefined mode.

1 PL0 accesses to the DBGDSCRint, DBGDTRRXint, DBGDTRTXint, DBGDIDR, DBGDSAR, and DBGDRAR are trapped to Undefined mode.

**Note**

All accesses to these registers are trapped, including LDC and STC accesses to DBGDTRTXint and DBGDTRRXint, and MRRC accesses to DBGDSAR and DBGDRAR.

Traps of PL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

**Bits [11:7]**

Reserved, RES0.

**ERR, bit [6]**

Used for save/restore of EDSCR.ERR.

When DBGOSLSR.OSLK == 0 (the OS lock is unlocked), this bit is RO. Software must treat it as UNKNOWN and use an SBZP policy for writes.

When DBGOSLSR.OSLK == 1 (the OS lock is locked), this bit is RW.

**MOE, bits [5:2]**

Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:

0001 Breakpoint

0011 Software breakpoint (BKPT) instruction

0101 Vector catch

1010 Watchpoint

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bits [1:0]**

Reserved, RES0.

## Accessing the DBGDSCRext:

To access the DBGDSCRext:

```
MRC p14,0,<Rt>,c0,c2,2 ; Read DBGDSCRext into Rt
MCR p14,0,<Rt>,c0,c2,2 ; Write Rt to DBGDSCRext
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0010 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.15 DBGDSCRint, Debug Status and Control Register, Internal View

The DBGDSCRint characteristics are:

**Purpose**

Main control register for the debug implementation. This is an internal, read-only view.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If DBGDSCRext.UDCCdis==1, PL0 read accesses to this register are trapped to Undefined mode.

If HDCR.TDA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL0, PL1, and PL2 read accesses to this register are trapped to EL3.

If MDSCR_EL1.TDCC==1, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGDSCRint is architecturally mapped to AArch64 System register MDCCSR_EL0.

This register is required in all implementations.

DBGDSCRint.{NS, SPNIDdis, SPIDdis, MDBGen, UDCCdis, MOE} are UNKNOWN when the register is accessed at EL0. However, although these values are not accessible at EL0 by instructions that are neither UNPREDICTABLE nor return UNKNOWN values, it is permissible for an implementation to return the values of DBGDSCRext.{NS, SPNIDdis, SPIDdis, MDBGen, UDCCdis, MOE} for these fields at EL0.

It is also permissible for an implementation to return the same values as defined for a read of DBGDSCRint at EL1 or above. (This is the case even if the implementation does not support AArch32 at EL1 or above.)

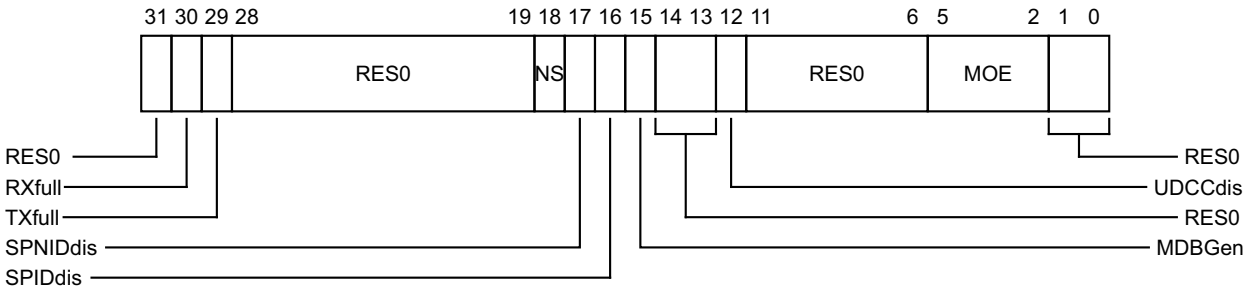
**Attributes**

DBGDSCRint is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The DBGDSCRint bit assignments are:




**Bit [31]**

Reserved, RES0.

**RXfull, bit [30]**

DTRRX full. Read-only view of the equivalent bit in the EDSCR.

**TXfull, bit [29]**

DTRTX full. Read-only view of the equivalent bit in the EDSCR.

**Bits [28:19]**

Reserved, RES0.

**NS, bit [18]**

Non-secure status.

Read-only view of the equivalent bit in the DBGDSCRext. ARM deprecates use of this field.

**SPNIDdis, bit [17]**

Secure privileged non-invasive debug disable.

Read-only view of the equivalent bit in the DBGDSCRext. ARM deprecates use of this field.

**SPIDdis, bit [16]**

Secure privileged invasive debug disable.

Read-only view of the equivalent bit in the DBGDSCRext. ARM deprecates use of this field.

**MDBGen, bit [15]**

Monitor debug events enable.

Read-only view of the equivalent bit in the DBGDSCRext.

**Bits [14:13]**

Reserved, RES0.

**UDCCdis, bit [12]**

User mode access to Debug Communications Channel disable.

Read-only view of the equivalent bit in the DBGDSCRext. ARM deprecates use of this field.

**Bits [11:6]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**MOE, bits [5:2]**

Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:

`0001` Breakpoint

`0011` Software breakpoint (BKPT) instruction

`0101` Vector catch

`1010` Watchpoint

Read-only view of the equivalent bit in the DBGDSCRext.

**Bits [1:0]**

Reserved, RES0.

## Accessing the DBGDSCRint:

To access the DBGDSCRint:

```
MRC p14,0,<Rt>,c0,c1,0 ; Read DBGDSCRint into Rt, where Rt can be R0-R14 or APSR_nzcv. The last form writes bits[31:28] of the transferred value to the N, Z, C and V condition flags and is specified by setting the RT field of the encoding to 0b1111.
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.16 DBGDTRRXext, Debug OS Lock Data Transfer Register, Receive, External View

The DBGDTRRXext characteristics are:

**Purpose**

Used for save/restore of DBGDTRRXint. It is a component of the Debug Communications Channel.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

ARM deprecates reads and writes of DBGDTRRXext through the CP14 interface when the OS lock is unlocked.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

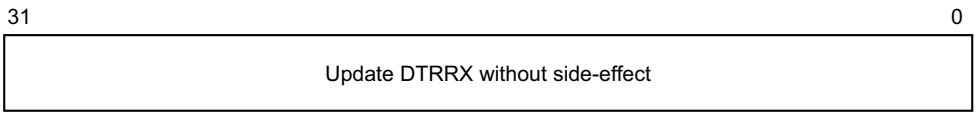
AArch32 System register DBGDTRRXext is architecturally mapped to AArch64 System register OSDTRRX_EL1.

**Attributes**

DBGDTRRXext is a 32-bit register.

## Field descriptions

The DBGDTRRXext bit assignments are:




**Bits [31:0]**

Update DTRRX without side-effect.

Writes to this register update the value in DTRRX and do not change RXfull.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

### Accessing the DBGDTRRXext:

To access the DBGDTRRXext:

```
MRC p14,0,<Rt>,c0,c0,2 ; Read DBGDTRRXext into Rt
MCR p14,0,<Rt>,c0,c0,2 ; Write Rt to DBGDTRRXext
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0000 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.17 DBGDTRRXint, Debug Data Transfer Register, Receive

The DBGDTRRXint characteristics are:

**Purpose**

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. It is a component of the Debug Communications Channel.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If DBGDSCRext.UDCCdis==1, PL0 read accesses to this register are trapped to Undefined mode.

If HDCR.TDA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL0, PL1, and PL2 read accesses to this register are trapped to EL3.

If MDSCR_EL1.TDCC==1, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGDTRRXint is architecturally mapped to AArch64 System register DBGDTRRX_EL0.

AArch32 System register DBGDTRRXint is architecturally mapped to External register DBGDTRRX_EL0.

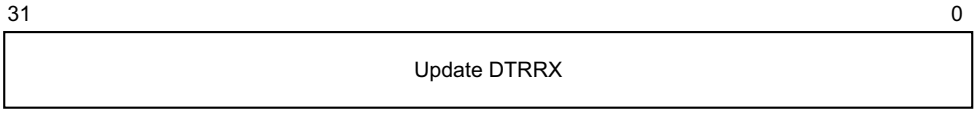
**Attributes**

DBGDTRRXint is a 32-bit register.

#### Field descriptions

The DBGDTRRXint bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:0]**

Update DTRRX.

If RXfull is set to 1, then reads of this register return the last value written to DTRRX and clear RXfull to 0.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

## Accessing the DBGDTRRXint:

To access the DBGDTRRXint:

```
MRC p14,0,<Rt>,c0,c5,0 ; Read DBGDTRRXint into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.18 DBGDTRTXext, Debug OS Lock Data Transfer Register, Transmit

The DBGDTRTXext characteristics are:

**Purpose**

Used for save/restore of DBGDTRTXint. It is a component of the Debug Communication Channel.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

ARM deprecates reads and writes of DBGDTRTXext through the CP14 interface when the OS lock is unlocked.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

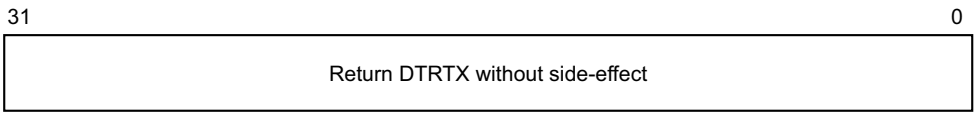
AArch32 System register DBGDTRTXext is architecturally mapped to AArch64 System register OSDTRTX_EL1.

**Attributes**

DBGDTRTXext is a 32-bit register.

## Field descriptions

The DBGDTRTXext bit assignments are:




**Bits [31:0]**

Return DTRTX without side-effect.

Reads of this register return the value in DTRTX and do not change TXfull.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

### Accessing the DBGDTRTXext:

To access the DBGDTRTXext:

```
MRC p14,0,<Rt>,c0,c3,2 ; Read DBGDTRTXext into Rt
MCR p14,0,<Rt>,c0,c3,2 ; Write Rt to DBGDTRTXext
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0011 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.19 DBGDTRTXint, Debug Data Transfer Register, Transmit

The DBGDTRTXint characteristics are:

**Purpose**

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. It is a component of the Debug Communication Channel.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-WO | Config-WO | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-WO | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If DBGDSCRext.UDCCdis==1, PL0 write accesses to this register are trapped to Undefined mode.

If HDCR.TDA==1, Non-secure EL0 and EL1 write accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL0 and EL1 write accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL0, PL1, and PL2 write accesses to this register are trapped to EL3.

If MDSCR_EL1.TDCC==1, PL0 write accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGDTRTXint is architecturally mapped to AArch64 System register DBGDTRTX_EL0.

AArch32 System register DBGDTRTXint is architecturally mapped to External register DBGDTRTX_EL0.

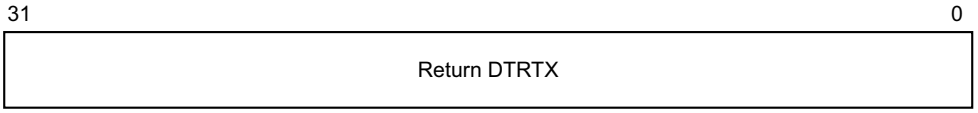
**Attributes**

DBGDTRTXint is a 32-bit register.

#### Field descriptions

The DBGDTRTXint bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:0]**

Return DTRTX.

If TXfull is set to 0, then writes of this register update the value in DTRTX and set TXfull to 1.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

## Accessing the DBGDTRTXint:

To access the DBGDTRTXint:

```
MCR p14,0,<Rt>,c0,c5,0 ; Write Rt to DBGDTRTXint
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.20 DBGOSDLR, Debug OS Double Lock Register

The DBGOSDLR characteristics are:

**Purpose**

Locks out the external debug interface.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDOSA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDOSA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDOSA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

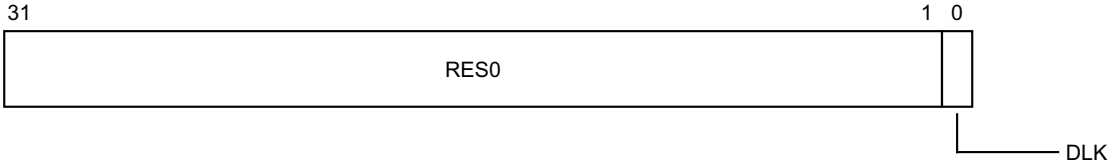
AArch32 System register DBGOSDLR is architecturally mapped to AArch64 System register OSDLR_EL1.

**Attributes**

DBGOSDLR is a 32-bit register.

## Field descriptions

The DBGOSDLR bit assignments are:




**Bits [31:1]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**DLK, bit [0]**

OS Double Lock control bit. Possible values are:

`0` OS Double Lock unlocked.

`1` OS Double Lock locked, if DBGPRCR.CORENPDRQ (Core no powerdown request) bit is set to 0 and the PE is in Non-debug state.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

## Accessing the DBGOSDLR:

To access the DBGOSDLR:

```
MRC p14,0,<Rt>,c1,c3,4 ; Read DBGOSDLR into Rt
MCR p14,0,<Rt>,c1,c3,4 ; Write Rt to DBGOSDLR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0001 | 0011 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.21 DBGOSECCR, Debug OS Lock Exception Catch Control Register

The DBGOSECCR characteristics are:

**Purpose**

Provides a mechanism for an operating system to access the contents of EDECCR that are otherwise invisible to software, so it can save/restore the contents of EDECCR over powerdown on behalf of the external debugger.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGOSECCR is architecturally mapped to AArch64 System register OSECCR_EL1.

AArch32 System register DBGOSECCR is architecturally mapped to External register EDECCR.

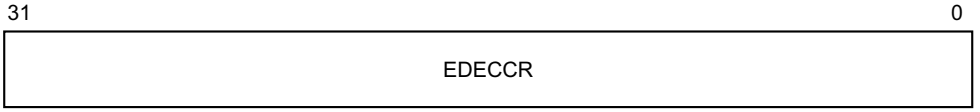
**Attributes**

DBGOSECCR is a 32-bit register.

#### Field descriptions

The DBGOSECCR bit assignments are:

***When OSLSR.OSLK==1:***





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EDECCR, bits [31:0]**

Used for save/restore to EDECCR over powerdown.

## Accessing the DBGOSECCR:

To access the DBGOSECCR:

```
MRC p14,0,<Rt>,c0,c6,2 ; Read DBGOSECCR into Rt
MCR p14,0,<Rt>,c0,c6,2 ; Write Rt to DBGOSECCR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0110 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.22 DBGOSLAR, Debug OS Lock Access Register

The DBGOSLAR characteristics are:

**Purpose**

Provides a lock for the debug registers. The OS lock also disables some Software debug events.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | WO | WO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDOSA==1, Non-secure EL1 write accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDOSA==1, Non-secure EL1 write accesses to this register are trapped to EL2 using AArch64.

If MDCR_EL3.TDOSA==1, PL1 and PL2 write accesses to this register are trapped to EL3 using AArch64.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGOSLAR is architecturally mapped to AArch64 System register OSLAR_EL1.

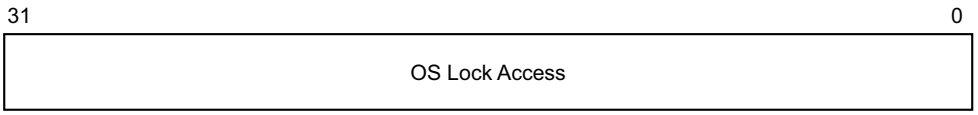
AArch32 System register DBGOSLAR is architecturally mapped to External register OSLAR_EL1.

**Attributes**

DBGOSLAR is a 32-bit register.

#### Field descriptions

The DBGOSLAR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:0]**

OS Lock Access. Writing the value `0xC5ACCE55` to the DBGOSLAR sets the OS lock to 1. Writing any other value sets the OS lock to 0.

Use DBGOSLSR.OSLK to check the current status of the lock.

### Accessing the DBGOSLAR:

To access the DBGOSLAR:

```
MCR p14,0,<Rt>,c1,c0,4 ; Write Rt to DBGOSLAR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0001 | 0000 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.23 DBGOSLSR, Debug OS Lock Status Register

The DBGOSLSR characteristics are:

**Purpose**

Provides status information for the OS lock.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDOSA==1, Non-secure EL1 read accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDOSA==1, Non-secure EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TDOSA==1, PL1 and PL2 read accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGOSLSR is architecturally mapped to AArch64 System register OSLSR_EL1.

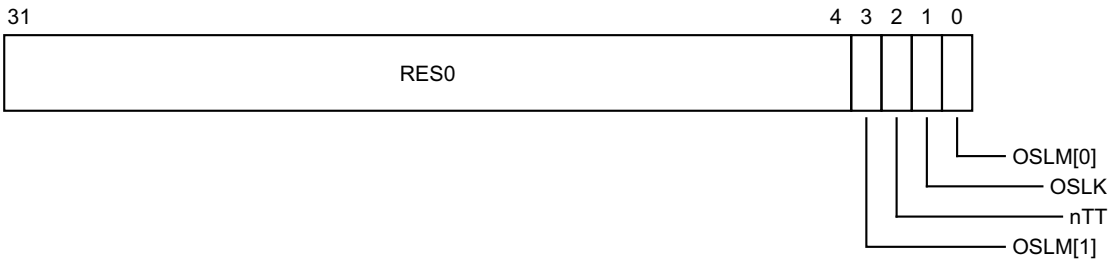
The OS lock status is also visible in the external debug interface through EDPRSR.

**Attributes**

DBGOSLSR is a 32-bit register.

#### Field descriptions

The DBGOSLSR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:4]**

Reserved, RES0.

**OSLM[1], bit [3]**

See below for description of the OSLM field.

**nTT, bit [2]**

Not 32-bit access. This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

**OSLK, bit [1]**

OS Lock Status. The possible values are:

0 OS lock unlocked.

1 OS lock locked.

The OS lock is locked and unlocked by writing to the OS Lock Access Register.

When this register has an architecturally-defined reset value, this field resets to 1.

This field resets to its defined reset value on Cold reset.

**OSLM[0], bit [0]**

OS lock model implemented. Identifies the form of OS save and restore mechanism implemented. In ARMv8 these bits are as follows:

10 OS lock implemented. DBGOSSRR not implemented.

All other values are reserved.

### Accessing the DBGOSLSR:

To access the DBGOSLSR:

```
MRC p14,0,<Rt>,c1,c1,4 ; Read DBGOSLSR into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0001 | 0001 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.24 DBGPRCR, Debug Power Control Register

The DBGPRCR characteristics are:

**Purpose**

Controls behavior of the PE on powerdown request.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDOSA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDOSA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDOSA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGPRCR is architecturally mapped to AArch64 System register DBGPRCR_EL1.

Bit [0] of this register is mapped to EDPRCR.CORENPDRQ, bit [0] of the external view of this register.

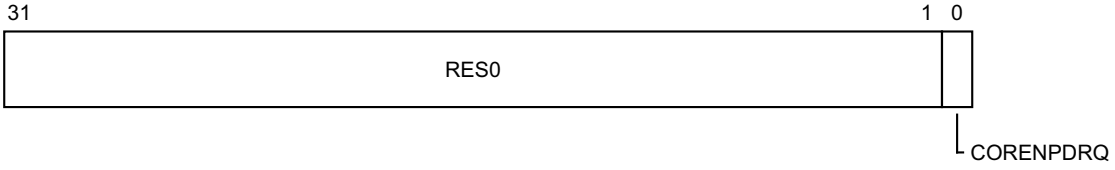
The other bits in these registers are not mapped to each other.

**Attributes**

DBGPRCR is a 32-bit register.

#### Field descriptions

The DBGPRCR bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:1]**

Reserved, RES0.

**CORENPDRQ, bit [0]**

Core no powerdown request. Requests emulation of powerdown. Possible values of this bit are:

0 On a powerdown request, the system powers down the Core power domain.

1 On a powerdown request, the system emulates powerdown of the Core power domain. In this emulation mode the Core power domain is not actually powered down.

Writes to this bit are permitted regardless of the state of the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request Core no powerdown regardless of whether invasive debug is permitted.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, this field resets to the value of EDPRCR.COREPURQ.

This field resets to its defined reset value on Cold reset.

## Accessing the DBGPRCR:

To access the DBGPRCR:

```
MRC p14,0,<Rt>,c1,c4,4 ; Read DBGPRCR into Rt
MCR p14,0,<Rt>,c1,c4,4 ; Write Rt to DBGPRCR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0001 | 0100 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.25 DBGVCR, Debug Vector Catch Register

The DBGVCR characteristics are:

**Purpose**

Controls Vector catch debug events.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGVCR is architecturally mapped to AArch64 System register DBGVCR32_EL2.

This register is required in all implementations.

**Attributes**

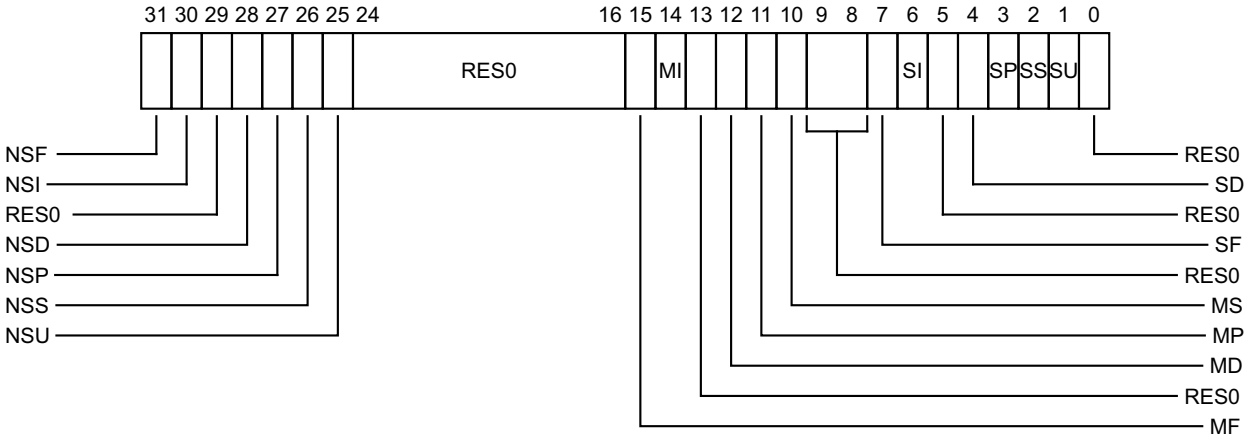
DBGVCR is a 32-bit register.

#### Field descriptions

The DBGVCR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

***When EL3 implemented and using AArch32:***




**NSF, bit [31]**

FIQ vector catch enable in Non-secure state.

The exception vector offset is `0x1C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**NSI, bit [30]**

IRQ vector catch enable in Non-secure state.

The exception vector offset is `0x18`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [29]**

Reserved, RES0.

**NSD, bit [28]**

Data Abort vector catch enable in Non-secure state.

The exception vector offset is `0x10`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**NSP, bit [27]**

Prefetch Abort vector catch enable in Non-secure state.

The exception vector offset is `0x0C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**NSS, bit [26]**

Supervisor Call (SVC) vector catch enable in Non-secure state.

The exception vector offset is `0x08`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**NSU, bit [25]**

Undefined Instruction vector catch enable in Non-secure state.

The exception vector offset is `0x04`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bits [24:16]**

Reserved, RES0.

**MF, bit [15]**

FIQ vector catch enable in Monitor mode.

The exception vector offset is `0x1C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**MI, bit [14]**

IRQ vector catch enable in Monitor mode.

The exception vector offset is `0x18`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [13]**

Reserved, RES0.

**MD, bit [12]**

Data Abort vector catch enable in Monitor mode.

The exception vector offset is `0x10`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**MP, bit [11]**

Prefetch Abort vector catch enable in Monitor mode.

The exception vector offset is `0x0C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**MS, bit [10]**

Secure Monitor Call (SMC) vector catch enable in Monitor mode.

The exception vector offset is `0x08`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [9:8]**

Reserved, RES0.

**SF, bit [7]**

FIQ vector catch enable in Secure state.

The exception vector offset is `0x1C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SI, bit [6]**

IRQ vector catch enable in Secure state.

The exception vector offset is `0x18`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [5]**

Reserved, RES0.

**SD, bit [4]**

Data Abort vector catch enable in Secure state.

The exception vector offset is `0x10`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SP, bit [3]**

Prefetch Abort vector catch enable in Secure state.

The exception vector offset is `0x0C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SS, bit [2]**

Supervisor Call (SVC) vector catch enable in Secure state.

The exception vector offset is `0x08`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SU, bit [1]**

Undefined Instruction vector catch enable in Secure state.

The exception vector offset is `0x04`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

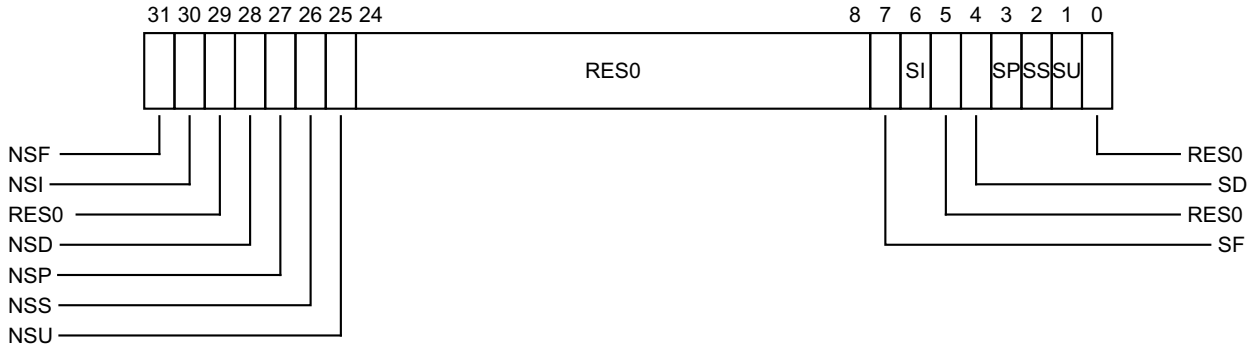
This field resets to its defined reset value on Warm reset.

**Bit [0]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

***When EL3 implemented and using AArch64:***




**NSF, bit [31]**

FIQ vector catch enable in Non-secure state.

The exception vector offset is `0x1C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**NSI, bit [30]**

IRQ vector catch enable in Non-secure state.

The exception vector offset is `0x18`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [29]**

Reserved, RES0.

**NSD, bit [28]**

Data Abort vector catch enable in Non-secure state.

The exception vector offset is `0x10`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**NSP, bit [27]**

Prefetch Abort vector catch enable in Non-secure state.

The exception vector offset is `0x0C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**NSS, bit [26]**

Supervisor Call (SVC) vector catch enable in Non-secure state.

The exception vector offset is `0x08`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**NSU, bit [25]**

Undefined Instruction vector catch enable in Non-secure state.

The exception vector offset is `0x04`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bits [24:8]**

Reserved, RES0.

**SF, bit [7]**

FIQ vector catch enable in Secure state.

The exception vector offset is `0x1C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SI, bit [6]**

IRQ vector catch enable in Secure state.

The exception vector offset is `0x18`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [5]**

Reserved, RES0.

**SD, bit [4]**

Data Abort vector catch enable in Secure state.

The exception vector offset is `0x10`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SP, bit [3]**

Prefetch Abort vector catch enable in Secure state.

The exception vector offset is `0x0C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SS, bit [2]**

Supervisor Call (SVC) vector catch enable in Secure state.

The exception vector offset is `0x08`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**SU, bit [1]**

Undefined Instruction vector catch enable in Secure state.

The exception vector offset is `0x04`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

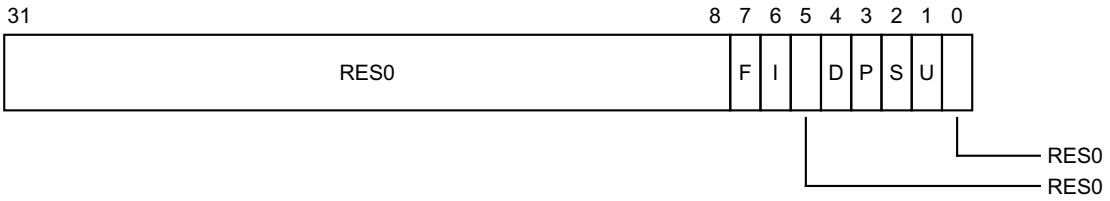
When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [0]**

Reserved, RES0.

***When EL3 not implemented:***




**Bits [31:8]**

Reserved, RES0.

**F, bit [7]**

FIQ vector catch enable.

The exception vector offset is `0x1C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**I, bit [6]**

IRQ vector catch enable.

The exception vector offset is `0x18`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [5]**

Reserved, RES0.

**D, bit [4]**

Data Abort vector catch enable.

The exception vector offset is `0x10`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**P, bit [3]**

Prefetch Abort vector catch enable.

The exception vector offset `0x0C`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**S, bit [2]**

Supervisor Call (SVC) vector catch enable.

The exception vector offset is `0x08`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**U, bit [1]**

Undefined Instruction vector catch enable.

The exception vector offset is `0x04`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**Bit [0]**

Reserved, RES0.

## Accessing the DBGVCR:

To access the DBGVCR:

```
MRC p14,0,<Rt>,c0,c7,0 ; Read DBGVCR into Rt
MCR p14,0,<Rt>,c0,c7,0 ; Write Rt to DBGVCR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0111 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.26 DBGWCR<n>, Debug Watchpoint Control Registers, n = 0 - 15

The DBGWCR<n> characteristics are:

**Purpose**

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

When the E field is zero, all the other fields in the register are ignored.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see:

- *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state.
- *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.
- *Software Access debug event* on page H3-4881 for accesses to this register taken to Debug state.

Subject to the prioritization rules:

If EDSCR.TDA==1, OSLSR.OSLK==0, and halting is allowed, EL1 and EL2 accesses to this register generate a Software Access debug event.

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGWCR<n> is architecturally mapped to AArch64 System register DBGWCR<n>_EL1.

AArch32 System register DBGWCR<n> is architecturally mapped to External register DBGWCR<n>_EL1.

If breakpoint n is not implemented then this register is unallocated.

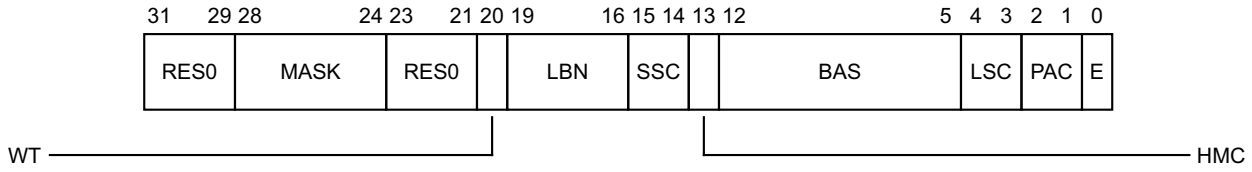
**Attributes**

DBGWCR<n> is a 32-bit register.

#### Field descriptions

The DBGWCR<n> bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:29]**

Reserved, RES0.

**MASK, bits [28:24]**

Address mask. Only objects up to 2GB can be watched using a single mask.

`00000` No mask.

`00001` Reserved.

`00010` Reserved.

If programmed with a reserved value, a watchpoint must behave as if either:

- MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCRn_EL1.
- The watchpoint is disabled.

Software must not rely on this property because the behavior of reserved values might change in a future revision of the architecture.

Other values mask the corresponding number of address bits, from `0b00011` masking 3 address bits (`0x00000007` mask for address) to `0b11111` masking 31 address bits (`0x7FFFFFFF` mask for address).

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [23:21]**

Reserved, RES0.

**WT, bit [20]**

Watchpoint type. Possible values are:

`0` Unlinked data address match.

`1` Linked data address match.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**LBN, bits [19:16]**

Linked breakpoint number. For Linked data address watchpoints, this specifies the index of the Context-matching breakpoint linked to.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**SSC, bits [15:14]**

Security state control. Determines the Security states under which a watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to its defined reset value on Cold reset.

**HMC, bit [13]**

Higher mode control. Determines the debug perspective for deciding when a watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**BAS, bits [12:5]**

Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n> is being watched.

| BAS | Description |
|---|---|
| xxxxxxx1 | Match byte at DBGWVR<n> |
| xxxxxx1x | Match byte at DBGWVR<n>+1 |
| xxxxx1xx | Match byte at DBGWVR<n>+2 |
| xxxx1xxx | Match byte at DBGWVR<n>+3 |

In cases where DBGWVR<n> addresses a double-word:

| BAS | Description, if DBGWVR<n>[2] == 0 |
|---|---|
| xxx1xxxx | Match byte at DBGWVR<n>+4 |
| xx1xxxxx | Match byte at DBGWVR<n>+5 |
| x1xxxxxx | Match byte at DBGWVR<n>+6 |
| 1xxxxxxx | Match byte at DBGWVR<n>+7 |

If DBGWVR<n>[2] == 1, only BAS[3:0] are used and BAS[7:4] are ignored. ARM deprecates setting DBGWVR<n>[2] == 1.

The valid values for BAS are `0b0000000`, or a binary number all of whose set bits are contiguous. All other values are reserved and must not be used by software.

If BAS is zero, no bytes are watched by this watchpoint.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**LSC, bits [4:3]**

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

`01` Match instructions that load from a watchpointed address.

`10` Match instructions that store to a watchpointed address.

`11` Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**PAC, bits [2:1]**

Privilege of access control. Determines the Exception level or levels at which a watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**E, bit [0]**

Enable watchpoint n. Possible values are:

`0` Watchpoint disabled.

`1` Watchpoint enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

### Accessing the DBGWCR<n>:

To access the DBGWCR<n>:

```
MRC p14,0,<Rt>,c0,<CRm>,7 ; Read DBGWCR<n> into Rt, where n is in the range 0 to 15
MCR p14,0,<Rt>,c0,<CRm>,7 ; Write Rt to DBGWCR<n>, where n is in the range 0 to 15
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | n<3:0> | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.27 DBGWFAR, Debug Watchpoint Fault Address Register

The DBGWFAR characteristics are:

**Purpose**

Previously returned information about the address of the instruction that accessed a watchpointed address. Is now deprecated and RES0.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TDA==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

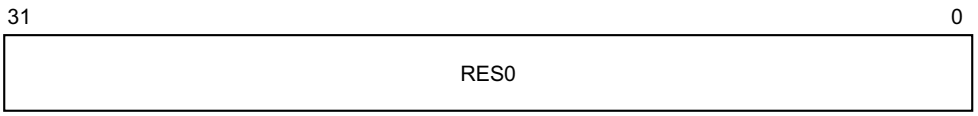
There is one instance of this register that is used in both Secure and Non-secure states.

**Attributes**

DBGWFAR is a 32-bit register.

#### Field descriptions

The DBGWFAR bit assignments are:




**Bits [31:0]**

Reserved, RES0.

#### Accessing the DBGWFAR:

To access the DBGWFAR:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRC p14,0,<Rt>,c0,c6,0 ; Read DBGWFAR into Rt
MCR p14,0,<Rt>,c0,c6,0 ; Write Rt to DBGWFAR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | 0110 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.28 DBGWVR<n>, Debug Watchpoint Value Registers, n = 0 - 15

The DBGWVR<n> characteristics are:

**Purpose**

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see:

- *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state.
- *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.
- *Software Access debug event* on page H3-4881 for accesses to this register taken to Debug state.

Subject to the prioritization rules:

If EDSCR.TDA==1, OSLSR.OSLK==0, and halting is allowed, EL1 and EL2 accesses to this register generate a Software Access debug event.

If MDCR_EL2.TDA==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DBGWVR<n> is architecturally mapped to AArch64 System register DBGWVR<n>_EL1[31:0].

AArch32 System register DBGWVR<n> is architecturally mapped to External register DBGWVR<n>_EL1[31:0].

If breakpoint n is not implemented then this register is unallocated.

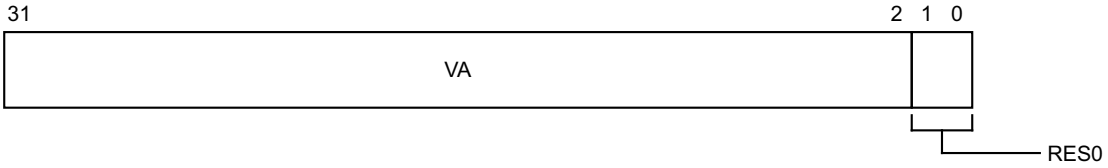
**Attributes**

DBGWVR<n> is a 32-bit register.

#### Field descriptions

The DBGWVR<n> bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**VA, bits [31:2]**

Bits[31:2] of the address value for comparison.

ARM deprecates setting DBGWVR<n>[2] == 1.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [1:0]**

Reserved, RES0.

## Accessing the DBGWVR<n>:

To access the DBGWVR<n>:

```
MRC p14,0,<Rt>,c0,<CRm>,6 ; Read DBGWVR<n> into Rt, where n is in the range 0 to 15
MCR p14,0,<Rt>,c0,<CRm>,6 ; Write Rt to DBGWVR<n>, where n is in the range 0 to 15
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1110 | 000 | 0000 | n<3:0> | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.29 DLR, Debug Link Register

The DLR characteristics are:

**Purpose**

In Debug state, holds the address to restart from.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| RW | RW | RW |

Access to this register is from Debug state only. During normal execution this register is unallocated.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

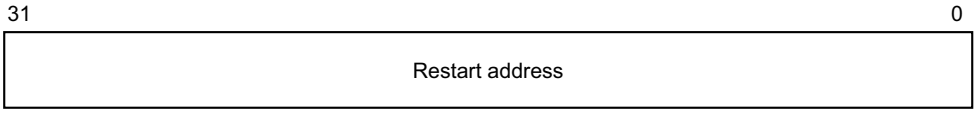
AArch32 System register DLR is architecturally mapped to AArch64 System register DLR_EL0[31:0].

**Attributes**

DLR is a 32-bit register.

#### Field descriptions

The DLR bit assignments are:




**Bits [31:0]**

Restart address.

#### Accessing the DLR:

To access the DLR:

```
MRC p15,3,<Rt>,c4,c5,1 ; Read DLR into Rt
MCR p15,3,<Rt>,c4,c5,1 ; Write Rt to DLR
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 011 | 0100 | 0101 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.30 DSPSR, Debug Saved Program Status Register

The DSPSR characteristics are:

**Purpose**

Holds the saved process state on entry to Debug state.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RW | RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| RW | RW | RW |

Access to this register is from Debug state only. During normal execution this register is unallocated.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register DSPSR is architecturally mapped to AArch64 System register DSPSR_EL0.

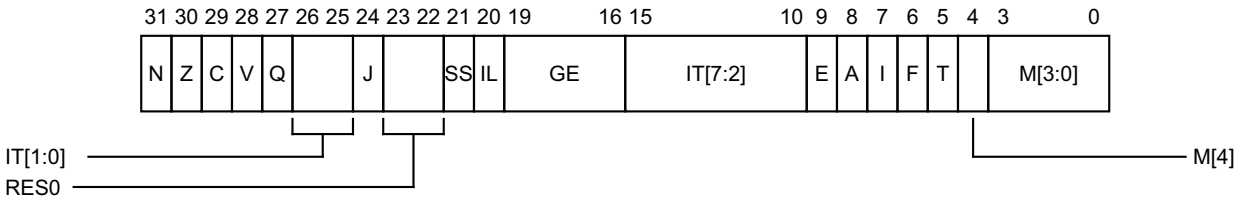
**Attributes**

DSPSR is a 32-bit register.

## Field descriptions

The DSPSR bit assignments are:

***When entering Debug state from AArch32 and exiting Debug state to AArch32:***




**N, bit [31]**

Set to the value of CPSR.N on entering Debug state, and copied to CPSR.N on exiting Debug state.

**Z, bit [30]**

Set to the value of CPSR.Z on entering Debug state, and copied to CPSR.Z on exiting Debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**C, bit [29]**

Set to the value of CPSR.C on entering Debug state, and copied to CPSR.C on exiting Debug state.

**V, bit [28]**

Set to the value of CPSR.V on entering Debug state, and copied to CPSR.V on exiting Debug state.

**Q, bit [27]**

Set to the value of CPSR.Q on entering Debug state, and copied to CPSR.Q on exiting Debug state.

**IT[1:0], bits [26:25]**

IT block state bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

**J, bit [24]**

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

**Bits [23:22]**

Reserved, RES0.

**SS, bit [21]**

Software step. Shows the value of PSTATE.SS immediately before Debug state was entered.

**IL, bit [20]**

Illegal Execution state bit. Shows the value of PSTATE.IL immediately before Debug state was entered.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**IT[7:2], bits [15:10]**

IT block state bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

**E, bit [9]**

Endianness state bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**A, bit [8]**

Asynchronous data abort mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**I, bit [7]**

IRQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**F, bit [6]**

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

**T, bit [5]**

T32 Instruction set state bit. Determines the AArch32 instruction set state that the Debug state entry was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

**M[4], bit [4]**

Execution state that Debug state was entered from. Possible values of this bit are:

1 Exception taken from AArch32.

**M[3:0], bits [3:0]**

AArch32 mode that Debug state was entered from. The possible values are:

| M[3:0] | Mode |
|---|---|
| 0b0000 | User |
| 0b0001 | FIQ |
| 0b0010 | IRQ |
| 0b0011 | Supervisor |
| 0b0110 | Monitor (only valid in Secure state, if EL3 is implemented and can use AArch32) |
| 0b0111 | Abort |
| 0b1010 | Hyp |
| 0b1011 | Undefined |
| 0b1111 | System |

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.

## Accessing the DSPSR:

To access the DSPSR:

```
MRC p15,3,<Rt>,c4,c5,0 ; Read DSPSR into Rt
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MCR p15,3,<Rt>,c4,c5,0 ; Write Rt to DSPSR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 011 | 0100 | 0101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.31 HDCR, Hyp Debug Control Register

The HDCR characteristics are:

**Purpose**

Controls the trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to functions provided by the debug and trace architectures and the Performance Monitors extension.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TDA==1, PL2 accesses to this register are trapped to EL3 using AArch64.

**Configurations**

AArch32 System register HDCR is architecturally mapped to AArch64 System register MDCR_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

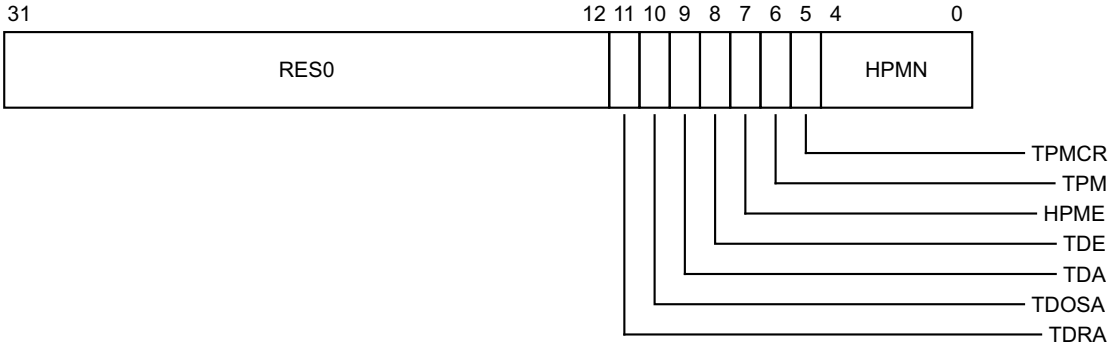
**Attributes**

HDCR is a 32-bit register.

#### Field descriptions

The HDCR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:12]**

Reserved, RES0.

**TDRA, bit [11]**

Trap Debug ROM Address register access. Traps Non-secure EL0 and EL1 CP14 accesses to the Debug ROM registers to Hyp mode.

`0` Non-secure EL0 and EL1 CP14 accesses to the Debug ROM registers are not trapped to Hyp mode.

`1` Non-secure EL0 and EL1 CP14 accesses to the DBGDRAR or DBGDSAR are trapped to Hyp mode.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

**TDOSA, bit [10]**

Trap debug OS-related register access. Traps Non-secure EL1 CP14 accesses to the powerdown debug registers to Hyp mode.

`0` Non-secure EL1 CP14 accesses to the powerdown debug registers are not trapped to Hyp mode.

`1` Non-secure EL1 CP14 accesses to the powerdown debug registers are trapped to Hyp mode.

The registers for which accesses are trapped are as follows:

- DBGOSLSR, DBGOSLAR, DBGOSDLR, and the DBGPRCR.
- Any IMPLEMENTATION DEFINED integration registers.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

**Note**

These registers are not accessible at PL0.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**TDA, bit [9]**

Trap debug access. Traps Non-secure EL0 and EL1 CP14 accesses to those CP14 registers that are not trapped by either of the following:

- HDCR.TDRA.
- HDCR.TDOSA.

`0` Has no effect on CP14 accesses to the debug registers.

`1` Non-secure EL0 or EL1 CP14 accesses to the debug registers, other than the registers trapped by HDCR.TDRA and HDCR.TDOSA, are trapped to Hyp mode.

HDCR.TDA does not trap accesses to the DBGDTRRXint or DBGDTRTXint when the PE is in Debug state.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

**TDE, bit [8]**

Trap Debug exceptions. The possible values of this bit are:

`0` Has no effect on Debug exceptions.

`1` Route Non-secure Debug exceptions to Hyp mode.

If HCR.TGE == 1, then this bit is ignored and treated as though it is 1 other than for the value read back from HDCR.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

**HPME, bit [7]**

Hypervisor Performance Monitors Enable. The possible values of this bit are:

`0` Hyp mode Performance Monitors disabled.

`1` Hyp mode Performance Monitors enabled.

When the value of this bit is 1, the Performance Monitors counters that are reserved for use from Hyp mode or Secure state are enabled. For more information see the description of the HPMN field.

If the Performance Monitors extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**TPM, bit [6]**

Trap Performance Monitors accesses. Traps Non-secure EL0 and EL1 accesses to all Performance Monitors registers to Hyp mode.

`0` Non-secure EL0 and EL1 accesses to all Performance Monitors registers are not trapped to Hyp mode.

`1` Non-secure EL0 and EL1 accesses to all Performance Monitors registers are trapped to Hyp mode.

**Note**

- EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

If the Performance Monitors extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to `0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to its defined reset value on Warm reset.

**TPMCR, bit [5]**

Trap PMCR accesses. Traps Non-secure EL0 and EL1 accesses to the PMCR to Hyp mode.

0 Non-secure EL0 and EL1 accesses to the PMCR are not trapped to Hyp mode.

1 Non-secure EL0 and EL1 accesses to the PMCR are trapped to Hyp mode.

**Note**

- EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

If the Performance Monitors extension is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

This field resets to its defined reset value on Warm reset.

**HPMN, bits [4:0]**

Defines the number of Performance Monitors counters that are accessible from Non-secure EL1 modes, and from Non-secure EL0 modes if unprivileged access is enabled.

If the Performance Monitors extension is not implemented, this field is RES0.

In Non-secure state, HPMN divides the Performance Monitors counters as follows. If software is accessing Performance Monitors counter n then, in Non-secure state:

- If n is in the range 0<=n<HPMN, the counter is accessible from EL1 and EL2, and from EL0 if unprivileged access to the counters is enabled. PMCR.E enables the operation of counters in this range.
- If n is in the range HPMN<=n<PMCR.N, the counter is accessible only from EL2. HDCR.HPME enables the operation of counters in this range.

If this field is set to 0, or to a value larger than PMCR.N, then the following CONSTRAINED UNPREDICTABLE behavior applies:

- The value returned by a direct read of HDCR.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if HDCR.HPMN is set to an UNKNOWN non-zero value less than PMCR.N.
  - All counters are reserved for EL2 use, meaning no counters are accessible from Non-secure EL1 and Non-secure EL0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to an architecturally UNKNOWN value.

This field resets to its defined reset value on Warm reset.

## Accessing the HDCR:

To access the HDCR:

```
MRC p15,4,<Rt>,c1,c1,1 ; Read HDCR into Rt
MCR p15,4,<Rt>,c1,c1,1 ; Write Rt to HDCR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 0001 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.32 SDCR, Secure Debug Configuration Register

The SDCR characteristics are:

**Purpose**

When EL3 is implemented and can use AArch32, controls debug and performance monitors functionality in Secure state.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|---|
| - | - | - | RW | - | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | - |

If EL3 is implemented and is using AArch64, any read or write to SDCR from Secure EL1 using AArch32 is trapped as an exception to EL3.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

This register is only accessible in Secure state.

AArch32 System register SDCR can be mapped to AArch64 System register MDCR_EL3, but this is not architecturally mandated.

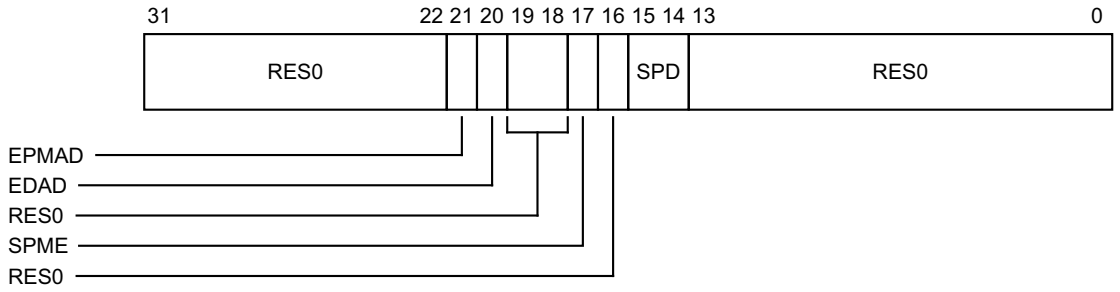
**Attributes**

SDCR is a 32-bit register.

#### Field descriptions

The SDCR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:22]**

Reserved, RES0.

**EPMAD, bit [21]**

External debug interface Performance Monitors registers disable. This disables access to these registers by an external debugger:

0 Access to Performance Monitors registers from external debugger is permitted.

1 Access to Performance Monitors registers from external debugger is disabled, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

If the Performance Monitors extension is not implemented or does not support external debug interface accesses this bit is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

This field resets to its defined reset value on Warm reset.

**EDAD, bit [20]**

External debug interface breakpoint and watchpoint register access disable. This disables access to these registers by an external debugger:

0 Access to breakpoint and watchpoint registers from external debugger is permitted.

1 Access to breakpoint and watchpoint registers from external debugger is disabled, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

**Bits [19:18]**

Reserved, RES0.

**SPME, bit [17]**

Secure Performance Monitors enable. This allows event counting in Secure state:

0 Event counting prohibited in Secure state, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

1 Event counting allowed in Secure state, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

**Bit [16]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SPD, bits [15:14]**

AArch32 Secure privileged debug. Enables or disables debug exceptions from Secure state, other than Software breakpoint instructions. Valid values for this field are:

`00` Legacy mode. Debug exceptions from Secure EL1 are enabled by the authentication interface.

`10` Secure privileged debug disabled. Debug exceptions from Secure EL1 are disabled.

`11` Secure privileged debug enabled. Debug exceptions from Secure EL1 are enabled.

Other values are reserved, and have the CONSTRAINED UNPREDICTABLE behavior that they must have the same behavior as `0b00`. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

If debug exceptions from Secure EL1 are enabled, then debug exceptions from Secure EL0 are also enabled.

Otherwise, debug exceptions from Secure EL0 are enabled only if SDER32_EL3.SUIDEN == 1.

Ignored in Non-secure state. Debug exceptions from Software breakpoint instruction debug events are always enabled.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

**Bits [13:0]**

Reserved, RES0.

## Accessing the SDCR:

To access the SDCR:

```
MRC p15,0,<Rt>,c1,c3,1 ; Read SDCR into Rt
MCR p15,0,<Rt>,c1,c3,1 ; Write Rt to SDCR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0001 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.3.33 SDER, Secure Debug Enable Register

The SDER characteristics are:

**Purpose**

Controls invasive and non-invasive debug in the Secure EL0 mode.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | - |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T1==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T1==1, Non-secure EL1 accesses to this register are trapped to EL2.

**Configurations**

This register is only accessible in Secure state.

AArch32 System register SDER is architecturally mapped to AArch64 System register SDER32_EL3.

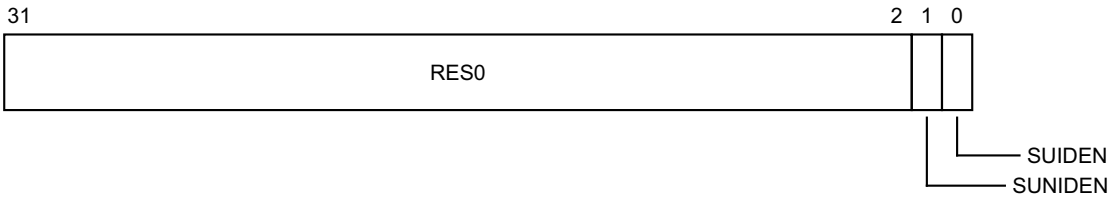
If EL3 is not implemented and EL1 supports AArch32, SDER is implemented only if the implemented Security state is Secure state.

**Attributes**

SDER is a 32-bit register.

#### Field descriptions

The SDER bit assignments are:




**Bits [31:2]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**SUNIDEN, bit [1]**

Secure User Non-Invasive Debug Enable:

0 Performance Monitors event counting prohibited in Secure EL0 unless allowed by MDCR_EL3.SPME, SDCR.SPME, or the IMPLEMENTATION DEFINED authentication interface ExternalSecureNoninvasiveDebugEnabled().

1 Performance Monitors event counting allowed in Secure EL0.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

**SUIDEN, bit [0]**

Secure User Invasive Debug Enable:

0 Debug exceptions other than Software Breakpoint Instruction exceptions from Secure EL0 are disabled, unless enabled by MDCR_EL3.SPD32 or SDCR.SPD.

1 Debug exceptions from Secure EL0 are enabled.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

## Accessing the SDER:

To access the SDER:

```
MRC p15,0,<Rt>,c1,c1,1 ; Read SDER into Rt
MCR p15,0,<Rt>,c1,c1,1 ; Write Rt to SDER
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 0001 | 0001 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.4 Performance Monitors registers

This section lists the Performance Monitors registers in AArch32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.1 PMCCFILTR, Performance Monitors Cycle Count Filter Register

The PMCCFILTR characteristics are:

**Purpose**

Determines the modes in which the Cycle Counter, PMCCNTR, increments.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

PMCCFILTR can also be accessed by using PMXEVTYPER with PMSELR.SEL set to `0b11111`.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, PL0 accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMCCFILTR is architecturally mapped to AArch64 System register PMCCFILTR_EL0.

AArch32 System register PMCCFILTR is architecturally mapped to External register PMCCFILTR_EL0.

Some or all RW fields of this register have defined reset values. These apply on Warm reset, and only if the PE resets into an Exception level that is using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

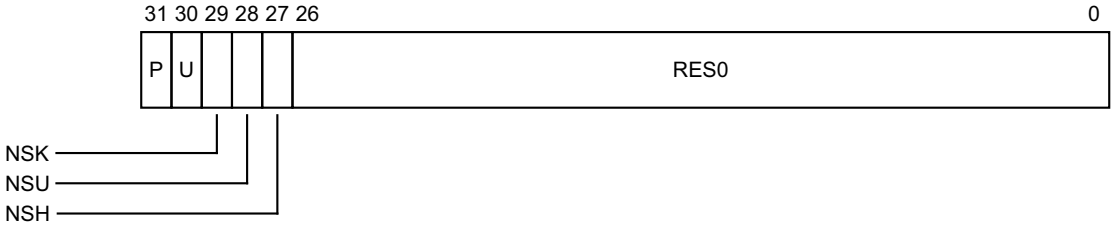
**Attributes**

PMCCFILTR is a 32-bit register.

#### Field descriptions

The PMCCFILTR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**P, bit [31]**

EL1 modes filtering bit. Controls counting in EL1. If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the NSK bit. The possible values of this bit are:

`0` Count cycles in EL1.

`1` Do not count cycles in EL1.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

**U, bit [30]**

EL0 filtering bit. Controls counting in EL0. If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the NSU bit. The possible values of this bit are:

`0` Count cycles in EL0.

`1` Do not count cycles in EL0.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

**NSK, bit [29]**

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, cycles in Non-secure EL1 are counted.

Otherwise, cycles in Non-secure EL1 are not counted.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to `0`.

This field resets to its defined reset value on Warm reset.

**NSU, bit [28]**

Non-secure User mode filtering bit. Controls counting in Non-secure EL0. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of U, cycles in Non-secure EL0 are counted.

Otherwise, cycles in Non-secure EL0 are not counted.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to `0`.

This field resets to its defined reset value on Warm reset.

**NSH, bit [27]**

Non-secure Hyp modes filtering bit. Controls counting in Non-secure EL2. If EL2 is not implemented, this bit is RES0.

`0` Do not count cycles in EL2.

`1` Count cycles in EL2.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to `0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to its defined reset value on Warm reset.

**Bits [26:0]**

Reserved, RES0.

### Accessing the PMCCFILTR:

To access the PMCCFILTR:

```
MRC p15,0,<Rt>,c14,c15,7 ; Read PMCCFILTR into Rt
MCR p15,0,<Rt>,c14,c15,7 ; Write Rt to PMCCFILTR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1110 | 1111 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.2 PMCCNTR, Performance Monitors Cycle Count Register

The PMCCNTR characteristics are:

**Purpose**

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See *Time as measured by the Performance Monitors cycle counter* on page D5-1858 for more information.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

The PMCR.{LC, D} bits configure whether PMCCNTR increments every clock cycle, or once every 64 clock cycles.

PMCCFILTR determines the modes and states in which the PMCCNTR can increment.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.CR==0, and PMUSERENR.EN==0, PL0 read accesses to this register are trapped to Undefined mode.

If PMUSERENR.EN==0, PL0 write accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.CR==0, and PMUSERENR_EL0.EN==0, PL0 read accesses to this register are trapped to EL1.

If PMUSERENR_EL0.EN==0, PL0 write accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMCCNTR is architecturally mapped to AArch64 System register PMCCNTR_EL0.

AArch32 System register PMCCNTR is architecturally mapped to External register PMCCNTR_EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions. This means that it is CONSTRAINED UNPREDICTABLE whether or not PMCCNTR continues to increment when clocks are stopped by WFI and WFE instructions.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

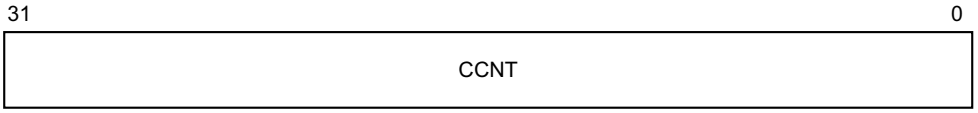
**Attributes**

PMCCNTR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

## Field descriptions

The PMCCNTR bit assignments are:

***When accessing as a 32-bit register:***



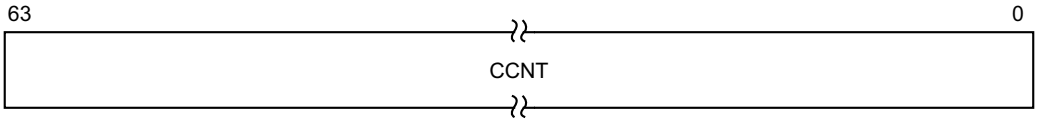

**CCNT, bits [31:0]**

Cycle count. Depending on the values of PMCR.{LC,D}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR.C sets this field to 0.

***When accessing as a 64-bit register:***




**CCNT, bits [63:0]**

Cycle count. Depending on the values of PMCR.{LC,D}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR.C sets this field to 0.

## Accessing the PMCCNTR:

To access the PMCCNTR when accessing as a 32-bit register:

```
MRC p15,0,<Rt>,c9,c13,0 ; Read PMCCNTR[31:0] into Rt
MCR p15,0,<Rt>,c9,c13,0 ; Write Rt to PMCCNTR[31:0]. PMCCNTR[63:32] are unchanged
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1101 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

To access the PMCCNTR when accessing as a 64-bit register:

```
MRRC p15,0,<Rt>,<Rt2>,c9 ; Read PMCCNTR[31:0] into Rt and PMCCNTR[63:32] into Rt2
MCRR p15,0,<Rt>,<Rt2>,c9 ; Write Rt to PMCCNTR[31:0] and Rt2 to PMCCNTR[63:32]
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0000 | 1001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.3 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

**Purpose**

Defines which common architectural and common microarchitectural feature events are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 read accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, PL0 read accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMCEID0 is architecturally mapped to AArch64 System register PMCEID0_EL0.

AArch32 System register PMCEID0 is architecturally mapped to External register PMCEID0_EL0.

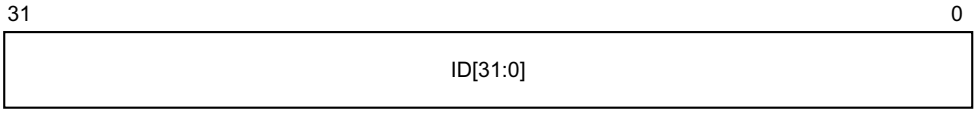
**Attributes**

PMCEID0 is a 32-bit register.

#### Field descriptions

The PMCEID0 bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**ID[31:0], bits [31:0]**

PMCEID0[n] maps to event n. For a list of event numbers and descriptions, see *Events, Event numbers and mnemonics* on page D5-1872.

For each bit:

`0` The common event is not implemented.

`1` The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

## Accessing the PMCEID0:

To access the PMCEID0:

```
MRC p15,0,<Rt>,c9,c12,6 ; Read PMCEID0 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1100 | 110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.4 PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

**Purpose**

Defines which common architectural and common microarchitectural feature events are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 read accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, PL0 read accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMCEID1 is architecturally mapped to AArch64 System register PMCEID1_EL0.

AArch32 System register PMCEID1 is architecturally mapped to External register PMCEID1_EL0.

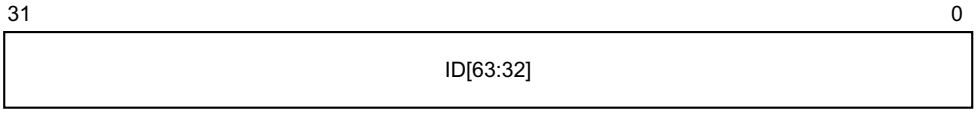
**Attributes**

PMCEID1 is a 32-bit register.

#### Field descriptions

The PMCEID1 bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**ID[63:32], bits [31:0]**

PMCEID1[n] maps to event (n + 32). For a list of event numbers and descriptions, see *Events, Event numbers and mnemonics* on page D5-1872.

For each bit:

`0` The common event is not implemented.

`1` The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

## Accessing the PMCEID1:

To access the PMCEID1:

```
MRC p15,0,<Rt>,c9,c12,7 ; Read PMCEID1 into Rt
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1100 | 111 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.5 PMCNTENCLR, Performance Monitors Count Enable Clear register

The PMCNTENCLR characteristics are:

**Purpose**

Disables the Cycle Count Register, PMCCNTR, and any implemented event counters PMEVCNTR<x>. Reading this register shows which counters are enabled.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

If EL2 is implemented, in Non-secure EL1 and EL0 modes, the value of HDCR.HPMN can change the behavior of accesses to PMCNTENCLR. See the description of the Px bit.

PMCNTENCLR is used in conjunction with the PMCNTENSET register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HSTR.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, PL0 accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMCNTENCLR is architecturally mapped to AArch64 System register PMCNTENCLR_EL0.

AArch32 System register PMCNTENCLR is architecturally mapped to External register PMCNTENCLR_EL0.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

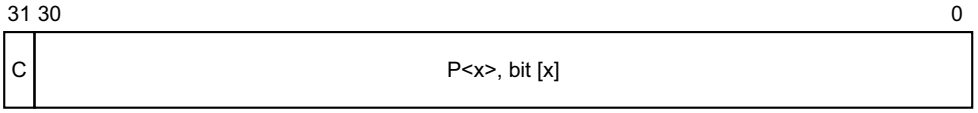
**Attributes**

PMCNTENCLR is a 32-bit register.

#### Field descriptions

The PMCNTENCLR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**C, bit [31]**

PMCCNTR disable bit. Disables the cycle counter register. Possible values are:

`0` When read, means the cycle counter is disabled. When written, has no effect.

`1` When read, means the cycle counter is enabled. When written, disables the cycle counter.

**P<x>, bit [x], for x = 0 to 30**

Event counter disable bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in HDCR.HPMN. Otherwise, N is the value in PMCR.N.

Possible values of each bit are:

`0` When read, means that PMEVCNTR<x> is disabled. When written, has no effect.

`1` When read, means that PMEVCNTR<x> is enabled. When written, disables PMEVCNTR<x>.

## Accessing the PMCNTENCLR:

To access the PMCNTENCLR:

```
MRC p15,0,<Rt>,c9,c12,2 ; Read PMCNTENCLR into Rt
MCR p15,0,<Rt>,c9,c12,2 ; Write Rt to PMCNTENCLR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1100 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.6 PMCNTENSET, Performance Monitors Count Enable Set register

The PMCNTENSET characteristics are:

**Purpose**

Enables the Cycle Count Register, PMCCNTR, and any implemented event counters PMEVCNTR<x>. Reading this register shows which counters are enabled.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

If EL2 is implemented, in Non-secure EL1 and EL0 modes, the value of HDCR.HPMN can change the behavior of accesses to PMCNTENSET. See the description of the Px bit.

PMCNTENSET is used in conjunction with the PMCNTENCLR register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, PL0 accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMCNTENSET is architecturally mapped to AArch64 System register PMCNTENSET_EL0.

AArch32 System register PMCNTENSET is architecturally mapped to External register PMCNTENSET_EL0.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

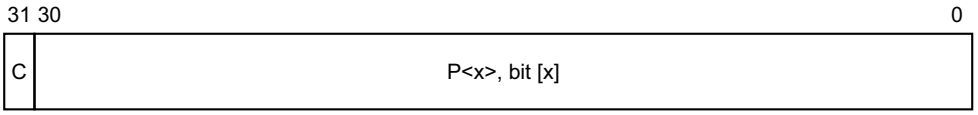
**Attributes**

PMCNTENSET is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## Field descriptions

The PMCNTENSET bit assignments are:




**C, bit [31]**

PMCCNTR enable bit. Enables the cycle counter register. Possible values are:

`0` When read, means the cycle counter is disabled. When written, has no effect.

`1` When read, means the cycle counter is enabled. When written, enables the cycle counter.

**P<x>, bit [x], for x = 0 to 30**

Event counter enable bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in HDCR.HPMN. Otherwise, N is the value in PMCR.N.

Possible values of each bit are:

`0` When read, means that PMEVCNTR<x> is disabled. When written, has no effect.

`1` When read, means that PMEVCNTR<x> event counter is enabled. When written, enables PMEVCNTR<x>.

## Accessing the PMCNTENSET:

To access the PMCNTENSET:

```
MRC p15,0,<Rt>,c9,c12,1 ; Read PMCNTENSET into Rt
MCR p15,0,<Rt>,c9,c12,1 ; Write Rt to PMCNTENSET
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1100 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.7 PMCR, Performance Monitors Control Register

The PMCR characteristics are:

**Purpose**

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HDCR.TPMCR==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPMCR==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, PL0 accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMCR is architecturally mapped to AArch64 System register PMCR_EL0.

AArch32 System register PMCR[6:0] is architecturally mapped to External register PMCR_EL0[6:0].

Some or all RW fields of this register have defined reset values. These apply on Warm reset, and only if the PE resets into an Exception level that is using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

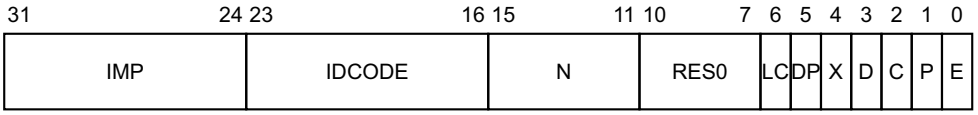
**Attributes**

PMCR is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The PMCR bit assignments are:




**IMP, bits [31:24]**

Implementer code. This field is RO with an IMPLEMENTATION DEFINED value.

The implementer codes are allocated by ARM. Values have the same interpretation as bits [31:24] of the MIDR.

**IDCODE, bits [23:16]**

Identification code. This field is RO with an IMPLEMENTATION DEFINED value.

Each implementer must maintain a list of identification codes that is specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

**N, bits [15:11]**

Number of event counters. A RO field that indicates the number counters implemented. The value of this field is the number of counters implemented, from `0b00000` for no counters to `0b11111` for 31 counters.

In an implementation that includes EL2, reads of this field from Non-secure EL1 and Non-secure EL0 return the value of HDCR.HPMN.

An implementation can implement only the Cycle Count Register, PMCCNTR. This is indicated by a value of `0b00000` for the N field.

**Bits [10:7]**

Reserved, RES0.

**LC, bit [6]**

Long cycle counter enable. Determines which PMCCNTR bit generates an overflow recorded by PMOVSR[31].

`0` Cycle counter overflow on increment that changes PMCCNTR[31] from 1 to 0.

`1` Cycle counter overflow on increment that changes PMCCNTR[63] from 1 to 0.

ARM deprecates use of PMCR.LC = 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**DP, bit [5]**

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

`0` PMCCNTR, if enabled, counts when event counting is prohibited.

`1` PMCCNTR does not count when event counting is prohibited.

Event counting is prohibited when ProfilingProhibited(IsSecure(),PSTATE.EL) == TRUE.

This bit is RW.

If EL3 is not implemented, this field is RES0.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to `0`.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**X, bit [4]**

Enable export of events in an IMPLEMENTATION DEFINED event stream. The possible values of this bit are:

0 Do not export events.

1 Export events where not prohibited.

This bit is used to permit events to be exported to another debug device, such as an OPTIONAL trace macrocell, over an event bus. If the implementation does not include such an event bus, this bit is RAZ/WI.

This bit does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If the implementation does not include an exported event stream, this bit is RAZ/WI. Otherwise this bit is RW.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to 0.

This field resets to its defined reset value on Warm reset.

**D, bit [3]**

Clock divider. The possible values of this bit are:

0 When enabled, PMCCNTR counts every clock cycle.

1 When enabled, PMCCNTR counts once every 64 clock cycles.

This bit is RW.

If PMCR.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

ARM deprecates use of PMCR.D = 1.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

**C, bit [2]**

Cycle counter reset. This bit is WO. The effects of writing to this bit are:

0 No action.

1 Reset PMCCNTR to zero.

This bit is always RAZ.

Resetting PMCCNTR does not clear the PMCCNTR overflow bit to 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

**P, bit [1]**

Event counter reset. This bit is WO. The effects of writing to this bit are:

0 No action.

1 Reset all event counters accessible in the current EL, not including PMCCNTR, to zero.

This bit is always RAZ.

In Non-secure EL0 and EL1, if EL2 is implemented, a write of 1 to this bit does not reset event counters that HDCR.HPMN reserves for EL2 use.

In EL2 and EL3, a write of 1 to this bit resets all the event counters.

Resetting the event counters does not clear any overflow bits to 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**E, bit [0]**

Enable. The possible values of this bit are:

`0` All counters, including PMCCNTR, are disabled.

`1` All counters are enabled by PMCNTENSET.

This bit is RW.

If EL2 is implemented, this bit does not affect the operation of event counters that HDCR.HPMN reserves for EL2 use.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

### Accessing the PMCR:

To access the PMCR:

```
MRC p15,0,<Rt>,c9,c12,0 ; Read PMCR into Rt
MCR p15,0,<Rt>,c9,c12,0 ; Write Rt to PMCR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1100 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.8 PMEVCNTR<n>, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n> characteristics are:

**Purpose**

Holds event counter n, which counts events, where n is 0 to 30.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

This register can be read at EL0 when PMUSERENR.EN or PMUSERENR.ER is set to 1, and can be written at EL0 when PMUSERENR.ER is set to 1.

PMEVCNTR<n> can also be accessed by using PMXEVCNTR with PMSELR.SEL set to n.

If <n> is greater than or equal to the number of accessible counters, reads and writes of PMEVCNTR<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- In Non-secure state, for an access from PL1 or a permitted access from PL0, if PMSELR.SEL, or PMSELR_EL0.SEL if EL1 is using AArch64, is greater than or equal to the number of accessible counters but is less than the number of implemented counters, the register access is trapped to EL2. Accesses from PL0 are permitted when:
  - EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  - EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

**Note**

In an implementation that includes EL2, in Non-secure state at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible counters.

Otherwise, the number of accessible counters is the number of implemented counters.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If PMUSERENR.EN==0, and PMUSERENR.ER==0, PL0 read accesses to this register are trapped to Undefined mode.

If PMUSERENR.EN==0, PL0 write accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 write accesses to this register are trapped to EL1.

If PMUSERENR_EL0.ER==0, and PMUSERENR_EL0.EN==0, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMEVCNTR<n> is architecturally mapped to AArch64 System register PMEVCNTR<n>_EL0.

AArch32 System register PMEVCNTR<n> is architecturally mapped to External register PMEVCNTR<n>_EL0.

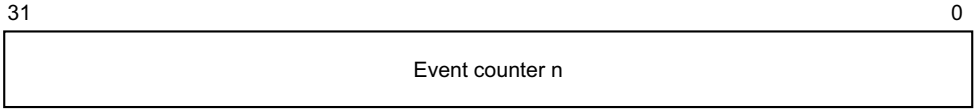
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMEVCNTR<n> is a 32-bit register.

## Field descriptions

The PMEVCNTR<n> bit assignments are:




**Bits [31:0]**

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

## Accessing the PMEVCNTR<n>:

To access the PMEVCNTR<n>:

```
MRC p15,0,<Rt>,c14,<CRm>,<opc2> ; Read PMEVCNTR<n> into Rt, where n is in the range 0 to 30
MCR p15,0,<Rt>,c14,<CRm>,<opc2> ; Write Rt to PMEVCNTR<n>, where n is in the range 0 to 30
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1110 | 10:n<4:3> | n<2:0> |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.9 PMEVTYPER<n>, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n> characteristics are:

**Purpose**

Configures event counter n, where n is 0 to 30.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

This register is accessible at EL0 when PMUSERENR.EN is set to 1.

PMEVTYPER<n> can also be accessed by using PMXEVTYPER with PMSELR.SEL set to n.

If <n> is greater or equal to the number of accessible counters, reads and writes of PMEVTYPER<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a `NOP`.
- In Non-secure state, for an access from PL1 or a permitted access from PL0, if PMSELR.SEL, or PMSELR_EL0.SEL if EL1 is using AArch64, is greater than or equal to the number of accessible counters but is less than the number of implemented counters, the register access is trapped to EL2. Accesses from PL0 are permitted when:
  - EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  - EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

**Note**

In an implementation that includes EL2, in Non-secure state at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible counters.

Otherwise, the number of accessible counters is the number of implemented counters.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, PL0 accesses to this register are trapped to Undefined mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If PMUSERENR_EL0.EN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMEVTYPER<n> is architecturally mapped to AArch64 System register PMEVTYPER<n>_EL0.

AArch32 System register PMEVTYPER<n> is architecturally mapped to External register PMEVTYPER<n>_EL0.

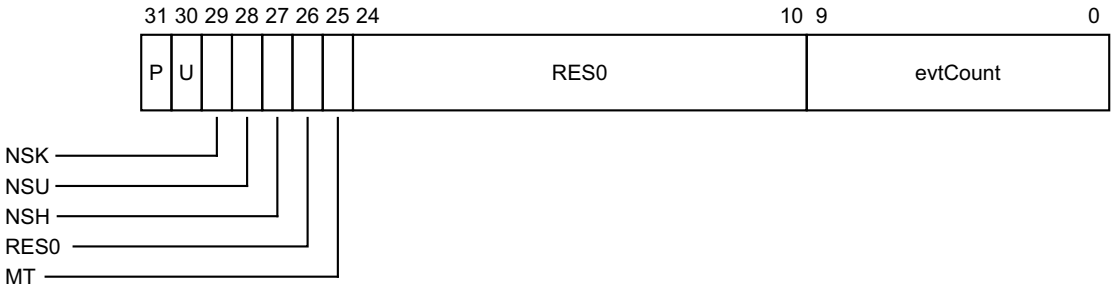
On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMEVTYPER<n> is a 32-bit register.

## Field descriptions

The PMEVTYPER<n> bit assignments are:




**P, bit [31]**

EL1 modes filtering bit. Controls counting in EL1. If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the NSK bit. The possible values of this bit are:

`0` Count events in EL1.

`1` Do not count events in EL1.

**U, bit [30]**

EL0 filtering bit. Controls counting in EL0. If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the NSU bit. The possible values of this bit are:

`0` Count events in EL0.

`1` Do not count events in EL0.

**NSK, bit [29]**

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.

**NSU, bit [28]**

Non-secure User mode filtering bit. Controls counting in Non-secure EL0. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of U, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**NSH, bit [27]**

Non-secure Hyp modes filtering bit. Controls counting in Non-secure EL2. If EL2 is not implemented, this bit is RES0.

0 Do not count events in EL2.

1 Count events in EL2.

**Bit [26]**

Reserved, RES0.

**MT, bit [25]**

Multi-threading. If MPIDR_EL1.MT is set to 0, this bit is RES0. Otherwise valid values for this bit are:

0 Count events only on controlling PE.

1 Count events from any PE with the same affinity at level 1 and above as this PE.

**Note**

Events from a different thread of a multi-threaded implementation are not Attributable to the thread counting the event.

**Bits [24:10]**

Reserved, RES0.

**evtCount, bits [9:0]**

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>.

Software must program this field with an event defined by the processor or a common event defined by the architecture.

If evtCount is programmed to an event that is reserved or not implemented, the behavior depends on the event type.

For common architectural and microarchitectural events:

- No events are counted.
- The value read back on evtCount is the value written.

For IMPLEMENTATION DEFINED events:

- It is UNPREDICTABLE what event, if any, is counted. UNPREDICTABLE in this case means the event must not expose privileged information.
- The value read back on evtCount is an UNKNOWN value with the same effect.

ARM recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.

### Accessing the PMEVTYPER<n>:

To access the PMEVTYPER<n>:

```
MRC p15,0,<Rt>,c14,<CRm>,<opc2> ; Read PMEVTYPER<n> into Rt, where n is in the range 0 to 30
MCR p15,0,<Rt>,c14,<CRm>,<opc2> ; Write Rt to PMEVTYPER<n>, where n is in the range 0 to 30
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1110 | 11:n<4:3> | n<2:0> |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.10 PMINTENCLR, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR characteristics are:

**Purpose**

Disables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR, and the event counters PMEVCNTR<n>. Reading the register shows which overflow interrupt requests are enabled.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

If EL2 is implemented, in Non-secure EL1 and EL0 modes, the value of HDCR.HPMN can change the behavior of accesses to PMINTENCLR. See the description of the P<x> bit.

PMINTENCLR is used in conjunction with the PMINTENSET register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMINTENCLR is architecturally mapped to AArch64 System register PMINTENCLR_EL1.

AArch32 System register PMINTENCLR is architecturally mapped to External register PMINTENCLR_EL1.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

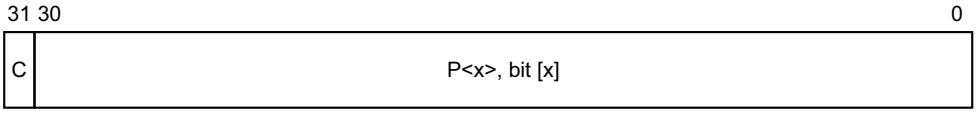
**Attributes**

PMINTENCLR is a 32-bit register.

#### Field descriptions

The PMINTENCLR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**C, bit [31]**

PMCCNTR overflow interrupt request disable bit. Possible values are:

`0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`1` When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow interrupt request disable bit for PMEVCNTR<x>.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in HDCR.HPMN. Otherwise, N is the value in PMCR.N.

Bits [30:N] are RAZ/WI.

Possible values are:

`0` When read, means that the PMEVCNTR<x> event counter interrupt request is disabled. When written, has no effect.

`1` When read, means that the PMEVCNTR<x> event counter interrupt request is enabled. When written, disables the PMEVCNTR<x> interrupt request.

## Accessing the PMINTENCLR:

To access the PMINTENCLR:

```
MRC p15,0,<Rt>,c9,c14,2 ; Read PMINTENCLR into Rt
MCR p15,0,<Rt>,c9,c14,2 ; Write Rt to PMINTENCLR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1110 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.11 PMINTENSET, Performance Monitors Interrupt Enable Set register

The PMINTENSET characteristics are:

**Purpose**

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR, and the event counters PMEVCNTR<n>. Reading the register shows which overflow interrupt requests are enabled.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

If EL2 is implemented, in Non-secure EL1 and EL0 modes, the value of HDCR.HPMN can change the behavior of accesses to PMINTENSET. See the description of the P<x> bit.

PMINTENSET is used in conjunction with the PMINTENCLR register.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL1 and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMINTENSET is architecturally mapped to AArch64 System register PMINTENSET_EL1.

AArch32 System register PMINTENSET is architecturally mapped to External register PMINTENSET_EL1.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

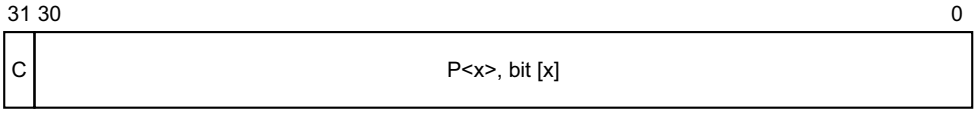
**Attributes**

PMINTENSET is a 32-bit register.

#### Field descriptions

The PMINTENSET bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**C, bit [31]**

PMCCNTR overflow interrupt request enable bit. Possible values are:

`0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`1` When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow interrupt request enable bit for PMEVCNTR<x>.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in HDCR.HPMN. Otherwise, N is the value in PMCR.N.

Bits [30:N] are RAZ/WI.

Possible values are:

`0` When read, means that the PMEVCNTR<x> event counter interrupt request is disabled. When written, has no effect.

`1` When read, means that the PMEVCNTR<x> event counter interrupt request is enabled. When written, enables the PMEVCNTR<x> interrupt request.

## Accessing the PMINTENSET:

To access the PMINTENSET:

```
MRC p15,0,<Rt>,c9,c14,1 ; Read PMINTENSET into Rt
MCR p15,0,<Rt>,c9,c14,1 ; Write Rt to PMINTENSET
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1110 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.12 PMOVSR, Performance Monitors Overflow Flag Status Register

The PMOVSR characteristics are:

**Purpose**

Contains the state of the overflow bit for the Cycle Count Register, PMCCNTR, and each of the implemented event counters PMEVCNTR<x>. Writing to this register clears these bits.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

If EL2 is implemented, in Non-secure EL1 and EL0 modes, the value of HDCR.HPMN can change the behavior of accesses to PMOVSR. See the description of the Px bit.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, PL0 accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMOVSR is architecturally mapped to AArch64 System register PMOVSCLR_EL0.

AArch32 System register PMOVSR is architecturally mapped to External register PMOVSCLR_EL0.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

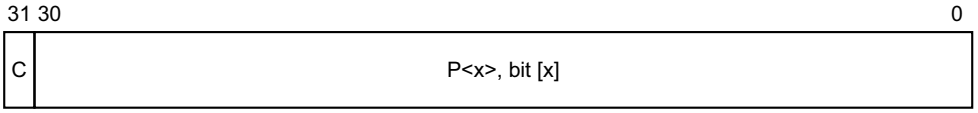
**Attributes**

PMOVSR is a 32-bit register.

#### Field descriptions

The PMOVSR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**C, bit [31]**

PMCCNTR overflow bit. Possible values are:

0 When read, means the cycle counter has not overflowed. When written, has no effect.

1 When read, means the cycle counter has overflowed. When written, clears the overflow bit to 0.

PMCR.LC is used to control from which bit of PMCCNTR (bit 31 or bit 63) an overflow is detected.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow clear bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in HDCR.HPMN. Otherwise, N is the value in PMCR.N.

Possible values of each bit are:

0 When read, means that PMEVCNTR<x> has not overflowed. When written, has no effect.

1 When read, means that PMEVCNTR<x> has overflowed. When written, clears the PMEVCNTR<x> overflow bit to 0.

## Accessing the PMOVSR:

To access the PMOVSR:

```
MRC p15,0,<Rt>,c9,c12,3 ; Read PMOVSR into Rt
MCR p15,0,<Rt>,c9,c12,3 ; Write Rt to PMOVSR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1100 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.13 PMOVSSET, Performance Monitors Overflow Flag Status Set register

The PMOVSSET characteristics are:

**Purpose**

Sets the state of the overflow bit for the Cycle Count Register, PMCCNTR, and each of the implemented event counters PMEVCNTR<x>.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

If EL2 is implemented, in Non-secure EL1 and EL0 modes, the value of HDCR.HPMN can change the behavior of accesses to PMOVSSET. See the description of the Px bit.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, PL0 accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMOVSSET is architecturally mapped to AArch64 System register PMOVSSET_EL0.

AArch32 System register PMOVSSET is architecturally mapped to External register PMOVSSET_EL0.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

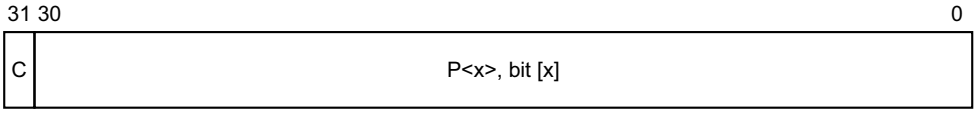
**Attributes**

PMOVSSET is a 32-bit register.

#### Field descriptions

The PMOVSSET bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**C, bit [31]**

PMCCNTR overflow bit. Possible values are:

`0` When read, means the cycle counter has not overflowed. When written, has no effect.

`1` When read, means the cycle counter has overflowed. When written, sets the overflow bit to 1.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow set bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in HDCR.HPMN. Otherwise, N is the value in PMCR.N.

Possible values are:

`0` When read, means that PMEVCNTR<x> has not overflowed. When written, has no effect.

`1` When read, means that PMEVCNTR<x> has overflowed. When written, sets the PMEVCNTR<x> overflow bit to 1.

## Accessing the PMOVSSET:

To access the PMOVSSET:

```
MRC p15,0,<Rt>,c9,c14,3 ; Read PMOVSSET into Rt
MCR p15,0,<Rt>,c9,c14,3 ; Write Rt to PMOVSSET
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1110 | 011 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.14 PMSELR, Performance Monitors Event Counter Selection Register

The PMSELR characteristics are:

**Purpose**

Selects the current event counter PMEVCNTR<x> or the cycle counter, CCNT.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

PMSELR is used in conjunction with PMXEVTYPER to determine the event that increments a selected event counter, and the modes and states in which the selected counter increments.

It is also used in conjunction with PMXEVCNTR, to determine the value of a selected event counter.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, and PMUSERENR.ER==0, PL0 accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, and PMUSERENR_EL0.ER==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMSELR is architecturally mapped to AArch64 System register PMSELR_EL0.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

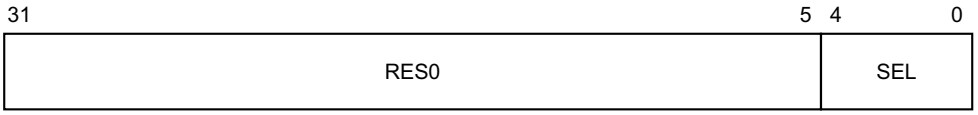
**Attributes**

PMSELR is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The PMSELR bit assignments are:




**Bits [31:5]**

Reserved, RES0.

**SEL, bits [4:0]**

Selects event counter, PMEVCNTR<x>, where x is the value held in this field. This value identifies which event counter is accessed when a subsequent access to PMXEVTYPER or PMXEVCNTR occurs.

This field can take any value from 0 (`0b00000`) to (PMCR.N)-1, or 31 (`0b11111`).

When PMSELR.SEL is `0b11111` it selects the cycle counter and:

- A read of the PMXEVTYPER returns the value of PMCCFILTR.
- A write of the PMXEVTYPER writes to PMCCFILTR.
- A read or write of PMXEVCNTR has CONSTRAINED UNPREDICTABLE effects, that can be one of the following:
  - Access to PMXEVCNTR is UNDEFINED.
  - Access to PMXEVCNTR behaves as a NOP.
  - Access to PMXEVCNTR behaves as if the register is RAZ/WI.
  - Access to PMXEVCNTR behaves as if the PMSELR.SEL field contains an UNKNOWN value.

If this field is set to a value greater than or equal to the number of implemented counters, but not equal to 31, the results of access to PMXEVTYPER or PMXEVCNTR are CONSTRAINED UNPREDICTABLE, and can be one of the following:

- Access to PMXEVTYPER or PMXEVCNTR is UNDEFINED.
- Access to PMXEVTYPER or PMXEVCNTR behaves as a NOP.
- Access to PMXEVTYPER or PMXEVCNTR behaves as if the register is RAZ/WI.
- Access to PMXEVTYPER or PMXEVCNTR behaves as if the PMSELR.SEL field contains an UNKNOWN value.
- Access to PMXEVTYPER or PMXEVCNTR behaves as if the PMSELR.SEL field contains `0b11111`.

### Accessing the PMSELR:

To access the PMSELR:

```
MRC p15,0,<Rt>,c9,c12,5 ; Read PMSELR into Rt
MCR p15,0,<Rt>,c9,c12,5 ; Write Rt to PMSELR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1100 | 101 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.15 PMSWINC, Performance Monitors Software Increment register

The PMSWINC characteristics are:

**Purpose**

Increments a counter that is configured to count the Software increment event, event `0x00`.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-WO | Config-WO | WO | WO | WO | WO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-WO | WO | WO |

If EL2 is implemented, in Non-secure EL1 and EL0 modes, the value of HDCR.HPMN can change the behavior of accesses to PMSWINC. See the description of the Px bit.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 write accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 write accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 write accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 write accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 write accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, and PMUSERENR.SW==0, PL0 write accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, and PMUSERENR_EL0.SW==0, PL0 write accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMSWINC is architecturally mapped to AArch64 System register PMSWINC_EL0.

AArch32 System register PMSWINC is architecturally mapped to External register PMSWINC_EL0.

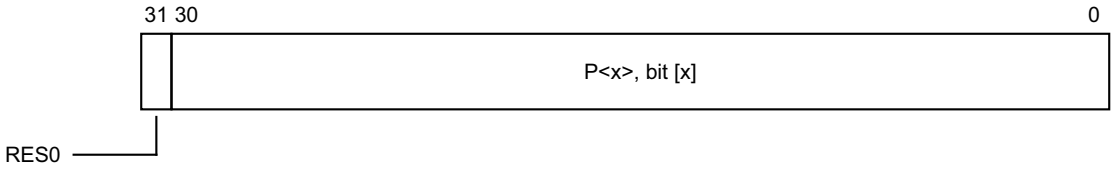
**Attributes**

PMSWINC is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The PMSWINC bit assignments are:




**Bit [31]**

Reserved, RES0.

**P<x>, bit [x], for x = 0 to 30**

Event counter software increment bit for PMEVCNTR<x>.

Bits [30:N] are WI.

When EL2 is implemented, in Non-secure EL1 and EL0, N is the value in HDCR.HPMN. Otherwise, N is the value in PMCR.N.

The effects of writing to this bit are:

0 No action. The write to this bit is ignored.

1 If PMEVCNTR<x> is enabled and configured to count the software increment event, increments PMEVCNTR<x> by 1. If PMEVCNTR<x> is disabled, or not configured to count the software increment event, the write to this bit is ignored.

### Accessing the PMSWINC:

To access the PMSWINC:

```
MCR p15,0,<Rt>,c9,c12,4 ; Write Rt to PMSWINC
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1100 | 100 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.16 PMUSERENR, Performance Monitors User Enable Register

The PMUSERENR characteristics are:

**Purpose**

Enables or disables User mode access to the Performance Monitors.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| RO | RO | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| RO | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMUSERENR is architecturally mapped to AArch64 System register PMUSERENR_EL0.

Some or all RW fields of this register have defined reset values. These apply on Warm reset, and only if the PE resets into an Exception level that is using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

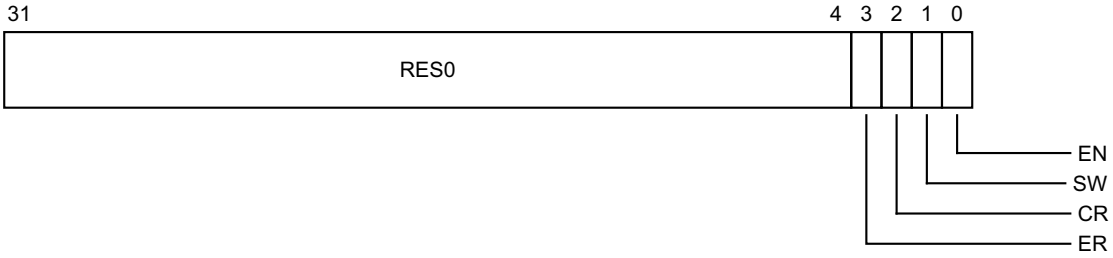
**Attributes**

PMUSERENR is a 32-bit register.

#### Field descriptions

The PMUSERENR bit assignments are:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




**Bits [31:4]**

Reserved, RES0.

**ER, bit [3]**

Event counter read trap control:

0 PL0 reads of the PMXEVCNTR and PMEVCNTR<n>, and PL0 read/write access to the PMSELR, are trapped to Undefined mode if PMUSERENR.EN is also 0.

1 PL0 reads of the PMXEVCNTR and PMEVCNTR<n>, and PL0 read/write access to the PMSELR, are not trapped to Undefined mode.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

**CR, bit [2]**

Cycle counter read trap control:

0 PL0 reads of the PMCCNTR are trapped to Undefined mode if PMUSERENR.EN is also 0.

1 PL0 reads of the PMCCNTR are not trapped to Undefined mode.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

**SW, bit [1]**

Software increment write trap control:

0 PL0 writes to the PMSWINC are trapped to Undefined mode if PMUSERENR.EN is also 0.

1 PL0 writes to the PMSWINC are not trapped to Undefined mode.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.

**EN, bit [0]**

Traps PL0 accesses to the Performance Monitors registers to Undefined mode:

0 PL0 accesses to the Performance Monitors registers are trapped to Undefined mode.

1 PL0 accesses to the Performance Monitors registers are not trapped to Undefined mode. Software can access all PMU registers at PL0.

**Note**

- The PMUSERENR is register is always RO at EL0 and not trapped by this bit.
- PL0 cannot read or write PMINTENSET and PMINTENCLR.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMUSERENR:

To access the PMUSERENR:

```
MRC p15,0,<Rt>,c9,c14,0 ; Read PMUSERENR into Rt
MCR p15,0,<Rt>,c9,c14,0 ; Write Rt to PMUSERENR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1110 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.17 PMXEVCNTR, Performance Monitors Selected Event Count Register

The PMXEVCNTR characteristics are:

**Purpose**

Reads or writes the value of the selected event counter, PMEVCNTR<x>. PMSELR.SEL determines which event counter is selected.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

If PMSELR.SEL is greater than or equal to the number of accessible counters then reads and writes of PMXEVCNTR are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if PMSELR.SEL has an UNKNOWN value less than the number of counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR.SEL is 31.
- In Non-secure state, for an access from PL1 or a permitted access from PL0, if PMSELR.SEL, or PMSELR_EL0.SEL if EL1 is using AArch64, is greater than or equal to the number of accessible counters but is less than the number of implemented counters, the register access is trapped to EL2. Accesses from PL0 are permitted when:
  - EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  - EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

**Note**

In an implementation that includes EL2, in Non-secure state at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible counters.

Otherwise, the number of accessible counters is the number of implemented counters.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, and PMUSERENR.ER==0, PL0 read accesses to this register are trapped to Undefined mode.

If PMUSERENR.EN==0, PL0 write accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 write accesses to this register are trapped to EL1.

If PMUSERENR_EL0.ER==0, and PMUSERENR_EL0.EN==0, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMXEVCNTR is architecturally mapped to AArch64 System register PMXEVCNTR_EL0.

On Warm reset, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

PMXEVCNTR is a 32-bit register.

## Field descriptions

The PMXEVCNTR bit assignments are:




**PMEVCNTR<x>, bits [31:0]**

Value of the selected event counter, PMEVCNTR<x>, where x is the value stored in PMSELR.SEL.

## Accessing the PMXEVCNTR:

To access the PMXEVCNTR:

```
MRC p15,0,<Rt>,c9,c13,2 ; Read PMXEVCNTR into Rt
MCR p15,0,<Rt>,c9,c13,2 ; Write Rt to PMXEVCNTR
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1101 | 010 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.4.18 PMXEVTYPER, Performance Monitors Selected Event Type Register

The PMXEVTYPER characteristics are:

**Purpose**

When PMSELR.SEL selects an event counter, this accesses a PMEVTYPER<n> register. When PMSELR.SEL selects the cycle counter, this accesses PMCCFILTR.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

If PMSELR.SEL is greater than or equal to the number of accessible counters then reads and writes of PMXEVTYPER are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if PMSELR.SEL has an UNKNOWN value less than the number of counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR.SEL is 31.
- In Non-secure state, for an access from PL1 or a permitted access from PL0, if PMSELR.SEL, or PMSELR_EL0.SEL if EL1 is using AArch64, is greater than or equal to the number of accessible counters but is less than the number of implemented counters, the register access is trapped to EL2. Accesses from PL0 are permitted when:
  - EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  - EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

**Note**

In an implementation that includes EL2, in Non-secure state at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible counters.

Otherwise, the number of accessible counters is the number of implemented counters.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If HDCR.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If HSTR.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If HSTR_EL2.T9==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL2.TPM==1, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If MDCR_EL3.TPM==1, PL0, PL1, and PL2 accesses to this register are trapped to EL3.

If PMUSERENR.EN==0, PL0 accesses to this register are trapped to Undefined mode.

If PMUSERENR_EL0.EN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register PMXEVTYPER is architecturally mapped to AArch64 System register PMXEVTYPER_EL0.

When the value of PMSELR.SEL is 31, to select the cycle counter, RW fields in this register have defined reset values that apply only when the PE resets into an Exception level that is using AArch32. See PMCCFILTR for the reset values.

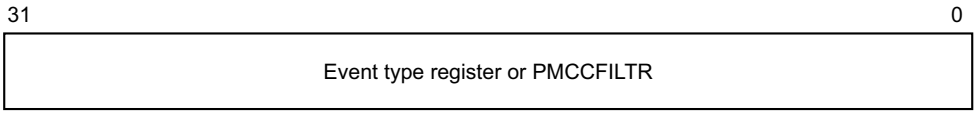
Otherwise, RW fields in this register reset to architecturally UNKNOWN values. This applies whenever PMSELR.SEL selects an event counter.

**Attributes**

PMXEVTYPER is a 32-bit register.

## Field descriptions

The PMXEVTYPER bit assignments are:




**Bits [31:0]**

Event type register or PMCCFILTR.

When PMSELR.SEL == 31, this register accesses PMCCFILTR.

Otherwise, this register accesses PMEVTYPER<n> where n is the value in PMSELR.SEL.

## Accessing the PMXEVTYPER:

To access the PMXEVTYPER:

```
MRC p15,0,<Rt>,c9,c13,1 ; Read PMXEVTYPER into Rt
MCR p15,0,<Rt>,c9,c13,1 ; Write Rt to PMXEVTYPER
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1001 | 1101 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## G6.5 Generic Timer registers

This section lists the Generic Timer registers in AArch32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.1 CNTFRQ, Counter-timer Frequency register

The CNTFRQ characteristics are:

**Purpose**

Holds the clock frequency of the system counter.

**Usage constraints**

If EL1 is the highest exception level implemented and is using AArch32, this register is accessible as follows:

| EL0 | EL1 |
|---|---|
| Config-RO | RW |

If EL2 is the highest exception level implemented and is using AArch32, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RW |

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | RO | RO | RW | RW |

If EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RO |

Can only be written at the highest Exception level implemented. For example, if EL3 is the highest implemented Exception level, CNTFRQ can only be written at EL3.

If EL3 is using AArch64, write accesses to CNTFRQ from Secure EL1modes are UNDEFINED.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CNTKCTL.PL0PCTEN==0, and CNTKCTL.PL0VCTEN==0, PL0 read accesses to this register are trapped to Undefined mode.

If CNTKCTL_EL1.EL0PCTEN==0, and CNTKCTL_EL1.EL0VCTEN==0, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register CNTFRQ is architecturally mapped to AArch64 System register CNTFRQ_EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

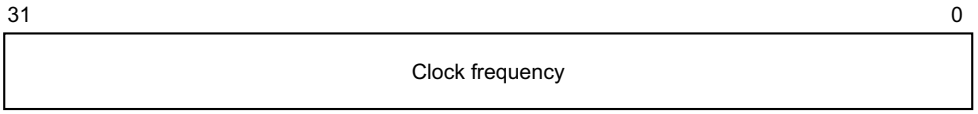
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTFRQ is a 32-bit register.

## Field descriptions

The CNTFRQ bit assignments are:




**Bits [31:0]**

Clock frequency. Indicates the system counter clock frequency, in Hz.

## Accessing the CNTFRQ:

To access the CNTFRQ:

```
MRC p15,0,<Rt>,c14,c0,0 ; Read CNTFRQ into Rt
MCR p15,0,<Rt>,c14,c0,0 ; Write Rt to CNTFRQ
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1110 | 0000 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.2 CNTHCTL, Counter-timer Hyp Control register

The CNTHCTL characteristics are:

**Purpose**

Controls the generation of an event stream from the physical counter, and access from Non-secure EL1 modes to the physical counter and the Non-secure EL1 physical timer.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch32 System register CNTHCTL is architecturally mapped to AArch64 System register CNTHCTL_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

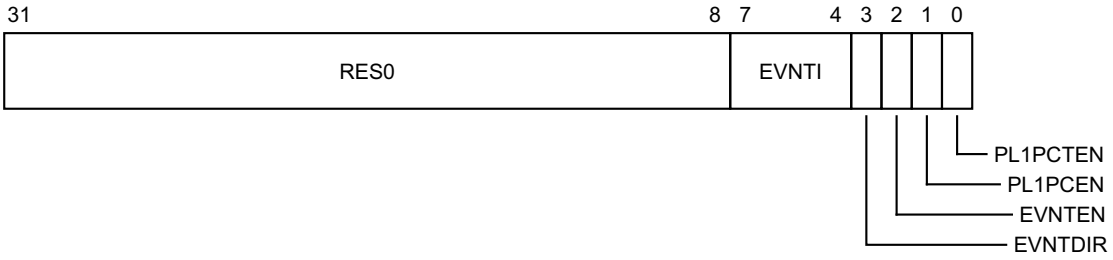
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 with EL2 using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTHCTL is a 32-bit register.

#### Field descriptions

The CNTHCTL bit assignments are:




**Bits [31:8]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EVNTI, bits [7:4]**

Selects which bit (0 to 15) of the corresponding counter register (CNTPCT or CNTVCT) is the trigger for the event stream generated from that counter, when that stream is enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**EVNTDIR, bit [3]**

Controls which transition of the counter register (CNTPCT or CNTVCT) trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

`0` A 0 to 1 transition of the trigger bit triggers an event.

`1` A 1 to 0 transition of the trigger bit triggers an event.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**EVNTEN, bit [2]**

Enables the generation of an event stream from the corresponding counter:

`0` Disables the event stream.

`1` Enables the event stream.

When this register has an architecturally-defined reset value, this field resets to `0`.

**PL1PCEN, bit [1]**

Traps Non-secure EL0 and EL1 accesses to the physical timer registers to Hyp mode.

`0` Non-secure EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are trapped to Hyp mode.

`1` Non-secure EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are not trapped to Hyp mode.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to `1`.

**PL1PCTEN, bit [0]**

Traps Non-secure EL0 and EL1 accesses to the physical counter register to Hyp mode.

`0` Non-secure EL0 and EL1 accesses to the CNTPCT are trapped to Hyp mode.

`1` Non-secure EL0 and EL1 accesses to the CNTPCT are not trapped to Hyp mode.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

When this register has an architecturally-defined reset value, this field resets to `1`.

## Accessing the CNTHCTL:

To access the CNTHCTL:

```
MRC p15,4,<Rt>,c14,c1,0 ; Read CNTHCTL into Rt
MCR p15,4,<Rt>,c14,c1,0 ; Write Rt to CNTHCTL
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1110 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.3 CNTHP_CTL, Counter-timer Hyp Physical Timer Control register

The CNTHP_CTL characteristics are:

**Purpose**

Control register for the Hyp mode physical timer.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch32 System register CNTHP_CTL is architecturally mapped to AArch64 System register CNTHP_CTL_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

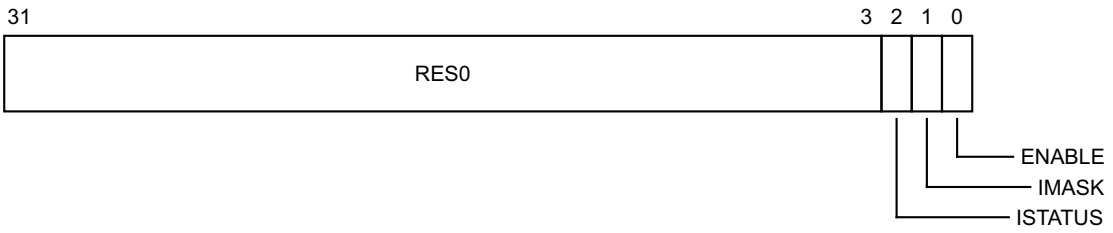
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into EL2 with EL2 using AArch32, or into EL3 with EL3 using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTHP_CTL is a 32-bit register.

#### Field descriptions

The CNTHP_CTL bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is asserted:

0 Timer condition is not asserted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see *Operation of the CompareValue views of the timers* on page D6-1904 and *Operation of the TimerValue views of the timers* on page D6-1904.

This bit is read-only.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**IMASK, bit [1]**

Timer output signal mask bit. Permitted values are:

0 Timer output signal is not masked.

1 Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

0 Timer disabled.

1 Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHP_TVAL continues to count down.

**Note**

Disabling the output signal might be a power-saving option.

When this register has an architecturally-defined reset value, this field resets to 0.

## Accessing the CNTHP_CTL:

To access the CNTHP_CTL:

```
MRC p15,4,<Rt>,c14,c2,1 ; Read CNTHP_CTL into Rt
MCR p15,4,<Rt>,c14,c2,1 ; Write Rt to CNTHP_CTL
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1110 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.4 CNTHP_CVAL, Counter-timer Hyp Physical CompareValue register

The CNTHP_CVAL characteristics are:

**Purpose**

Holds the compare value for the Hyp mode physical timer.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch32 System register CNTHP_CVAL is architecturally mapped to AArch64 System register CNTHP_CVAL_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

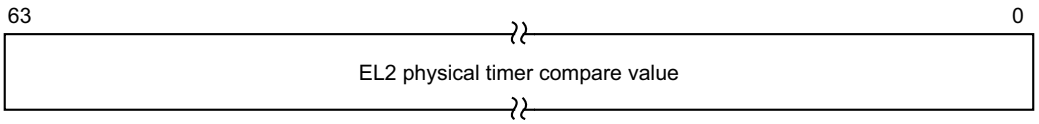
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTHP_CVAL is a 64-bit register.

#### Field descriptions

The CNTHP_CVAL bit assignments are:




**Bits [63:0]**

EL2 physical timer compare value.

#### Accessing the CNTHP_CVAL:

To access the CNTHP_CVAL:

```
MRRC p15,6,<Rt>,<Rt2>,c14 ; Read CNTHP_CVAL[31:0] into Rt and CNTHP_CVAL[63:32] into Rt2
MCRR p15,6,<Rt>,<Rt2>,c14 ; Write Rt to CNTHP_CVAL[31:0] and Rt2 to CNTHP_CVAL[63:32]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0110 | 1110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.5 CNTHP_TVAL, Counter-timer Hyp Physical Timer TimerValue register

The CNTHP_TVAL characteristics are:

**Purpose**

Holds the timer value for the Hyp mode physical timer.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

Where the value of CNTHP_CTL.ENABLE is 0:

- A write to CNTHP_TVAL updates the register
- The value held in CNTHP_TVAL continues to decrement
- A read of CNTHP_TVAL returns an UNKNOWN value.

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch32 System register CNTHP_TVAL is architecturally mapped to AArch64 System register CNTHP_TVAL_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

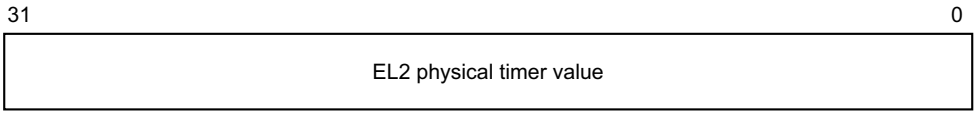
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTHP_TVAL is a 32-bit register.

#### Field descriptions

The CNTHP_TVAL bit assignments are:




**Bits [31:0]**

EL2 physical timer value.

#### Accessing the CNTHP_TVAL:

To access the CNTHP_TVAL:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRC p15,4,<Rt>,c14,c2,0 ; Read CNTHP_TVAL into Rt
MCR p15,4,<Rt>,c14,c2,0 ; Write Rt to CNTHP_TVAL
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 100 | 1110 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.6 CNTKCTL, Counter-timer Kernel Control register

The CNTKCTL characteristics are:

**Purpose**

Controls the generation of an event stream from the virtual counter, and access from EL0 modes to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register CNTKCTL is architecturally mapped to AArch64 System register CNTKCTL_EL1.

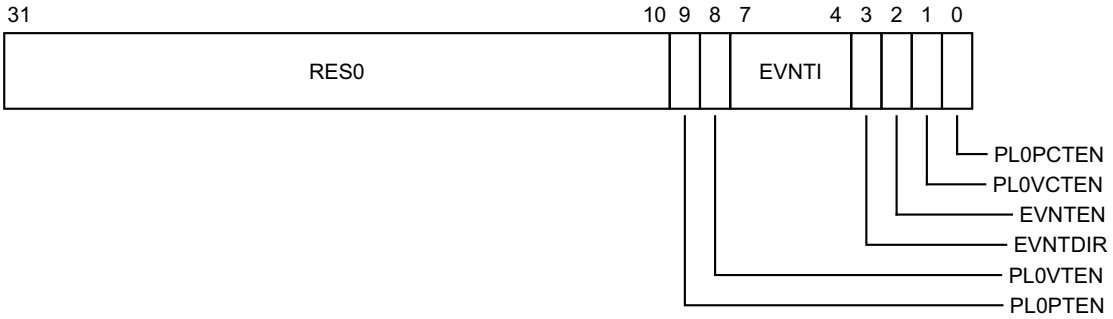
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTKCTL is a 32-bit register.

#### Field descriptions

The CNTKCTL bit assignments are:




**Bits [31:10]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**PL0PTEN, bit [9]**

Traps PL0 accesses to the physical timer registers to Undefined mode.

0 PL0 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL registers are trapped to Undefined mode.

1 PL0 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL registers are not trapped to Undefined mode.

When this register has an architecturally-defined reset value, this field resets to 0.

**PL0VTEN, bit [8]**

Traps PL0 accesses to the virtual timer registers to Undefined mode.

0 PL0 accesses to the CNTV_CTL, CNTV_CVAL, and CNTV_TVAL registers are trapped to Undefined mode.

1 PL0 accesses to the CNTV_CTL, CNTV_CVAL, and CNTV_TVAL registers are not trapped to Undefined mode.

When this register has an architecturally-defined reset value, this field resets to 0.

**EVNTI, bits [7:4]**

Selects which bit (0 to 15) of the corresponding counter register (CNTPCT or CNTVCT) is the trigger for the event stream generated from that counter, when that stream is enabled.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**EVNTDIR, bit [3]**

Controls which transition of the counter register (CNTPCT or CNTVCT) trigger bit, defined by EVNTI, generates an event when the event stream is enabled:

0 A 0 to 1 transition of the trigger bit triggers an event.

1 A 1 to 0 transition of the trigger bit triggers an event.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**EVNTEN, bit [2]**

Enables the generation of an event stream from the corresponding counter:

0 Disables the event stream.

1 Enables the event stream.

When this register has an architecturally-defined reset value, this field resets to 0.

**PL0VCTEN, bit [1]**

Traps PL0 accesses to the frequency register and virtual counter register to Undefined mode.

0 PL0 accesses to the CNTVCT are trapped to Undefined mode.
PL0 accesses to the CNTFRQ register are trapped to Undefined mode, if CNTKCTL.PL0PCTEN is also 0.

1 PL0 accesses to the CNTFRQ and CNTVCT are not trapped to Undefined mode.

When this register has an architecturally-defined reset value, this field resets to 0.

**PL0PCTEN, bit [0]**

Traps PL0 accesses to the frequency register and physical counter register to Undefined mode.

0 PL0 accesses to the CNTPCT are trapped to Undefined mode.
PL0 accesses to the CNTFRQ register are trapped to Undefined mode, if CNTKCTL.PL0VCTEN is also 0.

1 PL0 accesses to the CNTFRQ and CNTPCT are not trapped to Undefined mode.

When this register has an architecturally-defined reset value, this field resets to 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTKCTL:

To access the CNTKCTL:

```
MRC p15,0,<Rt>,c14,c1,0 ; Read CNTKCTL into Rt
MCR p15,0,<Rt>,c14,c1,0 ; Write Rt to CNTKCTL
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1110 | 0001 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.7 CNTP_CTL, Counter-timer Physical Timer Control register

The CNTP_CTL characteristics are:

**Purpose**

Control register for the EL1 physical timer.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

CNTP_CTL(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | Config-RW | - | - | - | RW |

CNTP_CTL(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | - | Config-RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

CNTP_CTL is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | Config-RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CNTHCTL.PL1PCEN==0, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If CNTHCTL_EL2.EL1PCEN==0, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If CNTKCTL.PL0PTEN==0, PL0 accesses to this register are trapped to Undefined mode.

If CNTKCTL_EL1.EL0PTEN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

AArch32 System register CNTP_CTL is architecturally mapped to AArch64 System register CNTP_CTL_EL0.

Some or all RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. If the PE resets into EL3 using AArch32 they apply only to the Secure instance of the register. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

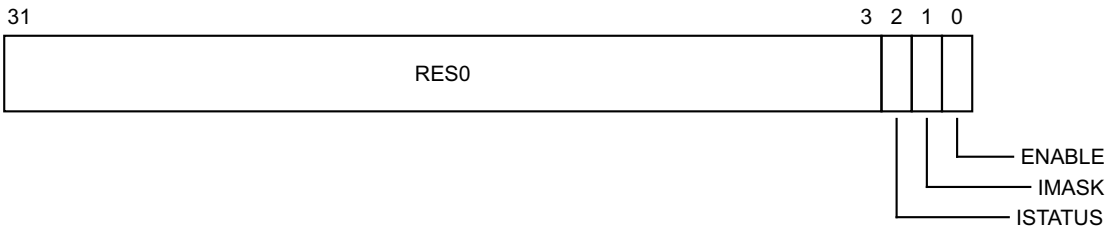
**Attributes**

CNTP_CTL is a 32-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The CNTP_CTL bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is asserted:

`0` Timer condition is not asserted.

`1` Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see *Operation of the CompareValue views of the timers* on page D6-1904 and *Operation of the TimerValue views of the timers* on page D6-1904.

This bit is read-only.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**IMASK, bit [1]**

Timer output signal mask bit. Permitted values are:

`0` Timer output signal is not masked.

`1` Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

`0` Timer disabled.

`1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL continues to count down.

**Note**

Disabling the output signal might be a power-saving option.

When this register has an architecturally-defined reset value, this field resets to `0`.

### Accessing the CNTP_CTL:

To access the CNTP_CTL:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
MRC p15,0,<Rt>,c14,c2,1 ; Read CNTP_CTL into Rt
MCR p15,0,<Rt>,c14,c2,1 ; Write Rt to CNTP_CTL
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1110 | 0010 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.8 CNTP_CVAL, Counter-timer Physical Timer CompareValue register

The CNTP_CVAL characteristics are:

**Purpose**

Holds the compare value for the EL1 physical timer.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

CNTP_CVAL(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | Config-RW | - | - | - | RW |

CNTP_CVAL(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | - | Config-RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

CNTP_CVAL is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | Config-RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CNTHCTL.PL1PCEN==0, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If CNTHCTL_EL2.EL1PCEN==0, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If CNTKCTL.PL0PTEN==0, PL0 accesses to this register are trapped to Undefined mode.

If CNTKCTL_EL1.EL0PTEN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

AArch32 System register CNTP_CVAL is architecturally mapped to AArch64 System register CNTP_CVAL_EL0.

RW fields in this register reset to architecturally UNKNOWN values.

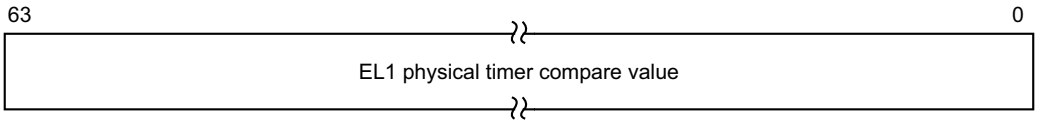
**Attributes**

CNTP_CVAL is a 64-bit register.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Field descriptions

The CNTP_CVAL bit assignments are:




**Bits [63:0]**

EL1 physical timer compare value.

### Accessing the CNTP_CVAL:

To access the CNTP_CVAL:

```
MRRC p15,2,<Rt>,<Rt2>,c14 ; Read CNTP_CVAL[31:0] into Rt and CNTP_CVAL[63:32] into Rt2
MCRR p15,2,<Rt>,<Rt2>,c14 ; Write Rt to CNTP_CVAL[31:0] and Rt2 to CNTP_CVAL[63:32]
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0010 | 1110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.9 CNTP_TVAL, Counter-timer Physical Timer TimerValue register

The CNTP_TVAL characteristics are:

**Purpose**

Holds the timer value for the EL1 physical timer. This provides a 32-bit downcounter.

**Usage constraints**

If EL3 is implemented and is using AArch32, there are separate Secure and Non-secure instances of this register.

CNTP_TVAL(S) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | Config-RW | - | - | - | RW |

CNTP_TVAL(NS) is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | - | Config-RW | RW | RW | - |

If EL3 is not implemented, or is implemented and is using AArch64, there is a single instance of this register.

CNTP_TVAL is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | Config-RW | RW |

Where the value of CNTP_CTL.ENABLE is 0:

- A write to CNTP_TVAL updates the register
- The value held in CNTP_TVAL continues to decrement
- A read of CNTP_TVAL returns an UNKNOWN value.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CNTHCTL.PL1PCEN==0, Non-secure EL0 and EL1 accesses to this register are trapped to Hyp mode.

If CNTHCTL_EL2.EL1PCEN==0, Non-secure EL0 and EL1 accesses to this register are trapped to EL2.

If CNTKCTL.PL0PTEN==0, PL0 accesses to this register are trapped to Undefined mode.

If CNTKCTL_EL1.EL0PTEN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

AArch32 System register CNTP_TVAL is architecturally mapped to AArch64 System register CNTP_TVAL_EL0.

RW fields in this register reset to architecturally UNKNOWN values.

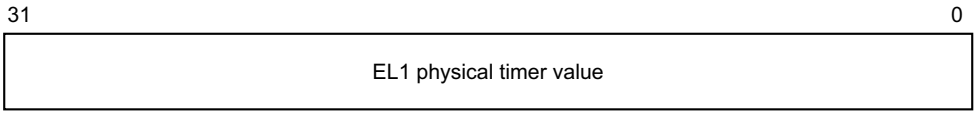

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Attributes**

CNTP_TVAL is a 32-bit register.

## Field descriptions

The CNTP_TVAL bit assignments are:




**Bits [31:0]**

EL1 physical timer value.

## Accessing the CNTP_TVAL:

To access the CNTP_TVAL:

```
MRC p15,0,<Rt>,c14,c2,0 ; Read CNTP_TVAL into Rt
MCR p15,0,<Rt>,c14,c2,0 ; Write Rt to CNTP_TVAL
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1110 | 0010 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.10 CNTPCT, Counter-timer Physical Count register

The CNTPCT characteristics are:

**Purpose**

Holds the 64-bit physical count value.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | Config-RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | Config-RO | RO |

When CNTKCTL.EL0PCTEN is set to 1, if CNTPCT is accessible from EL1 in the current Security state then it is also accessible from EL0 in that Security state.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CNTHCTL.PL1PCTEN==0, Non-secure EL0 and EL1 read accesses to this register are trapped to Hyp mode.

If CNTHCTL_EL2.EL1PCTEN==0, Non-secure EL0 and EL1 read accesses to this register are trapped to EL2.

If CNTKCTL.PL0PCTEN==0, PL0 read accesses to this register are trapped to Undefined mode.

If CNTKCTL_EL1.EL0PCTEN==0, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

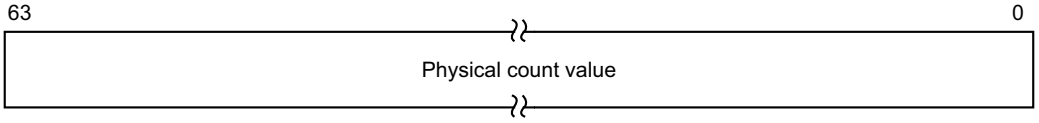
AArch32 System register CNTPCT is architecturally mapped to AArch64 System register CNTPCT_EL0.

**Attributes**

CNTPCT is a 64-bit register.

#### Field descriptions

The CNTPCT bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [63:0]**

Physical count value.

### Accessing the CNTPCT:

To access the CNTPCT:

```
MRRC p15,0,<Rt>,<Rt2>,c14 ; Read CNTPCT[31:0] into Rt and CNTPCT[63:32] into Rt2
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0000 | 1110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.11 CNTV_CTL, Counter-timer Virtual Timer Control register

The CNTV_CTL characteristics are:

**Purpose**

Control register for the virtual timer.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CNTKCTL.PL0VTEN==0, PL0 accesses to this register are trapped to Undefined mode.

If CNTKCTL_EL1.EL0VTEN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register CNTV_CTL is architecturally mapped to AArch64 System register CNTV_CTL_EL0.

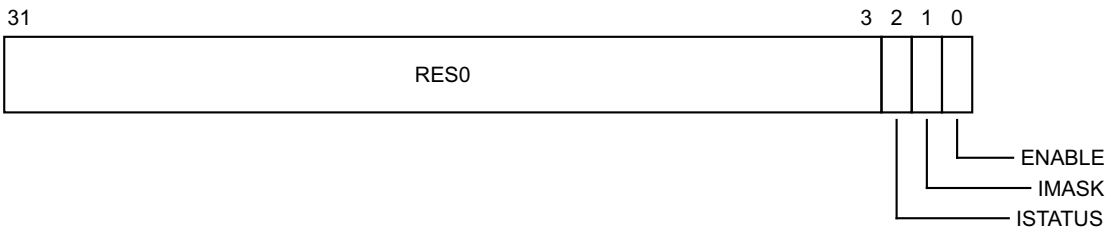
Some or all RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. Otherwise, RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTV_CTL is a 32-bit register.

#### Field descriptions

The CNTV_CTL bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:3]**

Reserved, RES0.

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is asserted:

0 Timer condition is not asserted.

1 Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see *Operation of the CompareValue views of the timers* on page D6-1904 and *Operation of the TimerValue views of the timers* on page D6-1904.

This bit is read-only.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**IMASK, bit [1]**

Timer output signal mask bit. Permitted values are:

0 Timer output signal is not masked.

1 Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

0 Timer disabled.

1 Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL continues to count down.

**Note**

Disabling the output signal might be a power-saving option.

When this register has an architecturally-defined reset value, this field resets to 0.

## Accessing the CNTV_CTL:

To access the CNTV_CTL:

```
MRC p15,0,<Rt>,c14,c3,1 ; Read CNTV_CTL into Rt
MCR p15,0,<Rt>,c14,c3,1 ; Write Rt to CNTV_CTL
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1110 | 0011 | 001 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.12 CNTV_CVAL, Counter-timer Virtual Timer CompareValue register

The CNTV_CVAL characteristics are:

**Purpose**

Holds the compare value for the virtual timer.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CNTKCTL.PL0VTEN==0, PL0 accesses to this register are trapped to Undefined mode.

If CNTKCTL_EL1.EL0VTEN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register CNTV_CVAL is architecturally mapped to AArch64 System register CNTV_CVAL_EL0.

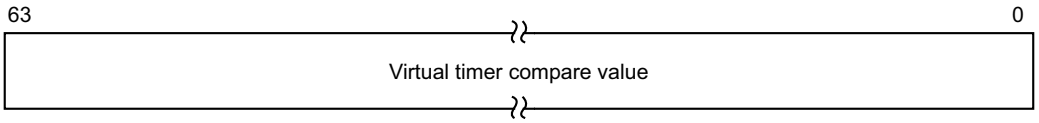
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTV_CVAL is a 64-bit register.

#### Field descriptions

The CNTV_CVAL bit assignments are:




**Bits [63:0]**

Virtual timer compare value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTV_CVAL:

To access the CNTV_CVAL:

```
MRRC p15,3,<Rt>,<Rt2>,c14 ; Read CNTV_CVAL[31:0] into Rt and CNTV_CVAL[63:32] into Rt2
MCRR p15,3,<Rt>,<Rt2>,c14 ; Write Rt to CNTV_CVAL[31:0] and Rt2 to CNTV_CVAL[63:32]
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0011 | 1110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.13 CNTV_TVAL, Counter-timer Virtual Timer TimerValue register

The CNTV_TVAL characteristics are:

**Purpose**

Holds the timer value for the virtual timer.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RW | Config-RW | RW | RW | RW | RW |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RW | RW | RW |

Where the value of CNTV_CTL.ENABLE is 0:

- A write to CNTV_TVAL updates the register
- The value held in CNTV_TVAL continues to decrement
- A read of CNTV_TVAL returns an UNKNOWN value.

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CNTKCTL.PL0VTEN==0, PL0 accesses to this register are trapped to Undefined mode.

If CNTKCTL_EL1.EL0VTEN==0, PL0 accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register CNTV_TVAL is architecturally mapped to AArch64 System register CNTV_TVAL_EL0.

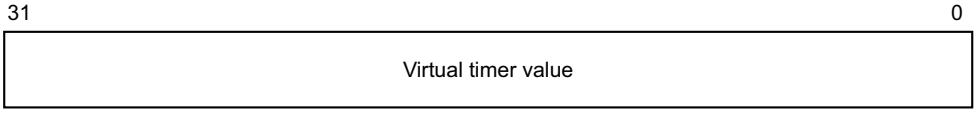
RW fields in this register reset to architecturally UNKNOWN values.

**Attributes**

CNTV_TVAL is a 32-bit register.

#### Field descriptions

The CNTV_TVAL bit assignments are:





Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [31:0]**

Virtual timer value.

### Accessing the CNTV_TVAL:

To access the CNTV_TVAL:

```
MRC p15,0,<Rt>,c14,c3,0 ; Read CNTV_TVAL into Rt
MCR p15,0,<Rt>,c14,c3,0 ; Write Rt to CNTV_TVAL
```

Register access is encoded as follows:

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 1111 | 000 | 1110 | 0011 | 000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.14 CNTVCT, Counter-timer Virtual Count register

The CNTVCT characteristics are:

**Purpose**

Holds the 64-bit virtual count value.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| Config-RO | Config-RO | RO | RO | RO | RO |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| Config-RO | RO | RO |

**Traps and Enables**

For a description of the prioritization of any generated exceptions, see *Synchronous exception prioritization for exceptions taken to AArch32 state* on page G1-3778 for exceptions taken to AArch32 state, and *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564 for exceptions taken to AArch64 state.

Subject to the prioritization rules:

If CNTKCTL.PL0VCTEN==0, PL0 read accesses to this register are trapped to Undefined mode.

If CNTKCTL_EL1.EL0VCTEN==0, PL0 read accesses to this register are trapped to EL1.

**Configurations**

There is one instance of this register that is used in both Secure and Non-secure states.

AArch32 System register CNTVCT is architecturally mapped to AArch64 System register CNTVCT_EL0.

The virtual count value is equal to the physical count value visible in CNTPCT minus the virtual offset visible in CNTVOFF.

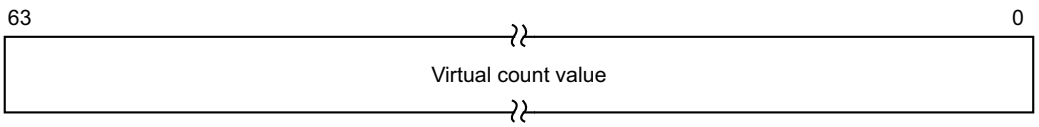
When EL2 is not implemented, CNTVOFF is RES0, and the value of this register is the same as the value of CNTPCT.

**Attributes**

CNTVCT is a 64-bit register.

#### Field descriptions

The CNTVCT bit assignments are:




**Bits [63:0]**

Virtual count value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTVCT:

To access the CNTVCT:

```
MRRC p15,1,<Rt>,<Rt2>,c14 ; Read CNTVCT[31:0] into Rt and CNTVCT[63:32] into Rt2
```

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0001 | 1110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### G6.5.15 CNTVOFF, Counter-timer Virtual Offset register

The CNTVOFF characteristics are:

**Purpose**

Holds the 64-bit virtual offset.

**Usage constraints**

If EL3 is implemented and is using AArch32, this register is accessible as follows:

| EL0 (NS) | EL0 (S) | EL1 (NS) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | RW | RW | - |

If EL3 is not implemented or EL3 is implemented and is using AArch64, this register is accessible as follows:

| EL0 | EL1 | EL2 (NS) |
|---|---|---|
| - | - | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

AArch32 System register CNTVOFF is architecturally mapped to AArch64 System register CNTVOFF_EL2.

If EL2 is not implemented, this register is RES0 from EL3.

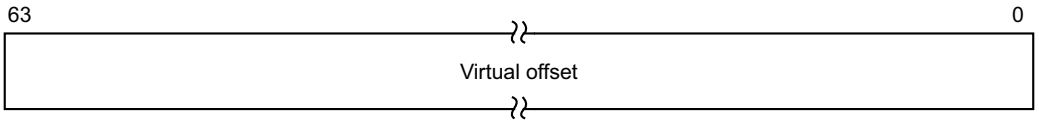
When EL2 is implemented and can use AArch32, on a reset into an Exception level that is using AArch32 this register resets to an architecturally UNKNOWN values.

**Attributes**

CNTVOFF is a 64-bit register.

#### Field descriptions

The CNTVOFF bit assignments are:




**Bits [63:0]**

Virtual offset.

#### Accessing the CNTVOFF:

To access the CNTVOFF:

```
MRRC p15,4,<Rt>,<Rt2>,c14 ; Read CNTVOFF[31:0] into Rt and CNTVOFF[63:32] into Rt2
MCRR p15,4,<Rt>,<Rt2>,c14 ; Write Rt to CNTVOFF[31:0] and Rt2 to CNTVOFF[63:32]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Register access is encoded as follows:

| coproc | opc1 | CRm |
|---|---|---|
| 1111 | 0100 | 1110 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Part H
## External Debug

# Chapter H1
# Introduction to External Debug

This chapter introduces the external debug components of ARMv8. It contains the following sections:

- *Introduction to external debug* on page H1-4816.
- *External debug* on page H1-4817.

**Note**

For information about self-hosted debug, see Chapter D2 *AArch64 Self-hosted Debug* and Chapter G2 *AArch32 Self-hosted Debug*.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H1.1 Introduction to external debug

ARMv8 supports both:

**Self-hosted debug**

The PE itself hosts a debugger. That is, developers developing software to run on the PE use debugger software running on the same PE.

**External debug**

The debugger is external to the PE. The debugging might be either on-chip, for example in a second PE, or off-chip, for example a JTAG debugger. External debug is particularly useful for:

- Hardware bring-up. That is, debugging during development when a system is first powered up and not all of the software functionality is available.
- PEs that are deeply embedded inside systems.

To support external debug, the ARM architecture defines required features that are called *external debug features*.

**Note**

**Definition of a debugger**

When the description of external debug in this part of the manual describes a *debugger* as controlling external debug this might be a second on-chip PE or a processor in an off-chip device such as a JTAG debugger.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H1.2 External debug

The following halting debug events are available in ARMv8:

- *Halting Step debug events* on page H3-4864:
  - The debugger can use this resource to make the PE step through code one line at a time.
- *Halt Instruction debug event* on page H3-4874:
  - This might occur when software executes the Halting software breakpoint instruction, `HLT`.
- *Exception Catch debug event* on page H3-4875:
  - This can be programmed to occur on all entries to a given Exception level.
- *External Debug Request debug event* on page H3-4878:
  - An embedded cross-trigger can signal this debug event.
- *OS Unlock Catch debug event* on page H3-4879:
  - This might occur when the state of the OS Lock changes from locked to unlocked.
- *Reset Catch debug events* on page H3-4880:
  - This might occur when the PE exits reset state.
- *Software Access debug event* on page H3-4881:
  - This can be programmed to occur when software tries to access the Breakpoint Value registers, the Breakpoint Control registers, the Watchpoint value registers, or the Watchpoint Control registers. It caused a trap to Debug state.

Halting debug events allow an external debugger to halt the PE. Breakpoints and watchpoints can also halt the PE. The PE then enters Debug state. When the PE is in Debug state:

- It stops executing instructions from the location indicated by the program counter, and is instead controlled through the external debug interface.
- The *Instruction Transfer Register,* ITR, passes instructions to the PE to execute in Debug state:
  - The ITR contains a single register, EDITR, and associated flow-control flags.
- The *Debug Communications Channel*, DCC, passes data between the PE and the debugger:
  - The DCC includes the data transfer registers, DTRRX and DTRTX, and associated flow-control flags.
  - Although the DCC is an essential part of Debug state operation, it can also be used in Non-debug state.
- The PE cannot service any interrupts in Debug state.

Chapter H2 *Debug State* describes Debug state in more detail.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Chapter H2
# **Debug State**

This chapter describes Debug state. It contains the following sections:


- *About Debug state* on page H2-4820.
- *Halting the PE on debug events* on page H2-4821.
- *Entering Debug state* on page H2-4828.
- *Behavior in Debug state* on page H2-4832.
- *Exiting Debug state* on page H2-4858.


**Note**

Table K13-1 on page K13-5708 disambiguates the general register references used in this chapter.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H2.1 About Debug state

In external debug, debug events allow an external debugger to halt the PE. The PE then enters Debug state. When the PE is in Debug state:

- It stops executing instructions from the location indicated by the program counter, and is instead controlled through the external debug interface.
- The *Instruction Transfer Register,* ITR, passes instructions to the PE to execute in Debug state.
- The *Debug Communications Channel*, DCC, passes data between the PE and the debugger.

The PE cannot service any interrupts in Debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H2.2 Halting the PE on debug events

For details of debug events see *Introduction to Halting debug events* on page H3-4862 and *Breakpoint and Watchpoint debug events* on page H2-4822.

On a debug event, the PE must do one of the following:

- Enter Debug state.
- Pend the debug event.
- Generate a debug exception.
- Ignore the debug event.

This behavior depends on both:

- Whether halting is allowed by the current state of the debug authentication interface. See *Halting allowed and halting prohibited*.
- The type of debug event and the programming of the debug control registers.
  - See *Halting debug events* for all Halting debug events.
  - See *Breakpoint and Watchpoint debug events* on page H2-4822 for Breakpoint and Watchpoint debug events.

See also *Other debug exceptions* on page H2-4823.

This means that behavior can be UNPREDICTABLE if the conditions change. See *Synchronization and Halting debug events* on page H3-4882.

*Summary of debug events and possible outcomes* on page H3-4862 summarizes the possible outcomes of each type of debug event.

### H2.2.1 Halting allowed and halting prohibited

Halting can be either allowed or prohibited:

- Halting is always prohibited in Debug state.
- Halting is always prohibited when `DoubleLockStatus()` == TRUE.
  - This means that OS Double Lock is locked, that is EDPRSR.DLK == 1.
- Halting is also controlled by the IMPLEMENTATION DEFINED authentication interface, and is prohibited when either:
  - The PE is in Non-secure state and `ExternalInvasiveDebugEnabled()` == FALSE.
  - The PE is in Secure state and `ExternalSecureInvasiveDebugEnabled()` == FALSE.

  **Note**

  See Appendix K2 *Recommended External Debug Interface* for more information on these functions.
- Otherwise, halting is allowed.

See *Pseudocode description of Halting on debug events* on page H2-4827.

### H2.2.2 Halting debug events

When a Halting debug event is generated, it causes entry to Debug state if both:

- Halting is allowed. See *Halting allowed and halting prohibited*.
- The Halting debug event is either:
  - A Halt Instruction debug event when Halting debug is enabled. This means that EDSCR.HDE == 1.
  - Not a Halt Instruction debug event.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

— An Halt Instruction debug event is the only Halting debug event that relies on EDSCR.HDE == 1. This is to prevent malicious code from causing an entry Debug state. EDSCR.HDE == 0 on a Cold reset.

— Halting on Breakpoint and Watchpoint Software debug events is also controlled by EDSCR.HDE. See *Breakpoint and Watchpoint debug events*.

— EDSCR.HDE can be written by software when the OS Lock is locked. Privileged code can use SDCR.TDOSA and HDCR.TDOSA to trap writes to these registers.

If a Halting debug event does not generate entry to Debug state because the conditions listed in this section do not hold, then:

- If the Halting debug event is a Halt Instruction debug event, it generates an Undefined Instruction exception.
- If the Halting debug event is an Exception Catch debug event or a Software Access debug event, it is ignored.
- In all other cases the Halting debug event is pended, meaning that:
  - The pending Halting debug event is recorded in EDESR.
  - The pending Halting debug event is taken when halting is allowed. See *Pending Halting debug events* on page H3-4882.

Pending Halting debug events are discarded by a Cold reset. The debugger can also force a pending event to be dropped by writing to EDESR. *Summary of actions from debug events* on page H2-4826 summarizes the possible outcome for each type of Debug event.

**Note**

Halting debug events never generate Debug exceptions.

### H2.2.3 Breakpoint and Watchpoint debug events

A breakpoint or watchpoint generates an entry to Debug state if all of the following conditions hold:

- Halting debug is enabled, that is EDSCR.HDE == 1.
- Halting is allowed. See *Halting allowed and halting prohibited* on page H2-4821.
- The OS Lock is unlocked, that is OSLSR.OSLK == 0.

The Address Mismatch breakpoint type is reserved when all of these conditions are met.

MDSCR_EL1.MDE or DBGDSCRext.MDBGen is ignored when determining whether to enter Debug state. A breakpoint or watchpoint that generates entry to Debug state is a Breakpoint or Watchpoint debug event and does not generate a debug exception.

A breakpoint or watchpoint that does not generate an entry to Debug state either:

- Generates a Breakpoint or Watchpoint exception.
- Is ignored.

**Note**

EDSCR.HDE is ignored when determining whether to generate a debug exception. The debug exception is suppressed only if the PE enters Debug state. This means that the use of Halting debug mode in Non-secure state does not affect the Exception model in Secure state.

See Chapter D2 *AArch64 Self-hosted Debug*, Chapter G2 *AArch32 Self-hosted Debug*, and the Note in *Other debug exceptions* on page H2-4823.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H2.2.4 Other debug exceptions

The following events never generate entry to Debug state:

- Software Breakpoint Instruction exceptions.
- Software Step exceptions.
- Vector Catch exceptions.

The behavior of these events is unchanged when Halting debug mode is enabled, that is when EDSCR.HDE == 1. This means that these events can do one of the following:

- They can generate a debug exception.
- They can be ignored.

For additional information, see Chapter D2 *AArch64 Self-hosted Debug* and Chapter G2 *AArch32 Self-hosted Debug*.

### H2.2.5 Debug state entry and debug event prioritization

The architecture does not define when asynchronous Halting debug events are taken, and therefore the prioritization of asynchronous debug events is IMPLEMENTATION DEFINED.

Synchronous Halting debug events do have a priority order.

The following are synchronous Halting debug events:

- Halting Step debug event.
- Halt Instruction debug event.
- Exception Catch debug event.
- Software Access debug event.
- Reset Catch debug event.

Each of these synchronous Halting debug events is treated as a synchronous exception generated by an instruction, or by the taking of an exception or reset. That is, the synchronous Halting debug event must be taken before any subsequent instructions are executed. Reset Catch debug events must be taken before the PE executes the instruction at the reset vector.

**Note**

Reset Catch and Exception Catch debug events can be generated asynchronously, because they can result from an asynchronous exception. However, if halting is allowed after the asynchronous exception has been processed, the Reset Catch or Exception Catch debug event is taken synchronously.

The Halting Step debug event is generated by the instruction after the stepped instruction. Therefore, if the stepped instruction generates any other synchronous exceptions or debug events, these are taken first.

OS Unlock Catch debug events are always pended and taken asynchronously.

Halting Step debug events and Reset Catch debug events might be pended and taken asynchronously at a later time.

The following list shows how the events are prioritized, with -2 being the highest priority.

**Note**

The priority numbering is the same as the numbering for AArch64 synchronous exception priorities listed in *Synchronous exception types, routing and priorities* on page D1-1563. The Debug events in this section with a negative priority are a higher priority than any synchronous exception. The debug events with fractional priorities have a priority between two or more exceptions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The priority for synchronous debug events is as follows:

**-2** Reset Catch debug event. See *Reset Catch debug events* on page H3-4880.

This debug event has a higher priority than the synchronous exceptions listed in *Synchronous exception types, routing and priorities* on page D1-1563.

**-1** Exception Catch debug event. See *Exception Catch debug event* on page H3-4875.

This debug event can be assigned one of two priorities. When it has a priority of -1, it has a higher priority than the synchronous exceptions listed in the Exception model. See *Exception Catch debug event* on page H3-4875.

**0** Halting Step debug event. See *Halting Step debug events* on page H3-4864.

This debug event has a higher priority than the synchronous exceptions listed in the Exception model.

**1** Software Step debug event. See *Software Step exceptions* on page D2-1686.

**1.5** Exception Catch debug event. See *Exception Catch debug event* on page H3-4875.

This debug event can be assigned one of two priorities, -1 or 1.5. See *Exception Catch debug event* on page H3-4875.

**2 - 3** These events are not debug events.

**4** Breakpoint exception or debug event or Address Matching Vector Catch exception. See *Breakpoint exceptions* on page D2-1653, and *Vector Catch exceptions* on page G2-3937.

These two debug events have the same priority.

**5 - 13** These events are not debug events.

**14** Halt Instruction debug event. See *Halt Instruction debug event* on page H3-4874.

**15 - 19** These events are not debug events.

**19.5** Software Access debug event. See *Software Access debug event* on page H3-4881.

**20 - 21** These events are not debug events.

**22** Watchpoint exception or debug event. See *Watchpoint exceptions* on page D2-1671 for exceptions taken from AArch64 state, or *Watchpoint exceptions* on page G2-3923 for exceptions taken from AArch32 state.

**23** This event is not a debug event.

For Reset Catch debug events and Halting Step debug events the priorities listed in this section only apply when halting is allowed at the time the event is generated. This means that the event is taken synchronously and not pended.

The prioritization of asynchronous Halting debug events, including pending Halting debug events taken asynchronously, is IMPLEMENTATION DEFINED. See *Taking Halting debug events asynchronously* on page H3-4883.

For more information on the prioritization of exceptions see *Synchronous exception types, routing and priorities* on page D1-1563.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Breakpoint debug events and Vector Catch exception

An Address Matching Vector Catch exception has the same priority as a Breakpoint debug event. See *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564.

The prioritization of these events is unchanged even if the breakpoint generates entry to Debug state instead of a Breakpoint exception. This means that if a single instruction generates both an Address Matching Vector Catch exception and a Breakpoint debug event, there is a CONSTRAINED UNPREDICTABLE choice of:

- The PE entering Debug state due to the Breakpoint debug event.
- A Vector Catch exception.

This only applies if all of the following are true:

- Halting debug is enabled.
- Halting is allowed.
- The OS Lock is unlocked.

An Exception Trapping Vector Catch exception must be generated immediately following the exception that generated it. This means that it does not appear in the priority table.

## H2.2.6 Forcing entry to Debug state

Entry to Debug state is normally precise, meaning that the PE cannot enter Debug state if it can neither complete nor abandon all currently executing instructions and leave the PE in a precise state.

A debugger can write a value of 1 to EDRCR.CBRRQ to allow imprecise entry to Debug state. An External Debug Request debug event must be pending before writing 1 to this bit. Support for this feature is OPTIONAL and it is IMPLEMENTATION DEFINED when it is effective at forcing entry to Debug state.

The PE ignores writes to this bit if either:

- External debugging is not enabled, meaning `ExternalInvasiveDebugEnabled()` == FALSE.
- Secure external debugging is not enabled, meaning `ExternalSecureInvasiveDebugEnabled()` == FALSE, and either:
  - EL3 is not implemented and the implemented Security state is Secure state.
  - EL3 is implemented.

Example H2-1 shows how entry to Debug state can be forced.

**Example H2-1 Forcing entry to Debug state**

---

The debugger pends an External Debug Request debug event through the CTI to halt a program that has stopped responding. However, the memory system is not responding and a memory access instruction cannot complete. This means that Debug state cannot be entered precisely. The debugger writes a value of 1 to EDRCR.CBRRQ. The PE cancels all outstanding memory accesses and enters Debug state. As some instructions might not have completed correctly, entry to Debug state is imprecise.

---

When Debug state is entered imprecisely, all memory access instructions executed through the ITR have UNPREDICTABLE behavior. The value of all registers is UNKNOWN, but might be useful for diagnostic purposes.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H2.2.7 Summary of actions from debug events

Table H2-1 on page H2-4826 shows the Software and Halting debug events. In Table H2-1 on page H2-4826 the columns have the following meaning:

**Debug event type**

This means the type of debug event where:

**Other software** Means one of:

- *Software Step exceptions* on page D2-1686.
- *Software Breakpoint Instruction exceptions* on page D2-1650.
- *Vector Catch exceptions* on page D2-1685 for AArch64 state or *Vector Catch exceptions* on page G2-3937 for AArch32 state.

**Other Halting** Means one of the following:

- *Halting Step debug events* on page H3-4864.
- *External Debug Request debug event* on page H3-4878.
- *Reset Catch debug events* on page H3-4880.
- *OS Unlock Catch debug event* on page H3-4879.

Other debug events are referred to explicitly.

**Authentication**

This means halting is allowed by the IMPLEMENTATION DEFINED external authentication interface. It is the result of one of the following pseudocode functions:

**In Secure state** `ExternalSecureInvasiveDebugEnabled()`.

**In Non-secure state** `ExternalInvasiveDebugEnabled()`.

**DLK** This is the value of EDPRSR.DLK. It indicates whether the OS Double Lock is locked, `DoubleLockStatus()` == TRUE.

**OSLK** This is the value of OSLSR.OSLK. It indicates whether the OS Lock is locked.

**HDE** This is the value of EDSCR.HDE. It indicates whether Halting debug is enabled.

The letter X in Table H2-1 indicates that the value can be either 0 or 1.

**Table H2-1 Debug authentication for external debug**

| Debug event type | Authentication | DLK | OSLK | HDE | Behavior |
|---|---|---|---|---|---|
| Other software | X | X | X | X | Handled by the Exception model |
| Breakpoint or Watchpoint debug event | X | 1 | X | X | Handled by the Exception model (ignored) |
| | X | 0 | 1 | X | Handled by the Exception model (ignored) |
| | FALSE | 0 | 0 | X | Handled by the Exception model |
| | TRUE | 0 | 0 | 0 | Handled by the Exception model |
| | TRUE | 0 | 0 | 1 | Entry to Debug state |
| Halt Instruction debug event | FALSE | X | X | X | UNDEFINED |
| | TRUE | 1 | X | X | UNDEFINED |
| | TRUE | 0 | X | 0 | UNDEFINED |
| | TRUE | 0 | X | 1 | Entry to Debug state |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table H2-1 Debug authentication for external debug (continued)**

| Debug event type | Authentication | DLK | OSLK | HDE | Behavior |
|---|---|---|---|---|---|
| Exception Catch debug event | FALSE | X | X | X | Ignored |
| | TRUE | 1 | X | X | Ignored |
| | TRUE | 0 | X | X | Entry to Debug state |
| Software Access debug event | FALSE | X | X | X | Ignored |
| | TRUE | 1 | X | X | Ignored |
| | TRUE | 0 | 1 | X | Ignored |
| | TRUE | 0 | 0 | X | Entry to Debug state |
| Other Halting | FALSE | X | X | X | Debug event is pended |
| | TRUE | 1 | X | X | Debug event is pended |
| | TRUE | 0 | X | X | Entry to Debug state |

### H2.2.8 Pseudocode description of Halting on debug events

The following pseudocode outlines the `Halting()`, `Restarting()`, `HaltingAllowed()`, and `HaltOnBreakpointOrWatchpoint()` functions.

```
// Halted()
// ========

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});                    // Halted

// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';                                   // Restarting

// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elsif IsSecure() then
        return ExternalSecureInvasiveDebugEnabled();
    else
        return ExternalInvasiveDebugEnabled();

// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H2.3 Entering Debug state

On entry to Debug state the preferred restart address and PSTATE are saved in DLR and DSPSR. The PE remains in the mode and security state from which it entered Debug state.

If EDRCR.CBRRQ has a value of 0, entry to Debug state is precise. If EDRCR.CBRRQ has a value of 1, then imprecise entry to Debug state is permitted.

If a Watchpoint debug event causes an entry to Debug state, the address of the access that generated the Watchpoint debug event is recorded in EDWAR. See *Determining the memory location that caused a Watchpoint exception* on page D2-1678 for an exception taken from AArch64 state, or *Determining the memory location that caused a Watchpoint exception* on page G2-3930 for an exception taken from AArch32 state.

Other than the effect on PSTATE and EDSCR, entry to Debug state is not a *Context synchronization operation*. The effects of entry to Debug state on PSTATE and EDSCR are synchronized.

### H2.3.1 Entering Debug state from AArch32 state

When entering Debug state from AArch32 state, the PE remains in AArch32 state. In AArch32 Debug state the PE executes T32 instructions, regardless of the value of PSTATE.T before entering Debug state.

To allow the debugger to determine the state of the PE, the current Execution state for all four Exception levels can be read from EDSCR.RW, and the current Exception level can be read from EDSCR.EL.

The current endianness state, PSTATE.E, is unchanged on entry to Debug state.

**Note**

- If EL1 is using AArch32 state, the current endianness state can differ from that indicated by SCTLR.EE.
- If EL2 is using AArch32 state, the current endianness state can differ from that indicated by HSCTLR.EE.
- On entry to Debug state from AArch32 state, PSTATE.SS is copied to DSPSR.SS, even though the PE remains in AArch32 state.

See also *Effect of entering Debug state on PSTATE* on page H2-4829.

### H2.3.2 Effect of entering Debug state on DLR and DSPSR

DLR is set to the preferred restart address for the debug event, that depends on the event type. The value of PSTATE is saved in DSPSR. For entry to Debug state from AArch32 state, the values saved in DSPSR.IT are always correct for the preferred restart address.

For synchronous Halting debug events, the preferred restart address is the address of the instruction that generated the debug event.

For asynchronous Halting debug events, including pending Halting debug events taken asynchronously, the preferred restart address is the address of the first instruction that must be executed on exit from Debug state.

This means that:

- For Breakpoint and Watchpoint debug events, the preferred restart address is the same as the preferred return address for a debug exception, as described in Chapter D2 *AArch64 Self-hosted Debug* and Chapter G2 *AArch32 Self-hosted Debug*.
- For Halt Instruction debug events DLR is set to the address of the `HLT` instruction and DSPSR.IT is correct for the `HLT` instruction.
- For Software Access debug events, DLR is set to the address of the accessing instruction and DSPSR.IT is correct for this instruction.
- For Halting Step debug events taken synchronously, DLR and DSPSR are set as the ELR and SPSR would be set for a Software Step exception. This is usually the address of, and PSTATE for, the instruction after the one that was stepped.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For Exception Catch debug events:
  - If the debug event is generated on taking an exception to a trapped Exception level, the DLR is set to the address of the exception vector the PE would have started fetching from. This is UNKNOWN if the VBAR for the Exception level has never been initialized. The DSPSR records the value of PSTATE after taking the exception. The exception catch occurs after the SPSR and the Link register are set, and the debugger can use these registers to determine where in the application program the exception occurred.

    **Note**

    Depending on the target Exception level and Execution state for the exception, the Link register is one of ELR_EL1, ELR_EL2, ELR_EL3, ELR_hyp, or LR (R14).

  - If the debug event is generated on an exception return to a trapped Exception level, the DLR is set to the target address of the exception return and the DSPSR records the value of PSTATE after the exception return.
- Reset Catch debug events taken synchronously behave like Exception Catch debug events.
- For Reset Catch debug events and Exception Catch debug events generated on reset to a trapped Exception level, the DLR is set to is set to the reset address and the DSPSR records the reset value of PSTATE.
- For pending Halting debug events and External Debug Request debug events, DLR is set to the address of the first instruction that must be executed on exit from Debug state and DSPSR.IT is correct for this instruction. See *Pending Halting debug events* on page H3-4882.

Normally DLR is aligned according to the instruction set state indicated in DSPSR. However, a debug event might be taken at a point where the PC is not aligned.

### H2.3.3 Effect of Debug state entry on System registers, the Event register, and exclusive monitors

Entering Debug state has no effect on System registers other than DLR and DSPSR. In particular, ESRs, FARs, and FSRs are not updated on entering Debug state. SCR is unchanged, even when entering Debug state from EL3.

Entering Debug state has no architecturally-defined effect on the Event Register and exclusive monitors.

**Note**

Entry to Debug state might set the Event Register or clear the exclusive monitors, or both. However, this is not a requirement, and debuggers must not rely on any implementation specific behavior.

Unless otherwise described in this reference manual, instructions executed in Debug state have their architecturally-defined effects on the System registers, the Event register, and exclusive monitors.

### H2.3.4 Effect of entering Debug state on PSTATE

The effect of an entry to Debug state on PSTATE is described in *Entering Debug state* on page H2-4828 and *Entering Debug state from AArch32 state* on page H2-4828.

PSTATE.{E, M, nRW, EL, SP} are unchanged on entry to Debug state.

PSTATE.IL is cleared to 0 on entry to Debug state, after being saved in DSPSR_EL0.

The other PSTATE fields are ignored and not observable in Debug state:

- PSTATE.{N, Z, C, V, Q, GE} are unchanged.
- PSTATE.{IT, T, SS, D, A, I, F} are set to UNKNOWN values, after being saved in DSPSR_EL0.

For more information see *Process state (PSTATE) in Debug state* on page H2-4832.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H2.3.5 Entering Debug state during loads and stores

The PE can enter Debug state during instructions that perform a sequence of memory accesses, as opposed to a single single-copy atomic access, because of a Watchpoint debug event. The effect of entering Debug state on such an instruction is the same as taking a Data Abort exception during such an instruction.

In addition, when executing in AArch64 state, the PE can enter Debug state during instructions that perform a sequence of memory accesses because of an External Debug Request debug event. The effect of entering Debug state on such an instruction is the same as taking an interrupt exception during such an instruction.

This applies to all memory types.

### H2.3.6 Entering Debug state and Software Step

When Software Step is active, a debug event that causes entry to Debug state behaves like an exception taken to an Exception level above the debug target Exception level. That is:

- If the instruction that is stepped generates a synchronous debug event that causes entry to Debug state, or an asynchronous debug event is taken before the step completes, the PE enters Debug state with DSPSR.SS set to 1.
- A pending Halting debug event or an asynchronous debug event can be taken after the step has completed. In this case the PE enters Debug state with DSPSR.SS set to 0.

In addition:

- If the instruction that is stepped generates an exception trapped by an Exception Catch debug event, the PE enters Debug state at the exception vector with DSPSR.SS set to 0. This is because PSTATE.SS is set to 0 by taking the exception.
- If the PE is reset, PSTATE.SS is reset to 0. If the following debug events are enabled, the PE enters Debug state with DSPSR.SS set to 0:
  - Reset Catch debug event at the reset Exception level.
  - Exception Catch debug event at the reset Exception level.
  - Halting Step debug event.
- If Halting Step is also active, then Halting Step and Software Step operate in parallel and can both become active-pending. In this case Halting step has a higher priority than Software step. This means that the PE enters Debug state and DSPSR.SS is set to 0.

### H2.3.7 Pseudocode description of entering Debug state

The following pseudocode shows the definition of the `DebugHalt()` function.

```
constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

The pseudocode for the `UpdateEDSCRFields()` function is as follows:

```
// UpdateEDSCRFields()
// ===================
// Update EDSCR processor state fields

UpdateEDSCRFields()
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;
        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        EDSCR.NS = if IsSecure() then '0' else '1';
        EDSCR.RW<1> = (if ELUsingAArch32(EL1) then '0' else '1');
        if PSTATE.EL != EL0 then
            EDSCR.RW<0> = EDSCR.RW<1>;
        else
            EDSCR.RW<0> = (if UsingAArch32() then '0' else '1');
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0') then
            EDSCR.RW<2> = EDSCR.RW<1>;
        else
            EDSCR.RW<2> = (if ELUsingAArch32(EL2) then '0' else '1');
        if !HaveEL(EL3) then
            EDSCR.RW<3> = EDSCR.RW<2>;
        else
            EDSCR.RW<3> = (if ELUsingAArch32(EL3) then '0' else '1');

    return;
```

The pseudocode for the `Halt()` function is as follows:

```
// Halt()
// ======

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    DLR_EL0 = ThisInstrAddr();
    DSPSR_EL0 = GetPSRFromPSTATE();
    DSPSR_EL0.SS = PSTATE.SS;                   // Always save PSTATE.SS

    EDSCR.ITE = '1';  EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0';                        // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = (if ExternalSecureInvasiveDebugEnabled() then '0' else '1');
    else
        assert EDSCR.SDD == '1';                // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if
    // UNKNOWN. PSTATE.{N,Z,C,V,Q,GE} are also not observable, but since these are not changed on
    // exception entry, this function also leaves them unchanged. PSTATE.{E,M,nRW,EL,SP} are
    // unchanged. PSTATE.IL is set to 0.
    if UsingAArch32() then
        PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1'; // PSTATE.J is RES0
    else
        PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                      // Signal entered Debug state
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    return;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H2.4 Behavior in Debug state

Instructions are executed in Debug state from the Instruction Transfer Register, ITR. The debugger controls which instructions are executed in Debug state by writing the instructions to the External Debug Instruction Transfer register, EDITR. The Execution state of the PE determines which instruction set is executed:

- If the PE is in AArch64 state it executes A64 instructions.
- If the PE is in AArch32 state it executes T32 instructions.

The PE does not execute A32 instructions in Debug state.

Some instructions are available only in Debug state. See *Debug state instructions, DCPS, DRPS, MRS, MSR* on page H2-4847. In Non-debug state these instructions are UNDEFINED.

### H2.4.1 Process state (PSTATE) in Debug state

PSTATE.{N, Z, C, V, Q, GE, IT, T, SS, D, A, I, F} are all ignored in Debug state:

- There are no conditional instructions in Debug state.
- In AArch32 state, the PE only executes T32 instructions and PSTATE.IT is ignored.
- Asynchronous exceptions and debug events are ignored.
- Software step is inactive.

Instructions executed in Debug state indirectly read PSTATE.{IL, E, M, nRW, EL, SP} as they would in Non-debug state.

### H2.4.2 Executing instructions in Debug state

The instructions executed in Debug state must be either A64 instructions or T32 instructions, depending on the current Execution state.

Each instruction falls into one of the following groups:

- Debug state instructions. These are instructions that are changed in Debug state. See *A64 instructions that are changed in Debug state* on page H2-4833 and *T32 instructions that are changed in Debug state* on page H2-4837.
- Instructions that are unchanged in Debug state. See *A64 instructions that are unchanged in Debug state* on page H2-4833 and *T32 instructions that are unchanged in Debug state* on page H2-4837.
- Instructions that are UNPREDICTABLE in Debug state. See *A64 instructions that are UNPREDICTABLE in Debug state* on page H2-4834 and *T32 instructions that are UNPREDICTABLE in Debug state* on page H2-4839.

All T32 instructions are treated as unconditional, regardless of PSTATE.IT. See *Process state (PSTATE) in Debug state*.

If EDSCR.SDD == 1 then an instruction executed in Non-secure state cannot cause entry into Secure state. See *Security in Debug state* on page H2-4845

#### Executing A64 instructions in Debug state

The following sections describe the behavior of the A64 instructions in Debug state:

- *A64 instructions that are changed in Debug state* on page H2-4833.
- *A64 instructions that are unchanged in Debug state* on page H2-4833.
- *A64 instructions that are UNPREDICTABLE in Debug state* on page H2-4834.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### A64 instructions that are changed in Debug state

The following A64 instructions are defined in Debug state, but are UNDEFINED in Non-debug state:

- DCPS

  **Note**

  DCPS can be UNDEFINED in certain conditions in Debug state. See *DCPS* on page H2-4847.

- DRPS
- MRS (DLR_EL0), MRS (DSPSR_EL0), MSR (DLR_EL0), MSR (DSPSR_EL0)

For more information see *Debug state instructions, DCPS, DRPS, MRS, MSR* on page H2-4847.

### A64 instructions that are unchanged in Debug state

The following list shows the instructions that are unchanged in Debug state:

**Any instruction that is UNDEFINED in Non-debug state**

This list of instructions excludes:

- Any instruction listed in *A64 instructions that are changed in Debug state*.
- Any instruction listed in *A64 instructions that are UNPREDICTABLE in Debug state* on page H2-4834 that is UNDEFINED because an enable or disable bit is not RES0 or RES1

**Instructions that move System or Special-purpose registers to or from a general-purpose register**

This list of instructions:

- Includes the instructions to transfer a general-purpose register to or from the DTR, which can be executed at any Exception level.
- Excludes PSTATE access instructions.

These instructions are:

- MRS <special_reg>, MSR <special_reg>

  **Note**

  This does not include NZCV, DAIF, DAIFSet, DAIFClr, SPSel, and CurrentEL.

- MRS <system_reg>, MSR <system_reg>

**Floating-point moves between a SIMD&FP register and a general-purpose register**

These instructions are:

- FMOV (between a general-purpose register and a single-precision register).
- FMOV (between a general-purpose register and a double-precision register).
- FMOV (between a general-purpose register and a SIMD element).

**SIMD moves between a SIMD&FP register and a general-purpose register**

These instructions are:

- INS (from a general-purpose register to a SIMD element).
- UMOV (from a SIMD element to a general-purpose register).

**Barriers** These instructions are:

- ISB.
- DSB.
- DMB.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Memory access instructions at various access sizes**

The following constraints apply:

- General purpose-registers only.
- One of the following addressing modes:
  - Unscaled (9-bit signed) immediate offset.
  - Immediate (9-bit signed) post-indexed.
  - Immediate (9-bit signed) pre-indexed.
  - Unprivileged (9-bit signed).
- Not literal.
- One of the following types:
  - (Single) register.
  - Exclusive.
  - Exclusive pair.
  - Acquire/Release.
  - Acquire/Release Exclusive.
  - Acquire/Release Exclusive pair.
- 32-bit and 64-bit target register variants.

These instructions are:

- `LDR`, `LDRB`, `LDRH`, `LDRSB`, `LDRSH`, `LDRSW` (immediate, not literal).
- `LDUR`, `LDURB`, `LDURH`, `LDURSB`, `LDURSH`, `LDURSW` (immediate).
- `LDTR`, `LDTRB`, `LDTRH`, `LDTRSB`, `LDTRSH`, `LDTRSW` (immediate).
- `LDAR`, `LDARB`, `LDARH`, `LDXR`, `LDXRB`, `LDXRH`, `LDAXR`, `LDAXRB`, `LDAXRH`.
- `LDXP`, `LDAXP`.
- `STR`, `STRB`, `STRH` (immediate).
- `STUR`, `STURB`, `STURH` (immediate).
- `STTR`,`STTRB`, `STTRH` (immediate).
- `STLR`, `STLRB`, `STLRH`, `STXR`, `STXRB`, `STXRH`, `STLXR`, `STLXRB`, `STLXRH`.
- `STXP`, `STLXP`.

**Move immediate to general-purpose register**

These instructions are:

- `MOVZ`, `MOVN`, `MOVK` (immediate).
- `MOV` (between a general-purpose register and the stack pointer).

**Cache maintenance, Send Event, NOP, and Clear Exclusive**

These instructions are:

- `IC`.
- `DC`.
- `TLBI`.
- `AT`.
- `SEV`, `SEVL`.
- `NOP` (no operation hint).
- `CLREX`.

### *A64 instructions that are UNPREDICTABLE in Debug state*

This subsection describes all instruction not listed in either:

- *A64 instructions that are changed in Debug state* on page H2-4833.
- *A64 instructions that are unchanged in Debug state* on page H2-4833.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

These instructions are CONSTRAINED UNPREDICTABLE in Debug state. In general, the permissible behaviors are:

- The instruction generates an Undefined Instruction exception.
- The instruction executes as a NOP.
- If the instruction reads the PC or PSTATE, it uses an UNKNOWN value.
- If the instruction modifies the PC or PSTATE, other than by advancing the PC to the sequentially next instruction, it sets DLR_EL0 and DSPSR_EL0 to UNKNOWN values.
- If the instruction is similar to a Debug state instruction, it executes as that Debug state instruction.
- The instruction has the same behavior as in Non-debug state.

The following list shows the permissible behaviors for A64 instruction in Debug state. An instruction might appear multiple times in the list, in which case the choice of permissible behaviors is any of those listed. An example of this is CCMP.

**Exception-generating instructions**

These instructions are:

- SVC.
- HVC.
- SMC.
- BRK.
- HLT.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- SVC behaves as DCPS1.
- HVC behaves as DCPS2.
- SMC behaves as DCPS3.
- They generate the exception that the instruction would generate in Non-debug state. The exception is taken as described in *Exceptions in Debug state* on page H2-4851

**Note**

SMC must not generate a Secure Monitor Call exception from Non-secure state if EDSCR.SDD is set to 1.

**Instructions that explicitly write to the PC (branches)**

These instructions are:

- B, B.cond, BL, BLR, BR, CBZ, CBNZ, RET, TBZ, TBNZ.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state without branching and set DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

**Exception return and related instructions**

These instructions are:

- ERET.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

- They execute as in Non-debug state without branching. They set DSPSR_EL0 and DLR_EL0 to UNKNOWN values, and either:
  - Execute as DRPS instead of performing an exception return, using UNKNOWN SPSR values.
  - Not change the Exception level.

**Instructions that explicitly modify PSTATE, other than DCPS and DRPS**

These instructions are:

- MSR DAIFSet (immediate), MSR DAIFClr (immediate), MSR SPSel (immediate).
- MSR NZCV (register), MSR DAIF (register), MSR SPSel (register).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state, setting DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

**Instructions that suspend execution**

These instructions are:

- WFE, WFI

These instructions behave in one of the following ways:

- Are UNDEFINED.
- Execute as a NOP.
- Generate a synchronous exception if the corresponding instruction would be trapped in Non-debug state. See *Configurable instruction enables and disables, and trap controls* on page D1-1575.

**Note**

This means that these instructions must not suspend execution.

**Instructions that read the PC**

These instructions are:

- LDR (literal), LDRSW (literal).
- ADR, ADRP.
- PRFM (literal).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state, using an UNKNOWN value for the PC operand.

**Instructions that read the PSTATE.{N, Z, C, V} or other PSTATE fields**

These instructions are:

- CSEL, CSINC, CSINV, CSNEG, CCMN, CCMP, FCSEL, FCCMP, FCCMPE.
- ADC, ADCS,SBC, SBCS.
- MRS NZCV, MRS DAIF, MRS SPSel, MRS CurrentEL.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state:
  - For the conditional operations and those using the PSTATE.C flag as an input, these instructions use an UNKNOWN value for the condition flag.
  - For the MRS instruction, they return an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Instructions that explicitly modify the PSTATE. {N, Z, C, V, Q, GE}**

These instructions are:

- `ADDS`, `SUBS`, `ADCS`, `SBCS`, `ANDS`, `BICS`, `CCMN`, `CCMP`.
- `FCMP`, `FCMPE`, `FCCMP`, `FCCMPE`.
- `MSR NZCV` (register).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state, setting DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

**All other instructions**

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They have the same behavior as in Non-debug state.

**Note**

This includes instructions defined as UNPREDICTABLE in Non-debug state. These instructions are UNPREDICTABLE in Debug state.

## Executing T32 instructions in Debug state

The following sections describe the behavior of the T32 instructions in Debug state:

- *T32 instructions that are changed in Debug state*.
- *T32 instructions that are unchanged in Debug state*.
- *T32 instructions that are UNPREDICTABLE in Debug state* on page H2-4839.

### *T32 instructions that are changed in Debug state*

The following T32 instructions are defined in Debug state, but are UNDEFINED in Non-debug state:

- `DCPS`

  **Note**

  `DCPS` can be UNDEFINED in certain conditions in Debug state. See *DCPS* on page H2-4847.

- `MRC p15,3,<Rt>,c4,c5,0` (DSPSR)
- `MCR p15,3,<Rt>,c4,c5,0` (DSPSR)
- `MRC p15,3,<Rt>,c4,c5,1` (DLR)
- `MCR p15,3,<Rt>,c4,c5,1` (DLR)

In addition, `ERET` is executed as DRPS in Debug state.

For more information see *Debug state instructions, DCPS, DRPS, MRS, MSR* on page H2-4847.

### *T32 instructions that are unchanged in Debug state*

The following list shows the instructions that are unchanged in Debug state. Any T32 instruction that uses the PC or APSR.{N, Z, C, V} as the source or destination register is not included in the list. Moreover, the list only includes the 32-bit T32 encodings.

**Any instruction that is UNDEFINED in Non-debug state**

The list of instructions:

- Excludes any instruction listed in *T32 instructions that are changed in Debug state*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Excludes any instruction listed in *T32 instructions that are UNPREDICTABLE in Debug state* on page H2-4839 that is UNDEFINED because an enable or disable bit is not RES0 or RES1

**Instructions that move System or Special-purpose registers to or from a general-purpose register**

The list of instructions:

- Includes the instructions to transfer a general-purpose register to or from the DTR, which can be executed at any Exception level.
- Excludes APSR and CPSR access instructions.
- Excludes instructions for accessing banked registers for the current mode.

These instructions are:

- `MRS <spec_reg>_<mode>`, `MSR <spec_reg>_<mode>`.

  **Note**

  This does not apply to cases which are UNPREDICTABLE in Non-debug state in the current mode.

- `MRC <system_reg>`, `MCR <system_reg>`

  **Note**

  This includes all allocated CP15 and CP14 System registers, other than an `MRC` move to APSR_nzcv.

- `MRS SPSR`, `MSR SPSR`
- `VMRS <vfp_system_reg>`, `VMSR <vfp_system_reg>`

  **Note**

  This includes all allocated Advanced SIMD and floating-point System registers, other than an a `VMRS` move to APSR_nzcv.

**Floating-point moves between a SIMD&FP register and a general-purpose register**

These instructions are:

- `VMOV` (between a general-purpose register and a single-precision register).
- `VMOV` (between a general-purpose register and a doubleword floating-point register).

**SIMD moves between a SIMD&FP register and a general-purpose register**

These instructions are:

- `VMOV` (between a general-purpose register and a scalar).

**Barriers** These instructions are:

- `ISB`.
- `DSB`.
- `DMB`.

**Memory access instructions at various access sizes**

The following constraints apply:

- General purpose-registers only.
- One of the following addressing modes:
  - Immediate (8-bit or 12-bit) offset.
  - Immediate (8-bit) post-indexed.
  - Immediate (8-bit) pre-indexed.
  - Unprivileged (8-bit).
- Not literal.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- One of the following types:
  - (Single) register.
  - Dual.
  - Exclusive.
  - Exclusive doubleword.
  - Acquire/Release.
  - Acquire/Release Exclusive.
  - Acquire/Release Exclusive doubleword.

These instructions are:

- `LDR.W`, `LDRB.W`, `LDRH.W`, `LDRD`, `LDRSB.W`, `LDRSH.W` (immediate, not literal).
- `LDRT`, `LDRBT`, `LDRHT`, `LDRSBT`, `LDRSHT` (immediate).
- `LDREX`, `LDREXB`, `LDREXH`, `LDA`, `LDAB`, `LDAH`, `LDAEX`, `LDAEXB`, `LDAEXH`.
- `LDREXD`, `LDAEXD`.
- `STR.W`, `STRB.W`, `STRH.W`, `STRD` (immediate).
- `STRT`, `STRBT`, `STRHT` (immediate).
- `STREX`, `STREXB`, `STREXH`, `STL`, `STLB`, `STLH`, `STLEX`, `STLEXB`, `STLEXH`.
- `STREXD`, `STLEXD`.

**Move to general-purpose register**

These instructions are:

- `MOVW`, `MOVT` (immediate).

**Cache maintenance, Send Event, `NOP`, and Clear Exclusive**

These instructions are:

- `ICIALLU`, `ICIALLUIS`, `ICIMVAU` (CP15 operations).
- `DCCIMVAC`, `DCCISW`, `DCCMVAC`, `DCCMVAU`, `DCCSW`, `DCIMVAC`, `DCISW` (CP15 operations).
- `TLBIALL`, `TLBIALLH`, `TLBIALLHIS`, `TLBIALLIS`, `TLBIALLNSNH`, `TLBIALLNSNHIS`, `TLBIASID`, `TLBIASIDIS`, `TLBIIPAS2`, `TLBIIPAS2IS`, `TLBIIPAS2L`, `TLBIIPAS2LIS`, `TLBIMVA`, `TLBIMVAA`, `TLBIMVAAIS`, `TLBIMVAAL`, `TLBIMVAALIS`, `TLBIMVAH`, `TLBIMVAHIS`, `TLBIMVAIS`, `TLBIMVAL`, `TLBIMVALH`, `TLBIMVALHIS`, `TALBIMVALIS` (CP15 operations).
- `ATS12NSOPR`, `ATS12NSOPW`, `ATS12NSOUR`, `ATS12NSOUW`, `ATS1CPR`, `ATS1CPW`, `ATS1CUR`, `ATS1CUW`, `ATS1HR`, `ATS1HW` (CP15 operations).
- `BPIALL`, `BPIALLIS`, `BPIMVA` (CP15 operations).
- `SEV.W`, `SEVL.W`.
- `NOP.W` (no operation hint).
- `CLREX`.

### *T32 instructions that are UNPREDICTABLE in Debug state*

This subsection describes all instruction not listed in either:

- *T32 instructions that are changed in Debug state* on page H2-4837.
- *T32 instructions that are unchanged in Debug state* on page H2-4837.

These instructions are CONSTRAINED UNPREDICTABLE in Debug state. In general, the permissible behaviors are:

- The instruction generates an Undefined Instruction exception.
- The instruction executes as a `NOP`.
- If the instruction reads the PC or PSTATE, it uses an UNKNOWN value.
- If the instruction modifies the PC or PSTATE, other than by advancing the PC to the sequentially next instruction, it sets DLR and DSPSR to UNKNOWN values.
- If the instruction is similar to a Debug state instruction, it executes as that Debug state instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The instruction has the same behavior as in Non-debug state.

The following list shows the permissible behaviors for T32 instruction in Debug state. An instruction might appear multiple times in the list, in which case the choice of permissible behaviors is any of those listed.

**Exception-generating instructions**

These instructions are:

- SVC.
- HVC.
- SMC.
- UDF.
- BKPT.
- HLT.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- SVC behaves as DCPS1.
- HVC behaves as DCPS2.
- SMC behaves as DCPS3.
- They generate the exception the instruction would generate in Non-debug state. The exception is taken as described in *Exceptions in Debug state* on page H2-4851

**Note**

SMC must not generate a Secure Monitor Call exception from Non-secure state if EDSCR.SDD is set to 1.

**Instructions that explicitly write to the PC (branches)**

These instructions are:

- B, B (conditional), CBZ, CBNZ BL.
- BX, BLX (register or immediate).
- BXJ, TBB, TBH.
- MOV pc and related instructions.
- LDR pc, LDM (with a register list includes the PC), POP (with a register list that includes the PC).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state without branching and set DSPSR and DLR to UNKNOWN values.

**Exception return and related instructions**

These instructions are:

- SRS, RFE, SUBS pc, 1r, and related instructions.

**Note**

The T32 ERET instruction is decoded as DRPS and is not included in this list.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- They execute as in Non-debug state without branching, setting DSPSR_EL0 and DLR_EL0 to UNKNOWN values, and either:
  - Execute as DRPS instead of performing an exception return, using UNKNOWN SPSR values.
  - Not changing Exception level or PE mode.

**Note**

The T32 ERET instruction is decoded as DRPS and is not included in this list.

**Instructions that explicitly modify PSTATE, other than DCPS and DRPS**

These instructions are:

- CPS, SETEND, IT.
- MSR CPSR (register or immediate).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state, setting DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

**Instructions that suspend execution**

These instructions are:

- WFE, WFI.

These instructions behave in one of the following ways:

- Are UNDEFINED.
- Execute as a NOP.
- Generate a synchronous exception if the corresponding instruction would be trapped in Non-debug state. See *Configurable instruction enables and disables, and trap controls* on page G1-3846.

  **Note**

  This means that these instructions must not suspend execution.

**Instructions that read the PC**

These instructions are:

- LDR (literal), LDRB (literal), LDRH (literal), LDRSB (literal), LDRSH (literal).
- ADR, ADRL, ADRH.
- PLD (literal), PLI (literal).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state using an UNKNOWN value for the PC operand.

**Instructions that read PSTATE.{N, Z, C, V} or other PSTATE fields**

These instructions are:

- SEL, VSEL.
- ADC, SBC, all instructions with an RRX shift.
- MRS CPSR.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- They execute as in Non-debug state:
  - For the conditional operations and those using the PSTATE.C flag as an input, these instructions use an UNKNOWN value for the condition flag.
  - For the `MRS` instruction, they return an UNKNOWN value

**Instructions that explicitly modify PSTATE.{N, Z, C, V, Q, GE}**

These instructions are:

- `CMP, TST, TEQ, CMN.`
- `<opc>S.`
- `MRC p14,0,APSR_nzcv,c0,c1,0` (DBGDSCRint).
- `MSR CPSR`, `MSR APSR`, (register or immediate).
- `VMRS APSR_nzcv,FPSCR.`
- `QADD, QDADD, QSUB, QDSUB.`
- `SMLABB. SMLABT, SMLATB, SMLATT, SMLAD, SMLAWB, SMLAWT, SMLSD, SMUAD.`
- `SSAT, SSAT16, USAT, USAT16.`
- `SADD, SADD8, SADD16, SASX, SSAX, SSUB, SSUB8, SSUB16.`
- `UADD, UADD8, UADD16, UASX, USAX, USAUB, USUN8, USUB16.`

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state, setting DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

**All other instructions**

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They have the same behavior as in Non-debug state.

**Note**

This includes instructions defined as UNPREDICTABLE in Non-debug state. These instructions are UNPREDICTABLE in Debug state. This includes some T32 instructions that specify R15 as a destination or source register, such as:

```
MOV.W R15, #<uimm16>
LDREX R15, [Rn]
```

Appendix K1 *Architectural Constraints on UNPREDICTABLE behaviors* describes the CONSTRAINED UNPREDICTABLE behavior for these instructions. In Debug state these CONSTRAINED UNPREDICTABLE choices are further restricted:

- Instructions that specify R15 as a destination register:
  - Are not permitted to branch, because the architecture does not define a branch operation in Debug state.
  - Might set DLR and DSPSR to UNKNOWN values.
  - Might have any of the other permitted behaviors.
- Instructions that specify R15 as a source operand:
  - Cannot use PC + offset, because there is no architecturally-defined PC in Debug state.
  - Might have any of the other permitted behaviors, including using an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H2.4.3 Decode tables

The syntax in the tables is defined as follows:

**1** The bit has a fixed value of 1.

**0** The bit has a fixed value of 0.

**!=** The field has any value other than the value or values specified. The field might be an encoding field in the instruction whose value is supplied by the debugger.

**Note**

The instruction encodings in Chapter C6 *A64 Base Instruction Descriptions* and Chapter F6 *T32 and A32 Base Instruction Set Instruction Descriptions* might show these bits as (0) or (1). A debugger must set these bits to 0 or 1, as appropriate.

Any other value indicates an encoding field in the instruction whose value is supplied by the debugger. Some values might be reserved or UNDEFINED, in which case the instruction is UNDEFINED or UNPREDICTABLE in Debug state, as it is in Non-debug state.

For more information about the instruction encodings, see:

- Chapter C6 *A64 Base Instruction Descriptions*.
- Chapter F6 *T32 and A32 Base Instruction Set Instruction Descriptions*.

For information about the syntax used in Table H2-2, Table H2-3, Table H2-4 on page H2-4844, and Table H2-5 on page H2-4845, see:

- *Common syntax terms* on page C1-121.
- *Assembler symbols* on page F2-2443.

Table H2-2 shows the A64 instructions that are modified in Debug state. For details of how these are packed in EDITR, see *EDITR, External Debug Instruction Transfer Register* on page H9-5016.

**Table H2-2 Modified A64 instructions in Debug state**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | Description |
|---|---|---|---|---|---|---|---|---|
| 1 1 0 1 | 0 1 0 0 | 1 0 1 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 opt | DCPS<opt> |
| 1 1 0 1 | 0 1 0 1 | 0 0 L 1 | 1 0 1 1 | 0 1 0 0 | 0 1 0 1 | 0 0 0 | Rt | MRS\|MSR accessing DSPSR |
| 1 1 0 1 | 0 1 0 1 | 0 0 L 1 | 1 0 1 1 | 0 1 0 0 | 0 1 0 1 | 0 0 1 | Rt | MRS\|MSR accessing DLR |
| 1 1 0 1 | 0 1 1 0 | 1 0 1 1 | 1 1 1 1 | 0 0 0 0 | 0 0 1 1 | 1 1 1 0 | 0 0 0 0 | DRPS |

Table H2-3 show the T32 instructions that are modified in Debug state, with the first halfword on the left side and the second halfword on the right side. For details of how these are packed in EDITR, See *EDITR, External Debug Instruction Transfer Register* on page H9-5016.

**Table H2-3 Modified T32 instructions in Debug state**

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | Description |
|---|---|---|---|---|---|---|---|---|
| 1 1 1 0 | 1 1 1 0 | 0 1 1 L | 0 1 0 0 | Rt | 1 1 1 1 | 0 0 0 1 | 0 1 0 1 | MRC\|MCR accessing DSPSR |
| 1 1 1 0 | 1 1 1 0 | 0 1 1 L | 0 1 0 0 | Rt | 1 1 1 1 | 0 0 1 1 | 0 1 0 1 | MRC\|MCR accessing DLR |
| 1 1 1 1 | 0 0 1 1 | 1 1 0 1 | 1 1 1 0 | 1 0 0 0 | 1 1 1 1 | 0 0 0 0 | 0 0 0 0 | ERET (decoded as DRPS) |
| 1 1 1 1 | 0 1 1 1 | 1 0 0 0 | 1 1 1 1 | 1 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 opt | DCPS<opt> |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H2-4 lists the A64 instructions that are unchanged in Debug state.

**Table H2-4 A64 instructions that are unchanged in Debug state**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | Description |
|---|---|---|---|---|---|---|---|---|
| sf 0 0 1 | 0 0 0 1 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 1 1 | 1 1 1 | Rd | MOV <Rn>,SP |
| sf 0 0 1 | 0 0 0 1 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 | Rn | 1 1 1 1 1 | MOV SP,<Rn> |
| sf opc 1 | 0 0 1 0 | 1 hw | imm16 | | | | Rd | MOVN, MOVK, MOVZ |
| 1 1 0 1 | 0 1 0 1 | 0 0 0 0 | 0 0 1 1 | 0 0 1 0 | 0 0 0 0 | 0 0 0 1 | 1 1 1 1 | NOP |
| 1 1 0 1 | 0 1 0 1 | 0 0 0 0 | 0 0 1 1 | 0 0 1 0 | 0 0 0 0 | 1 0 L 1 | 1 1 1 1 | SEV, SEVL |
| 1 1 0 1 | 0 1 0 1 | 0 0 0 0 | 0 0 1 1 | 0 0 1 1 | 0 0 0 0 | 0 1 0 1 | 1 1 1 1 | CLREX |
| 1 1 0 1 | 0 1 0 1 | 0 0 0 0 | 0 0 1 1 | 0 0 1 1 | option | 1 opc 1 | 1 1 1 1 | DSB, DMB, ISB |
| 1 1 0 1 | 0 1 0 1 | 0 0 0 0 | 1 op1 | CRn | CRm | op2 | Rt | IC, DC, TLBI, AT |
| 1 1 0 1 | 0 1 0 1 | 0 0 L 1 | 0 op1 | CRn | CRm | op2 | Rt | MRS\|MSR accessing System register |
| 1 1 0 1 | 0 1 0 1 | 0 0 L 1 | 1 op1 | !=0100 | CRm | op2 | Rt | MRS\|MSR accessing System register |
| 1 1 0 1 | 0 1 0 1 | 0 0 L 1 | 1 op1 | 0 1 0 0 | !=0010 | op2 | Rt | MRS\|MSR accessing Special-purpose register |
| size 0 0 | 1 0 0 0 | o2 L 0 | Rs | o0 Rt2 | | Rn | Rt | LD(A\|X\|AX)R{B\|H}, ST(L\|X\|LX)R{B\|H} |
| size 0 0 | 1 0 0 0 | o2 L 1 | Rs | o0 Rt2 | | Rn | Rt | LD{A}XP, ST{L}XP |
| !=11 1 1 | 1 0 0 0 | opc 0 | imm9 | | 0 0 | Rn | Rt | LDUR{B\|H\|SB\|SH\|SW}, STUR{B\|H} |
| 1 1 1 1 | 1 0 0 0 | !=10 0 | imm9 | | 0 0 | Rn | Rt | LDUR, STUR |
| size 1 1 | 1 0 0 0 | opc 0 | imm9 | | 1 0 | Rn | Rt | LDTR{B\|H\|SB\|SH\|SW}, STTR{B\|H} |
| size 1 1 | 1 0 0 0 | opc 0 | imm9 | | P 1 | Rn | Rt | LDR{B\|H\|SB\|SH\|SW}, STR{B\|H} |
| 0 1 0 0 | 1 1 1 0 | 0 0 0 | imm5 | 0 0 0 1 | 1 1 | Rn | Rd | INS <Vd>.<Ts>[<index>],<Rn> |
| 0 Q 0 0 | 1 1 1 0 | 0 0 0 | imm5 | 0 0 1 1 | 1 1 | Rn | Rd | UMOV <Rd>,<Vd>.<Ts>[<index>] |
| 0 0 0 1 | 1 1 1 0 | 0 0 1 0 | 0 1 1 op | 0 0 0 0 | 0 0 | Rn | Rd | FMOV <Sd>,<Wn>, FMOV <Wd>,<Sn> |
| 1 0 0 1 | 1 1 0 1 | 1 0 1 0 | 0 1 1 op | 0 0 0 0 | 0 0 | Rn | Rd | FMOV <Dd>,<Wn>, FMOV <Rd>,<Dn> |
| 1 0 0 1 | 1 1 1 0 | 1 0 1 0 | 1 1 1 op | 0 0 0 0 | 0 0 | Rn | Rd | FMOV <Vd>.D[1],<Wn><br>FMOV <Wd>,<Vn>.D[1] |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

Table H2-5 lists the T32 instructions that are unchanged in Debug state. It shows the T32 instructions with the first halfword on the left side and the second halfword on the right side.

**Table H2-5 T32 instructions that are unchanged in Debug state**

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | Description |
|---|---|---|---|---|---|---|---|---|
| 1 1 1 0 | 1 1 0 0 | 0 1 0 op | Rt2 | Rt | 1 0 1 1 | 0 0 M 1 | Vm | VMOV <Dm>,<Rt>,<Rt2><br>VMOV <Rt>,<Rt2>,<Dm> |
| 1 1 1 0 | 1 1 1 0 | 0 0 0 op | Vn | Rt | 1 0 1 0 | N 0 0 1 | 0 0 0 0 | VMOV <Sn>,<Rt>, VMOV <Rt>,<Sn> |
| 1 1 1 0 | 1 1 1 0 | 0 opc 0 | Vd | Rt | 1 0 1 1 | D opc2 1 | 0 0 0 0 | VMOV.<size> <Dd>[<x>],<Rt> |
| 1 1 1 0 | 1 1 1 0 | U opc 1 | Vn | Rt | 1 0 1 1 | D opc2 1 | 0 0 0 0 | VMOV.<dt> <Rt>,<Dd>[<x>] |
| 1 1 1 0 | 1 1 1 0 | 1 1 1 op | reg | Rt | 1 0 1 0 | 0 0 0 1 | 0 0 0 0 | VMRS, VMSR |
| 1 1 1 0 | 1 1 0 0 | 0 1 0 op | Rt2 | Rt | 1 1 1 cp | opc1 | CRm | MCRR\|MRCC accessing System registers |
| 1 1 1 0 | 1 1 1 0 | opc1 op | CRn | Rt | 1 1 1 cp | opc2 1 | CRm | MCR\|MRC accessing System registers |
| 1 1 1 0 | 1 0 0 0 | 0 1 0 L | Rn | Rt | Rd | imm8 | | LDREX, STREX |
| 1 1 1 0 | 1 0 0 0 | 1 1 0 L | Rn | Rt | Rt2 | 0 1 op3 | Rd | LDREX(B\|H\|D), STREX(B\|H\|D) |
| 1 1 1 0 | 1 0 0 0 | 1 1 0 L | Rn | Rt | Rt2 | 1 op3 | Rd | LDA{EX}{B\|H\|D}, STL{EX}{B\|H\|D} |
| 1 1 1 0 | 1 0 0 !=0x10 !=xx0x | L | !=1111 | Rt | Rt2 | imm8 | | LDRD, STRD |
| 1 1 1 1 | 0 i 1 0 | T 1 0 0 | imm4 | 0 imm3 | Rd | imm8 | | MOVW, MOVT |
| 1 1 1 1 | 0 0 1 1 | 1 0 0 R | Rn | 1 0 0 0 | M1 | 0 0 1 M | 0 0 0 0 | MSR <spec_reg> _<mode>,<Rn> |
| 1 1 1 1 | 0 0 1 1 | 1 0 0 1 | Rn | 1 0 0 0 | 1 1 1 1 | 0 0 0 0 | 0 0 0 0 | MSR SPSR, <Rn> |
| 1 1 1 1 | 0 0 1 1 | 1 0 1 0 | 1 1 1 1 | 1 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | NOP.W |
| 1 1 1 1 | 0 0 1 1 | 1 0 1 0 | 1 1 1 1 | 1 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 1 0 L | SEV.W, SEVL.W |
| 1 1 1 1 | 0 0 1 1 | 1 0 1 1 | 1 1 1 1 | 1 0 0 0 | 1 1 1 1 | 0 0 1 0 | 1 1 1 1 | CLREX |
| 1 1 1 1 | 0 0 1 1 | 1 0 1 1 | 1 1 1 1 | 1 0 0 0 | 1 1 1 1 | 0 1 op | option | DSB, DMB, ISB |
| 1 1 1 1 | 0 0 1 1 | 1 1 1 R | M1 | 1 0 0 0 | Rd | 0 0 1 M | 0 0 0 0 | MRS <Rd>,<spec_reg> _<mode> |
| 1 1 1 1 | 0 0 1 1 | 1 1 1 1 | 1 1 1 1 | 1 0 0 0 | Rd | 0 0 0 0 | 0 0 0 0 | MRS <Rd>,SPSR |
| 1 1 1 1 | 1 0 0 0 | 1 op1 0 | Rn | Rt | imm12 | | | STR{B\|H}.W (12-bit immediate) |
| 1 1 1 1 | 1 0 0 0 | 0 op1 0 | Rn | Rt | 1 P U W | imm8 | | STR{B\|H\|}{T} (8-bit immediate) |
| 1 1 1 1 | 1 0 0 S | 1 op1 1 | !=1111 | !=1111 | imm12 | | | LDR{SB\|SH\|B\|H}.W (12-bit immediate) |
| 1 1 1 1 | 1 0 0 S | 0 op1 1 | !=1111 | !=1111 | 1 P U W | imm8 | | LDR{SB\|SH\|B\|H}{T} (8-bit immediate) |

## H2.4.4 Security in Debug state

If EL3 is implemented or the implemented Security state is Secure state, security in Debug state is governed by the Secure debug disabled flag, EDSCR.SDD.

**On entry to Debug state**

If entering in Secure state, EDSCR.SDD is set to 0. Otherwise EDSCR.SDD is set to the inverse of `ExternalSecureInvasiveDebugEnabled()`. That is:

- If `ExternalSecureInvasiveDebugEnabled()` == TRUE, EDSCR.SDD is set to 0.
- If `ExternalSecureInvasiveDebugEnabled()` == FALSE, EDSCR.SDD is set to 1.

**Note**

Normally, if `ExternalSecureInvasiveDebugEnabled()` == FALSE then halting is prohibited and it is not possible to enter Debug state from Secure state. However, because changes to the authentication signals require a *Context synchronization operation* to guarantee their effect, there is a period during which the PE might halt even though the authentication signals prohibit halting.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**In Debug state**

The value of EDSCR.SDD does not change, even if `ExternalSecureInvasiveDebugEnabled()` changes.

**Note**

- DBGAUTHSTATUS_EL1.{SNID, SID, NSNID, NSID} are not frozen in Debug state.
- If EDSCR.SDD set to 1 in Debug state, then there is no means to enter Secure state from Non-secure state. In this case it is impossible for the PE to be in Secure state. This is a general principle of behavior in Debug state.

**In Non-debug state**

EDSCR.SDD returns the inverse of `ExternalSecureInvasiveDebugEnabled()`. If the authentication signals that control `ExternalSecureInvasiveDebugEnabled()` change, a *Context synchronization operation* is required to guarantee their effect.

**Note**

- In Non-debug state, EDSCR.SDD is unaffected by the Security state of the PE.
- A *Context synchronization operation* is also required to guarantee that changes in the authentication signals are visible in DBGAUTHSTATUS_EL1.{SNID, SID, NSNID, NSID}.

If EL3 is not implemented and the implemented Security state is Non-Secure state, EDSCR.SDD is RES1.

### H2.4.5 Privilege in Debug state

The only additional privileges offered to Debug state are:

- The privilege to execute *Debug state instructions, DCPS, DRPS, MRS, MSR* on page H2-4847.
- The privilege to execute DTR access instructions regardless of the Exception level and traps.

In Non-debug state, the Debug state instructions are UNDEFINED, except for the T32 `DRPS` instruction. The T32 `DRPS` instruction uses the encoding of the Non-debug T32 `ERET` instruction.

In Debug state, the Debug state instructions can be executed at any Exception level. However, there are some cases where the instructions are UNDEFINED. For more information, see *Debug state instructions, DCPS, DRPS, MRS, MSR* on page H2-4847. These instructions generate an Undefined Instruction exception when they are UNDEFINED. If this Undefined Instruction exception is taken to an Exception level using AArch64, it is reported using ESR_ELx.EC, with the code `0x00`, and if taken to AArch32 Hyp mode, reported using HSR.EC `of 0x00`.

The DTR access instructions can be executed at any Exception level, including EL0, regardless of any control register settings that might force these instructions to be UNDEFINED or trapped in Non-debug state. These instruction are:

- The `MRS` and `MSR` instructions that access DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0 in AArch64 state.
- The `MRC` and `MCR` instructions that access DBGDTRTXint and DBGDTRRXint in AArch32 state.

All other instructions operate with the privilege determined by the current Exception level and security state. This applies to all Special-purpose and System registers accesses, memory accesses, and UNDEFINED instructions, and includes generating exceptions when the System registers trap or disable an instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H2.4.6 Debug state instructions, DCPS, DRPS, MRS, MSR

ARMv8 defines instructions to change between Exception levels in Debug state. These instructions can also change the mode at the current Exception level.

#### DCPS

DCPS allows the debugger to move the PE to a higher Exception level or to a specific mode at the current Exception level.

If the DCPS instruction is executed in AArch32 state and the target Exception level is using AArch64:

- The current instruction set switches from T32 to A64.
- The effect on registers that are not visible or only partially visible in AArch32 state is the same as for system calls in Non-debug state. See *Execution state* on page D1-1517.

Otherwise, the instruction set state does not change.

If the target Exception level is the same as the current Exception level, then the PE does not change Exception level. However, the PE can change mode.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state:

- In AArch64, the current endianness is set according to SCTLR_ELx.EE for the target Exception level.
- In AArch32, the current endianness is set according to SCTLR.EE or HSCTLR.EE for the target Exception level.

The assembler syntax for the DCPS instructions is:

```
DCPS1    {#<uimm16>}
DCPS2    {#<uimm16>}
DCPS3    {#<uimm16>}
```

<uimm16> is only available in the A64 encoding and is ignored by hardware.

The decode can be found in the instruction descriptions for DCPS1, DCPS2, and DCPS3 for A64, and DCPS1, DCPS2, DCPS3 for T32.

DCPS is UNDEFINED in Non-debug state.

Table H2-6 on page H2-4848 shows the target of the instruction. In Table H2-6 on page H2-4848 the entries have the following meaning:

**EL1h/Svc** This means that the target is:
- EL1h if EL1 is using AArch64.
- EL1 and Supervisor mode if EL1 is using AArch32.

**EL2h/Hyp** This means that the target is:
- EL2h if EL2 is using AArch64.
- EL2 and Hyp mode if EL2 is using AArch32.

**EL3h/Monitor** This means that the target is:
- EL3h if EL3 is using AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- EL3 and Monitor mode if EL3 is using AArch32.

**Table H2-6 Target for DCPS instructions in Debug state**

| Instruction | Target when taken from Exception level: | | | | |
|---|---|---|---|---|---|
| | EL0 | EL1 | EL2 | EL3 (AArch64) | EL3 (AArch32) |
| DCPS1 | EL1h/Svc | EL1h/Svc | EL2h/Hyp | EL3h | Svc, clears NS to 0 |
| DCPS2 | EL2h/Hyp | EL2h/Hyp | EL2h/Hyp | EL3h | UNDEFINED |
| DCPS3 | EL3h/Monitor | EL3h/Monitor | EL3h/Monitor | EL3h | Monitor, clears NS to 0 |

**Note**

- In AArch32 Monitor mode, DCPS1 and DCPS3 clear SCR.NS to 0.
- In AArch64, at EL3, DCPS does not change SCR_EL3.NS.

However:

- DCPS1 is UNDEFINED at EL0 in Non-secure state if both:
  - EL2 is implemented.
  - HCR_EL2.TGE == 1.
- DCPS2 is UNDEFINED at all Exception levels if EL2 is not implemented.
- DCPS2 is UNDEFINED at the following Exception levels if EL2 is implemented:
  - At EL0 and EL1 in Secure state.
  - At EL3 if EL3 is using AArch32.
- DCPS3 is UNDEFINED at all Exception levels if either:
  - EDSCR.SDD == 1.
  - EL3 is not implemented.

`DCPS` is also defined in T32, see *DCPS1, DCPS2, DCPS3 on page F6-2670*. There is no A32 encoding.

On executing a `DCPS` instruction:

- If the target Exception level is using AArch64:
  - ELR_ELx of the target Exception level becomes UNKNOWN.
  - SPSR_ELx of the target Exception level becomes UNKNOWN.
  - ESR_ELx of the target Exception level becomes UNKNOWN.
  - DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- If the target Exception level is using AArch32 DLR and DSPSR become UNKNOWN and:
  - If the target Exception level is EL1 or EL3, the LR and SPSR of the target mode become UNKNOWN.
  - If the target Exception level is EL2, then ELR_hyp, SPSR_hyp, and HSR become UNKNOWN.

If the target Exception level is using AArch32, and the target Exception level is EL1 or EL3, the LR and SPSR of the target mode become UNKNOWN.

The pseudocode for `DCPSInstruction()` is as follows:

```
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
case target_el of
    when EL1
        if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
        elsif HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then UndefinedFault();
        else handle_el = EL1;

    when EL2
        if !HaveEL(EL2) then UndefinedFault();
        elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
        elsif IsSecure() then UndefinedFault();
        else handle_el = EL2;

    when EL3
        if EDSCR.SDD == '1' || !HaveEL(EL3) then UndefinedFault();
        handle_el = EL3;

if ELUsingAArch32(handle_el) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    assert UsingAArch32();                  // Cannot move from AArch64 to AArch32
    case handle_el of
        when EL1  AArch32.WriteMode(M32_Svc);
        when EL2  AArch32.WriteMode(M32_Hyp);
        when EL3  AArch32.WriteMode(M32_Monitor);
    if handle_el == EL2 then
        ELR_hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
    else
        LR = bits(32) UNKNOWN;
    SPSR[] = bits(32) UNKNOWN;
    PSTATE.E = SCTLR[].EE;
else                                        // Targeting AArch64
    if UsingAArch32() then AArch64.MaybeZeroRegisterUppers();
    ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(32) UNKNOWN;  ESR[] = bits(32) UNKNOWN;
    PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;

DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;
UpdateEDSCRFields();                        // Update EDSCR processor state flags.

return;
```

## DRPS

DRPS allows the debugger to move the PE to a lower Exception level or to another mode at the current Exception level by copying the current SPSR to PSTATE.

If DRPS is executed in AArch64 state and the target Exception level is using AArch32:

- The current instruction set switches from A64 to T32.
- The effect on registers that are not visible or only partially visible in AArch32 state is the same as for exception returns in Non-debug state. See *Execution state* on page D1-1517.

Otherwise the instruction set state does not change.

If the target Exception level is the same as the current Exception level, then the PE does not change Exception level. However, the PE can change mode.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state:

- If targeting an Exception level using AArch64, current endianness is set according to SCTLR_ELx.EE, or SCTLR_EL1.E0E for the target Exception level.
- If targeting an Exception level using AArch32, current endianness is set by SPSR.E as appropriate.

The assembler syntax for the DRPS instruction is:

```
DRPS
```

The decode can be found in the instruction description for DRPS.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the SPSR specifies an illegal exception return, then PSTATE.{M, nRW, EL, SP} are unchanged and PSTATE.IL is set to 1. For further information on illegal exception returns, see *Illegal return events from AArch64 state* on page D1-1553.

PSTATE.{N, Z, C, V, Q, GE, IT, T, SS, D, A, I, F} are ignored in Debug state. This means that the effect of DRPS on these fields is to set them to an UNKNOWN value that might be the value from the SPSR. For more information see *Process state (PSTATE) in Debug state* on page H2-4832.

All other PSTATE fields are copied from SPSR.

DRPS is UNDEFINED at EL0 and in Non-debug state. In Debug state, the T32 encoding for ERET is decoded as DRPS. There is no A32 encoding for DRPS.

**Note**

Unlike an exception return, DRPS has no architecturally-defined effect on the Event Register and exclusive monitors. DRPS might set the Event Register or clear the exclusive monitors, or both, but this is not a requirement and debuggers must not rely on any implementation specific behavior.

On executing a DRPS instruction:

- If the target Exception level is using AArch64:
  - DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- If the target Exception level is using AArch32:
  - DLR and DSPSR become UNKNOWN.

The pseudocode for DRPSInstruction() is as follows:

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    SetPSTATEFromPSR(SPSR[]);

    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
    else
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;

    UpdateEDSCRFields();                                // Update EDSCR processor state flags.

    return;
```

### MRS and MSR instructions to access DLR_EL0 and DSPSR_EL0

The other Debug state instructions are MRS and MSR (register) instructions to read or write DLR_EL0 and DSPSR_EL0, and the equivalent MRC and MCR operations in AArch32 state.

```
MRS <Xt>, DLR_EL0      ; Copy DLR_EL0 to <Xt>
MRS <Xt>, DSPSR_EL0    ; Copy DSPSR_EL0 to <Xt>
MSR DLR_EL0, <Xt>      ; Copy <Xt> to DLR_EL0
MSR DSPSR_EL0, <Xt>    ; Copy <Xt> to DSPSR_EL0
```

These instructions can be executed at any Exception level when in Debug state, including EL0. They are UNDEFINED in Non-debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H2.4.7 Exceptions in Debug state

The following sections describe how exceptions are handled in Debug state:

- *Generating exceptions when in Debug state*.
- *Taking exceptions when in Debug state*.
- *Reset in Debug state* on page H2-4854.

#### Generating exceptions when in Debug state

In Debug state:

- Instruction Abort exceptions cannot happen because instructions are not fetched from memory.
- Interrupts, including SError and virtual interrupts are ignored and remain pending:
  - The pending interrupt remains visible in ISR.
- Debug exceptions are ignored.
- SCR.EA is treated as if it were set to 0, regardless of its actual state, other than for the purpose of reading the bit.
- All instruction bit patterns that are an allocated instruction at the current Exception level, but listed in *Executing instructions in Debug state* on page H2-4832 as UNDEFINED in Debug state, generate Undefined Instruction exceptions, which are taken to the current Exception level, or to EL1 if executing at EL0. This includes `SVC`, `HVC`, `SMC`, `BRK`, and `HLT`. The priority and syndrome for these exceptions is the same as for executing an encoding that does not have an allocated instruction.
- Instructions executed at EL2, EL1 and EL0 that are configured by EL3 control registers to trap to EL3:
  - Generate the appropriate trap exception taken to EL3 if EDSCR.SDD == 0.
  - Generate an Undefined Instruction exception taken to the current Exception level, or to EL1 if executing at EL0, if EDSCR.SDD == 1. If the exception is taken to an Exception level using AArch64 or to AArch32 Hyp mode, this is reported with an exception class of `0x00`.

  Otherwise configurable traps, enables, and disables, for instructions are unaffected by Debug state, and executing the affected instructions generates the appropriate exceptions.

Otherwise, synchronous exceptions, including Data Aborts, are generated as they would be in Non-debug state and taken to the appropriate Exception level in Debug state.

**Note**

If EDSCR.SDD == 1 then an exception from Non-secure state is never taken to Secure state. See *Security in Debug state* on page H2-4845.

#### Taking exceptions when in Debug state

When the PE is in Debug state, all exceptions are synchronous. When an exception is generated, it is taken to Debug state. This means that:

- The target Exception level is as defined for the exception in Non-debug state.
- If the target Exception level is using AArch32 then the target PE mode is as defined for the exception in Non-debug state.
- The exception is reported as defined for the exception in Non-debug state, using the syndrome register or registers for the target Exception level. In AArch64, these are ESR_ELx, and FAR_ELx. In AArch32, these are DFSR, DFAR, HSR, HDFAR, and HPFAR. For example:
  - If a Data Abort exception is taken to Abort mode at EL1 or EL3 and the exception is taken from AArch32 state and using the Short-descriptor translation table format, the DFSR reports the exception using the Short-descriptor format fault encoding. For exceptions other than Data Abort exceptions taken to Abort mode, DFSR is not updated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— If an instruction is trapped to an Exception level using AArch64 due to a configurable trap, disable, or enable, the exception code reported is the same as it would be in Non-debug state.

The effect on auxiliary syndrome registers, such as AFSR, is IMPLEMENTATION DEFINED.

- The PE remains in Debug state and changes to the target mode.
- If EL3 is using AArch32 and the exception is taken from Monitor mode, SCR.NS is cleared to 0.
- If the exception is taken to an Exception level using AArch32, the PE continues to execute T32 instructions, regardless of the TE bit in the System register for the target Exception level.
- The endianness switches to that indicated by the EE bit of the System register for the target Exception level.
- The SPSR for the target Exception level or mode is corrupted and becomes UNKNOWN.
- If the target Exception level is using AArch64, ELR_ELx for the target Exception level becomes UNKNOWN.
- If the target Exception level is EL2 using AArch32, ELR_hyp becomes UNKNOWN.
- If the target Exception level is EL1 or EL3 using AArch32, LR_<mode> for the target mode becomes UNKNOWN.
- DLR and DSPSR become UNKNOWN.
- The cumulative error flag, EDSCR.ERR, is set to 1. See *Cumulative error flag* on page H4-4895.
- PSTATE.IL is cleared to 0.
- PSTATE.{IT, T, SS, D, A, I, F} are set to UNKNOWN values, and PSTATE.{N, Z, C, V, Q, GE} are unchanged. However, these fields are ignored and are not observable in Debug state. For more information see *Process state (PSTATE) in Debug state* on page H2-4832.

The debugger must save any state that can be corrupted by an exception before executing an instruction that might generate another exception.

### Pseudocode description of taking exceptions in Debug state

The pseudocode function `TakeException()` in *Pseudocode description of exception entry to AArch64 state* on page D1-1539 shows the behavior when the PE takes an exception to an Exception level using AArch64 in Non-debug state. In Debug state, this is replaced with the function `TakeExceptionInDebugState()`.

```
// AArch64.TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception Level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;  PSTATE.nRW = '0';  PSTATE.SP = '1';

    SPSR[] = bits(32) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then                                 // Coming from AArch32
        PSTATE.IT = '00000000';  PSTATE.T = '0';    // PSTATE.J is RES0
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.
    EndOfInstruction();
```

The pseudocode functions EnterMode(), EnterHypMode(), and EnterMonitorMode() in *Additional pseudocode functions for exception handling* on page G1-3829 show the behavior when the PE takes an exception to an Exception level using AArch32 in Non-debug state. In Debug state, EnterMode() is replaced with the function EnterModeInDebugState().

```
// AArch32.EnterModeInDebugState()
// ===============================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.
    EndOfInstruction();
```

In Debug state, EnterHypMode() is replaced with the function EnterHypModeInDebugState().

```
// AArch32.EnterHypModeInDebugState()
// ==================================
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    EDSCR.ERR = '1';
    UpdateEDSCRFields();
    EndOfInstruction();
```

In Debug state, EnterMonitorMode() is replaced with EnterMonitorModeInDebugState().

```
// AArch32.EnterMonitorModeInDebugState()
// ======================================
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
if PSTATE.M == M32_Monitor then SCR.NS = '0';
AArch32.WriteMode(M32_Monitor);
SPSR[] = bits(32) UNKNOWN;
R[14] = bits(32) UNKNOWN;
// In Debug state, the PE always execute T32 instructions when in AArch32 state, and
// PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
PSTATE.T = '1';                              // PSTATE.J is RES0
PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
PSTATE.E = SCTLR.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
EDSCR.ERR = '1';
UpdateEDSCRFields();                         // Update EDSCR processor state flags.
EndOfInstruction();
```

### Reset in Debug state

If the PE is reset when in Debug state, it exits Debug state and enters Non-debug reset state. When the PE is in reset state, EDSCR.STATUS == `0b000010` and writes to EDITR are ignored.

**Note**

If EDECR.RCE == 1, meaning that a Reset Catch debug event is programmed, and if halting is allowed on exiting reset state, then on exiting reset state the PE halts and re-enters Debug state. See *Reset Catch debug events* on page H3-4880. All PE registers have taken their reset values, which might be UNKNOWN.

## H2.4.8 Accessing registers in Debug state

Register accesses are unchanged in Debug state. The view of each register is determined by either the current Exception level or the mode, or both, and accesses might be disabled or trapped by controls at a higher Exception level.

### General-purpose register access, other than SP access in AArch64 state

A single general-purpose register can be read by issuing an `MSR` instruction through the ITR to write DBGDTR_EL0 in AArch64 state, or an `MCR` instruction through the ITR to write DBGDTRTXint in AArch32 state. The debugger can then read the DTR register or registers through the external debug interface. The reverse sequence writes to a general-purpose register.

Figure H2-1 on page H2-4855 shows the reading and writing of general-purpose registers, other than SP, in Debug state in AArch64 state.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

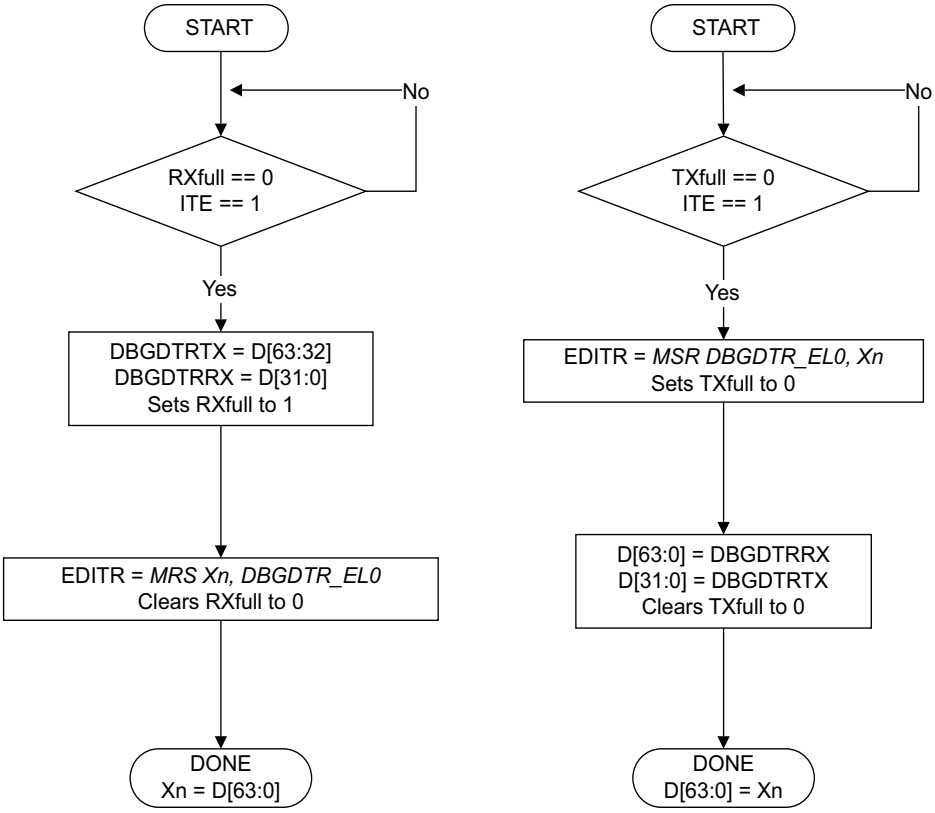



**Figure H2-1 Reading and writing general-purpose registers, other than SP, in Debug state in AArch64 state**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

Figure H2-2 shows the reading and writing of general-purpose registers in Debug state in AArch32 state.

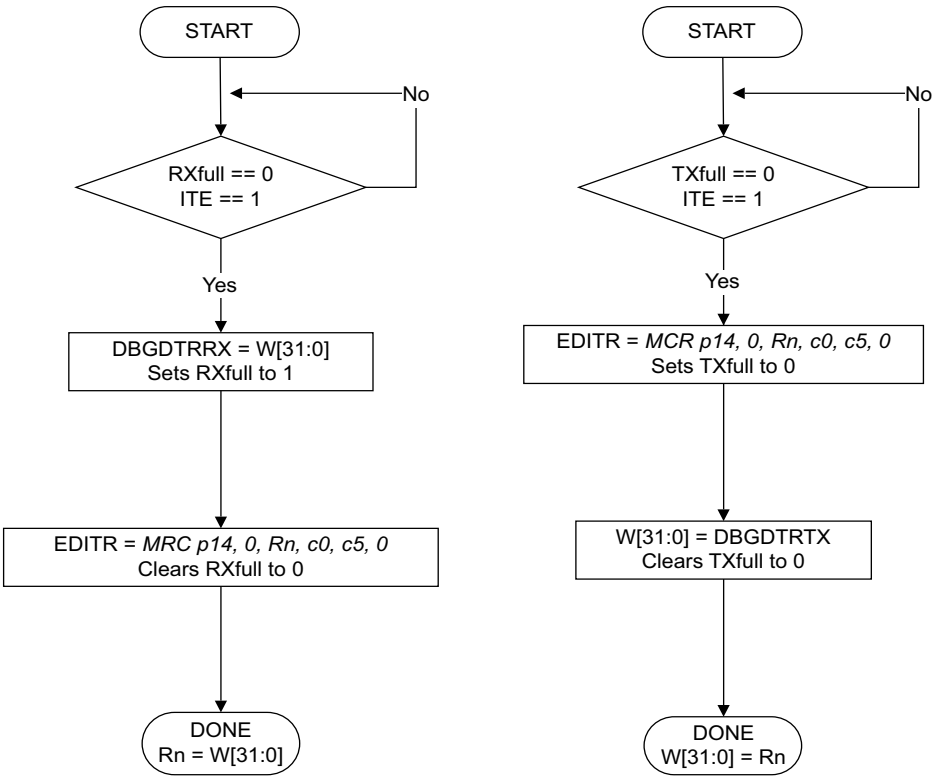



**Figure H2-2 Reading and writing general-purpose registers in Debug state in AArch32 state**

### SIMD and floating-point register, System register, and SP access in AArch64 state

To read a SIMD and floating-point register or a System register, the debugger must first copy the value into a general-purpose register using:

- An `FMOV` instruction in AArch64 or a `VMOV` instruction in AArch32 for floating-point transfers to SIMD and FP registers.
- A `UMOV` instruction in AArch64 or a `VMOV` instruction in AArch32 for SIMD transfers to SIMD and FP registers.
- An `MRS` instruction in AArch64 or an `MRC` instruction in AArch32 for System registers.
- A `MOV Xd,SP` instruction for the SP register in AArch64 state.

The debugger can then read out the particular general-purpose register. The reverse sequence writes a register.

### PC and PSTATE access

The debugger reads the program counter and PSTATE of the process being debugged through the DLR_EL0 and DSPSR_EL0 System registers. The actual values of PC and PSTATE cannot be directly observed in Debug state:

- Instructions that are used for direct reads and writes of PC and PSTATE in Non-debug state are UNDEFINED in Debug state.
- On taking an exception, ELR_ELx and SPSR_ELx at the target exception level are UNKNOWN. They do not record the PC and PSTATE.

PSTATE.{IL, E, M, nRW, EL, SP} are indirectly read by instructions executed in Debug state, but all other PSTATE fields are ignored and cannot be observed. See also:

- *Process state (PSTATE) in Debug state* on page H2-4832.
- *Executing instructions in Debug state* on page H2-4832.
- *Exceptions in Debug state* on page H2-4851.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H2.4.9 Accessing memory in Debug state

How the PE accesses memory is unchanged in Debug state. This includes:

- The operation of the MMU, including address translation, tagged address handling, access permissions, memory attribute determination, and the operation of any TLBs.
- The operation of any caches and coherency mechanisms.
- Alignment support.
- Endianness support.
- The Memory order model.

#### Simple memory transfers

Simple memory accesses can be performed in Debug state by issuing memory access instructions through the ITR and passing data through the DTR registers. *Executing instructions in Debug state* on page H2-4832 lists the memory access instructions that are supported in Debug state.

#### Bulk memory transfers

Memory access mode can accelerate bulk memory transfers in Debug state. See *DCC and ITR access modes* on page H4-4889.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H2.5 Exiting Debug state

The PE exits Debug state when it receives a Restart request trigger event. If EDSCR.ITE == 0 the behavior of any instruction issued through the ITR in Normal access mode or an operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state after the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

**Note**

- Implementations can set EDSCR.ITE to 1 to indicate that further instructions can be accepted by ITR before the previous instructions have completed. If any previous instruction has not completed and EDSCR.ITE == 1, then the PE must complete these instructions in Debug state before executing the restart sequence. EDSCR.ITE == 0 indicates that the PE is not ready to restart.
- A debugger must observe that any instructions issued through EDITR that might generate a synchronous exception, as complete, before issuing a restart request. It can do this by observing the completion of a later instruction, as synchronous exceptions must occur in program order. For example, a debugger can observe that an instruction that reads or writes a DTR register is complete because of its effect on the EDSCR.{TXfull, RXfull} flags.

On exiting Debug state, the PE sets the program counter to the address in DLR, where:

- If exiting to AArch32 state:
  - Bits[63:32] of DLR are ignored.
  - Bits[31:1] of the PC are set to the value of bits[31:1] of DLR.
  - Bit[0] of the PC is set to a CONSTRAINED UNPREDICTABLE choice of 0 or the value of bit[0] in DLR.
- If exiting to AArch64 state:
  - Bits[63:56] of DLR_EL0[31:0] might be ignored as part of tagged address handling. See *Address tagging in AArch64 state* on page D4-1742.
  - Otherwise the PC is set from DLR_EL0[31:0].

Exit from Debug state can give rise to a misaligned PC exception when the program counter is used. Unlike an exception return, this might also happen when returning to AArch32 state. For more information, see *PC alignment checking* on page D1-1531.

On exiting Debug state, PSTATE is set from DSPSR in the same way that an exception return sets PSTATE from SPSR_ELx:

- The same illegal exception return checks that apply to an exception return also apply to exiting Debug state. If the return from Debug state is an illegal exception return then the effect on PSTATE and the PC is the same as for any other illegal exception return. See *Exception return* on page D1-1552 and *Exception return to an Exception level using AArch32* on page G1-3796.
- The checks on the PSTATE.IT bits that apply to exiting Debug state into AArch32 state are the same as those that apply to an exception return. See Appendix K1 *Architectural Constraints on UNPREDICTABLE behaviors*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- PSTATE.SS is copied from DSPSR.SS if all of the following hold:
  - MDSCR_EL1.SS == 1.
  - The debug target Exception level is using AArch64.
  - Software step exceptions from the restart Exception level are enabled.

  Otherwise PSTATE.SS is set to 0.

  **Note**

  Unlike a return using ERET, PSTATE.SS must be restored from DSPSR.SS because otherwise it is UNKNOWN.

  However, if OSDLR.DLK == 1 and DBGPRCR.CORENPDRQ == 0, meaning the OS Double Lock is locked in Non-debug state and therefore Software Step exceptions are disabled, but otherwise Software Step exceptions would be enabled from the restart Exception level, it is CONSTRAINED UNPREDICTABLE whether PSTATE.SS is copied from DSPSR.SS.

**Note**

- One important difference between Debug state exit and an exception return is that the PE can exit Debug state at EL0. Despite this, the behavior of an exit from Debug state is similar to an exception return. For example, PSTATE.{D, A, I, F} is updated regardless of the value of SCTLR_EL1.UMA.
- Exit from Debug state has no architecturally-defined effect on the Event Register and exclusive monitors. An exit from Debug state might set the Event Register or clear the exclusive monitors, or both, but this is not a requirement and debuggers must not rely on any implementation specific behavior.

The pseudocode for ExitDebugState() is as follows.

```
// ExitDebugState()
// ================

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the processor is restarting, debuggers must use EDPRSR.SDR
    // to detect that the processor has restarted.
    EDSCR.STATUS = '000001';                            // Signal restarting
    EDESR<2:0> = '000';                                 // Clear any pending Halting debug events

    new_pc = DLR_EL0;
    spsr = DSPSR;

    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr);                             // Can update privileged bits, even at EL0.

    if UsingAArch32() then
        if ConstrainUnpredictableBool() then new_pc<0> = '0';
        BranchTo(new_pc<31:0>, BranchType_UNKNOWN);     // AArch32 branch
    else
        // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
        if spsr<4> == '1' && ConstrainUnpredictableBool() then
            new_pc<63:32> = Zeros();
        BranchTo(new_pc, BranchType_DBGEXIT);           // A type of branch that is never predicted

    (EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');         // Atomically signal restarted
    UpdateEDSCRFields();                                // Stop signalling processor state.
    DisableITRAndResumeInstructionPrefetch();

    return;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Chapter H3
# Halting Debug Events

This chapter describes a particular class of debug events. It contains the following sections:


- *Introduction to Halting debug events* on page H3-4862.
- *Halting Step debug events* on page H3-4864.
- *Halt Instruction debug event* on page H3-4874.
- *Exception Catch debug event* on page H3-4875.
- *External Debug Request debug event* on page H3-4878.
- *OS Unlock Catch debug event* on page H3-4879.
- *Reset Catch debug events* on page H3-4880.
- *Software Access debug event* on page H3-4881.
- *Synchronization and Halting debug events* on page H3-4882.


**Note**

Table K13-1 on page K13-5708 disambiguates the general register references used in this chapter.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H3.1 Introduction to Halting debug events

External debug defines Halting debug events. The following Halting debug events are available in ARMv8:

- *Halting Step debug events* on page H3-4864.
- *Halt Instruction debug event* on page H3-4874.
- *Exception Catch debug event* on page H3-4875.
- *External Debug Request debug event* on page H3-4878.
- *OS Unlock Catch debug event* on page H3-4879.
- *Reset Catch debug events* on page H3-4880.
- *Software Access debug event* on page H3-4881.

If halting is allowed, a Halting debug event halts the PE. The PE enters Debug state.

In addition, breakpoints and watchpoints might halt the PE if halting is allowed. See *Breakpoint and Watchpoint debug events* on page H2-4822. Because breakpoints and watchpoints can generate an exception or halt the PE, Breakpoint and Watchpoint debug events are not classified as Halting debug events.

For a definition of Debug state, see Chapter H2 *Debug State*. For a definition of halting allowed, see *Halting allowed and halting prohibited* on page H2-4821.

*Debug state entry and debug event prioritization* on page H2-4823 describes the behavior when multiple debug events are generated by an instruction.

See also *Synchronization and Halting debug events* on page H3-4882.

Table H3-1 shows the behavior of Breakpoint, Watchpoint, and Halting debug events.

**Table H3-1 Summary of debug events and possible outcomes**

| Debug event type | PE behavior when halting is: | |
|---|---|---|
| | Allowed | Prohibited |
| *Breakpoint and Watchpoint debug events* on page H2-4822 | Halt | See Table D2-1 on page D2-1639 and Table G2-1 on page G2-3885 |
| *Halt Instruction debug event* on page H3-4874 | Halt | UNDEFINED |
| *Software Access debug event* on page H3-4881 | Halt | Ignored |
| *Exception Catch debug event* on page H3-4875 | Halt | Ignored |
| *Halting Step debug events* on page H3-4864 | Halt | Pended |
| *External Debug Request debug event* on page H3-4878 | Halt | Pended |
| *Reset Catch debug events* on page H3-4880 | Halt | Pended |
| *OS Unlock Catch debug event* on page H3-4879 | Pended | Pended |

Table H3-2 shows where the pseudocode for each Halting debug event type is located.

**Table H3-2 Pseudocode description of Halting debug events**

| Halting debug event type | Pseudocode |
|---|---|
| *Halt Instruction debug event* on page H3-4874 | *HLT* on page C6-531 for AArch64 and *HLT* on page F6-2685 for AArch32 |
| *Software Access debug event* on page H3-4881 | *Pseudocode description of Software Access debug event* on page H3-4881 |
| *Exception Catch debug event* on page H3-4875 | *Pseudocode description of Exception Catch debug events* on page H3-4877 |
| *Halting Step debug events* on page H3-4864 | *Pseudocode description of Halting Step debug events* on page H3-4873 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table H3-2 Pseudocode description of Halting debug events (continued)**

| Halting debug event type | Pseudocode |
|---|---|
| *External Debug Request debug event* on page H3-4878 | *Pseudocode description of External Debug Request debug events* on page H3-4878 |
| *Reset Catch debug events* on page H3-4880 | *Pseudocode description of Reset Catch debug event* on page H3-4880 |
| *OS Unlock Catch debug event* on page H3-4879 | *Pseudocode description of OS Unlock Catch debug event* on page H3-4879 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H3.2 Halting Step debug events

Halting Step is a debug resource that a debugger can use to make the PE step through code one instruction at a time. This section describes the Halting Step debug events. It is divided into the following sections:

- *Overview of a Halting Step debug event*.
- *The Halting Step state machine*.
- *Using Halting Step* on page H3-4867.
- *Detailed Halting Step state machine behavior* on page H3-4867.
- *Synchronization and the Halting Step state machine* on page H3-4870.
- *Stepping T32 IT instructions* on page H3-4871.
- *Disabling interrupts while stepping* on page H3-4872.
- *Syndrome information on Halting Step* on page H3-4872.
- *Pseudocode description of Halting Step debug events* on page H3-4873.

The architecture describes the behavior as a simple Halting Step state machine. See *The Halting Step state machine*.

### H3.2.1 Overview of a Halting Step debug event

The behavior of Halting Step is defined by a state machine, shown in Figure H3-1 on page H3-4866. A Halting Step debug event executes a single instruction and then returns control to the debugger. When the debugger software wants to execute a Halting Step:

1. With the PE in Debug state, the debugger activates Halting Step.
2. The debugger signals the PE to exit Debug state and return to the instruction that is to be stepped.
3. The PE executes that single instruction.
4. The PE enters Debug state before executing the next instruction.

However, an exception might be generated while the instruction is being stepped. That is either:

- A synchronous exception generated by the instruction being stepped.
- An asynchronous exception taken before or after the instruction being stepped.

Halting Step has its own enable control bit, EDECR.SS and EDESR.SS.

**Note**

Because the Halting Step state machine states occur as a result of normal PE operation, the states can be described as both:

- PE states.
- Halting Step states.

### H3.2.2 The Halting Step state machine

The state machine states are:

**Inactive** Halting Step is inactive. No Halting Step debug events can be generated, therefore execution is not affected by Halting Step. The PE is in this state whenever either of the following is true:

- Halting Step is disabled. That is, EDECR.SS is set to 0 and EDESR.SS is set to 0.
- Halting is prohibited. See *Halting the PE on debug events* on page H2-4821.

In Figure H3-1 on page H3-4866 this state is shown in red.

**Active-not-pending**

Halting Step is enabled and active. This is the state in which the PE steps an instruction. EDECR.SS == 1 and EDESR.SS == 0. A debugger must only set EDECR.SS to 1 when the PE is in Debug state.

In Figure H3-1 on page H3-4866 this state is shown in green.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Active-pending**

Halting Step is enabled and active. The step has completed, and the PE enters Debug state. EDESR.SS == 1.

In Figure H3-1 on page H3-4866 this state is shown in green.

Whenever Halting Step is enabled and active, whether the state machine is in the active-not-pending state or in the active-pending state depends on EDESR.SS. *Halting Step state machine states* on page H3-4867 shows this.

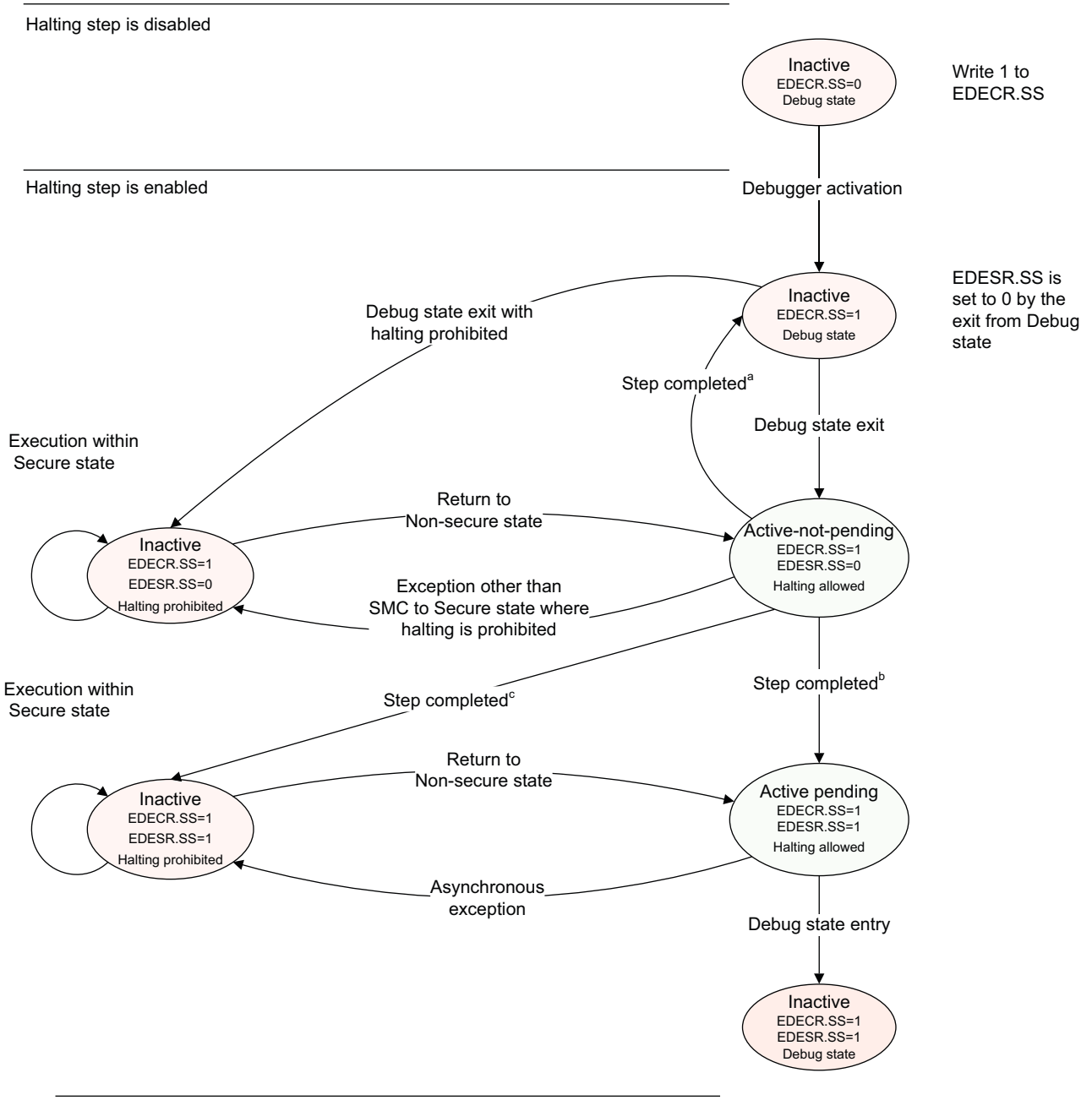
In the simple sequential execution of the program the PE executes the Halting Step state machine, as follows:

1. Initially, Halting Step is inactive.
2. After exiting Debug state, Halting Step is active-not-pending.
3. The PE executes an instruction and Halting Step is active-pending.
4. The pending Debug state entry is taken on the next instruction and the step is complete.

Exceptions and other changes to the PE context can interrupt this sequence.

Figure H3-1 on page H3-4866 shows a Halting Step state machine.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




a. Step completed occurs when:
   - A debug event, other than a Halting Step debug event, causes entry into Debug state.

b. Step completed occurs when:
   - An instruction is executed without taking an exception.
   - An exception is taken to a state where halting is allowed.
   - A reset.

c. Step completed occurs when:
   - An SMC exception is taken to Secure state where halting is prohibited.

**Figure H3-1 Halting Step state machine**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Note**

Figure H3-1 on page H3-4866 only describes state transitions to and from the inactive state by exit from Debug state, executing an exception return, or taking an exception. Other changes to the PE context, including writes to registers such as EDECR and OSDLR and changes to the authentication interface can also cause changes to the Halting Step state machine. These can lead to UNPREDICTABLE behavior. See *Synchronization and the Halting Step state machine* on page H3-4870.

The following bits control the state machine, as shown in Table H3-3:

- EDECR.SS. This is the Halting Step enable bit.

  **Note**

  — The EDECR value is preserved over powerdown, meaning that the step active state is maintained over a powerdown event.

  — A debugger must only set EDECR.SS to 1 when the PE is in Debug state.

- EDESR.SS.

Table H3-3 shows the Halting Step state machine states. The letter X in a register column means that the relevant bit can be set to either zero or one.

**Table H3-3 Halting Step state machine states**

| Halting | EDECR.SS | EDESR.SS | Halting Step state |
|---|---|---|---|
| Prohibited | X | X | Inactive |
| Allowed | 0 | 0 | Inactive |
| Allowed | 1 | 0 | Active-not-pending |
| Allowed | X | 1 | Active-pending |

### H3.2.3 Using Halting Step

To step a single instruction the PE must be in Debug state:

1. The debugger sets EDECR.SS to 1 to enable Halting step.
2. The debugger signals the PE to exit Debug state with DLR set to the address of the instruction being stepped. The PE clears EDESR.SS to 0 and the Halting Step state machine enter the active-not-pending state.
3. The PE executes the instruction being stepped.

   If an exception is taken to a state where halting is prohibited, then EDESR.SS is always correct for the preferred return address of the exception.
4. The PE enters Debug state before executing the next instruction and the step is complete.

### H3.2.4 Detailed Halting Step state machine behavior

The behavior of the Halting Step state machine is described in the following sections:

- *Entering the active-not-pending state* on page H3-4868.
- *PE behavior in the active-not-pending state* on page H3-4868.
- *Entering the active-pending state* on page H3-4869.
- *PE behavior in the inactive state when in Non-debug state* on page H3-4870.
- *PE behavior in Debug state* on page H3-4870.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Entering the active-not-pending state

The PE enters the active-not-pending state:

- By exiting Debug state with EDECR.SS == 1.
- By an exception return from a state where halting is prohibited to a state where halting is allowed with EDECR.SS == 1 and EDESR.SS == 0.
- As described in *Synchronization and the Halting Step state machine* on page H3-4870.

### PE behavior in the active-not-pending state

When the PE is in the active-not-pending state it does one of the following:

- It executes one instruction and does one of the following:
  - Completes it without generating a synchronous exception.
  - Generates a synchronous exception.
  - Generates a debug event that causes entry to Debug state.
- It takes an asynchronous exception without executing any instruction.
- It takes an asynchronous debug event into Debug state.

#### *If no exception or debug event is generated*

If no exception or debug event is generated the PE sets EDESR.SS to 1. This means that the Halting Step state machine advances to the active-pending state.

#### *If an exception or debug event is generated*

The PE sets EDESR.SS according to all of the following:

- The type of exception.
- The target Exception level of the exception.
- If the exception is taken to Secure state, whether halting is prohibited in Secure state.
  - This is determined by the result of `ExternalSecureInvasiveDebugEnabled()`.

If an exception or debug event is generated, the PE sets EDESR.SS to 1 if one of the following applies:

- A synchronous exception is generated by the instruction and one of the following applies:
  - The exception is taken to EL1 or EL2.
  - The exception is not an SMC exception and `ExternalSecureInvasiveDebugEnabled()` == TRUE.
  - The exception is an SMC exception.
- An asynchronous exception is generated before executing an instruction and this is either:
  - Taken to EL1 or EL2.
  - Taken to EL3 and `ExternalSecureInvasiveDebugEnabled()` == TRUE.
- A PE reset occurs.

Otherwise EDESR.SS is unchanged. This happens when:

- No instruction is executed because either:
  - An asynchronous exception is taken to EL3 and `ExternalSecureInvasiveDebugEnabled()` == FALSE.
  - An asynchronous debug event causes entry to Debug state.
- An instruction is executed and either:
  - Generates a synchronous exception other than an SMC exception which is taken to EL3, and `ExternalSecureInvasiveDebugEnabled()` == FALSE.
  - Generates a synchronous debug event and causes entry to Debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If halting is prohibited after taking the exception or debug event, then the Halting Step state machine advances to the inactive state. Otherwise the Halting Step state machine advances to the active-pending state.

**Note**

The underlying criteria for the value of EDESR.SS on an exception are:

- Whether halting is allowed at the target of the exception. If halting is allowed, the PE must step into the exception. If halting is prohibited, the PE must step over the exception.
- Whether the preferred return address of the exception is the instruction itself or the next instruction, if the PE steps over the exception.

Table H3-4 shows the behavior of the active-not-pending state. The letter X indicates that `ExternalSecureInvasiveDebugEnabled()` can be either TRUE or FALSE.

**Table H3-4 Summary of active-not-pending state behavior**

| Event | Target Exception level | `ExternalSecureInvasiveDebugEnabled()` | Value written to EDESR.SS |
|---|---|---|---|
| No exception or debug event | Not applicable | X | 1 |
| SMC exception | EL3 | X | 1 |
| Reset | Highest | X | 1 |
| Exception, other than SMC exception | EL1 | X | 1 |
| | EL2 | X | 1 |
| | EL3 | TRUE | 1 |
| | | FALSE | Unchanged |
| Debug event | Debug state | X | Unchanged |

### Entering the active-pending state

The PE enters the active-pending state by one of the following:

- From the active-not-pending state by:
  - Executing an instruction without taking an exception.
  - Taking an exception so that the PE remains in a state where halting is allowed.
- An exception return from a state where halting is prohibited when EDESR.SS == 1.

  **Note**

  That is, an exception return from Secure state with `ExternalSecureInvasiveDebugEnabled()` == FALSE to Non-secure state with `ExternalInvasiveDebugEnabled()` == TRUE.
- A reset when the value of EDECR.SS == 1, regardless of the state the PE was in before the reset occurred.
- Following the description in *Synchronization and the Halting Step state machine* on page H3-4870.

When the PE is in the active-pending state, it enters Debug state before executing an instruction. However, if `ExternalSecureInvasiveDebugEnabled()` == FALSE, the architecture does not define the prioritization of this Debug state entry with respect to any pending asynchronous exception that is taken from Non-secure state to EL3.

If an exception is prioritized over the halt, then EDESR.SS is unchanged. On return from the exception the Halting Step state machine re-enters the active-pending state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The entry into Debug state has higher priority than all other types of exception, including all other asynchronous exceptions.

**Note**

This means that it is possible to step a reentrant exception in the exception vector table.

### PE behavior in the inactive state when in Non-debug state

EDESR.SS is not updated by the execution of an instruction or the taking of an exception when Halting Step is inactive. This means that EDESR.SS is not changed by an exception handled in a state where halting is prohibited.

On return to a state where halting is allowed, the Halting Step state machine is restored either to the active-pending state or the active-not-pending state, depending on the value of EDESR.SS. The return to a state where halting is allowed is normally by an exception return, which is a *Context synchronization operation*.

See also *Synchronization and the Halting Step state machine*.

### PE behavior in Debug state

Halting Step is inactive in Debug state because halting is prohibited, see *Halting allowed and halting prohibited* on page H2-4821.

Entry to Debug state does not change EDESR.SS.

EDESR.SS is cleared to 0 on exiting Debug state as the result of a restart request. If EDECR.SS == 1, Halting Step enters the active-not-pending state.

**Note**

This means that EDESR.SS is never cleared to 0 by the execution of an instruction in Debug state, or by taking an exception when in Debug state as described in *PE behavior in the active-not-pending state* on page H3-4868, because the Halting Step state machine is not in the active-not-pending state. EDESR.SS can be cleared by a write to EDESR, see the register description.

However, if the PE exits Debug state as the result of a PE reset and EDECR.SS == 1, then Halting Step immediately enters the active-pending state, as EDESR.SS is set to the value of EDECR.SS.

## H3.2.5 Synchronization and the Halting Step state machine

The Halting Step state machine also changes state if:

- Halting becomes allowed or prohibited other than by exit from Debug state, an exception return, or taking an exception. This means that halting becomes allowed or prohibited because:
  - The security state changes without an exception return. See *State and mode changes without explicit context synchronization operations* on page G2-3946.
  - The external authentication interface changes.
  - The OS Double Lock status, `DoubleLockStatus()`, changes.
- A write to EDECR when the PE is in Non-debug state changes the value of EDECR.SS.
- A write to EDESR when the PE is in Non-debug state clears EDESR.SS to 0.

These operations are guaranteed to take effect only after a *Context synchronization operation*.

The PE must perform the required behavior of the new state before or immediately following the next *Context synchronization operation*, but it is not required to do so immediately. This means that the PE can perform the required behavior of the old state before the next *Context synchronization operation*. This is illustrated in Example H3-1 on page H3-4871 and Example H3-2 on page H3-4871.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Example H3-1 Synchronization requirements 1**

EDECR.SS is set to 1 in Debug state, requesting the active-not-pending state on exit from Debug state. On exit from Debug state the PE immediately takes an exception to Secure state. `ExternalSecureInvasiveDebugEnabled()` == FALSE, meaning that halting is prohibited in Secure state. The PE does not step any instructions but executes the software in Secure state as normal. EDESR.SS remains set to 0. If `ExternalSecureInvasiveDebugEnabled()` subsequently becomes TRUE, meaning that halting is now allowed, the PE must perform the required behavior of the active-not-pending state before or immediately following the next *Context synchronization operation*, but it is not required to do so immediately.

**Example H3-2 Synchronization requirements 2**

EDECR.SS is set to 1 in Debug state. On exit from Debug the PE executes an `MSR` instruction that sets OSDLR_EL1.DLK to 1 and `DoubleLockStatus()` becomes TRUE. This change requires a *Context synchronization operation* to guarantee its effect, meaning it is CONSTRAINED UNPREDICTABLE whether:

- Halting is allowed:
  - The PE enters Debug state on the next instruction.
- Halting is prohibited:
  - The PE does not enter Debug state.

The value in EDESR.SS depends on whether halting was allowed or prohibited when the write to OSDLR_EL1.DLK completed, and so it might be 0 or 1. If a second `MSR` instruction clears OSDLR_EL1.DLK to 0, the PE must perform the required behavior of the state indicated by EDESR.SS before or immediately following the next *Context synchronization operation*, but it is not required to do so immediately.

See also *Synchronization and Halting debug events* on page H3-4882.

### H3.2.6 Stepping T32 `IT` instructions

In an implementation that supports the ITD control, the architecture permits a combination of one T32 `IT` instruction and another 16-bit T32 instruction to be treated as a single 32-bit instruction when the value of the ITD field that applies to the current Exception level is 1.

For the purpose of stepping an item, it is IMPLEMENTATION DEFINED whether:

- The PE considers such a pair of instructions to be one instruction.
- The PE considers such a pair of instructions be two instructions.

It is IMPLEMENTATION DEFINED whether this behavior depends on the value of the applicable ITD bit. For example:

- The debug logic might consider such a pair of instructions as one instruction, regardless of the state of the applicable ITD field.
- The debug logic might consider such a pair of instructions as two instructions, regardless of the state of the applicable ITD field.
- The debug logic might consider such a pair of instructions as one instruction when the value of the applicable ITD field is 1, and as two instructions when the value of the ITD field is 0.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

The ITD control fields are:

**HSCTLR.ITD** Applies to execution at EL2 when EL2 is using AArch32.

**SCTLR.ITD** Applies to execution at EL0 or EL1 when EL1 is using AArch32.

**SCTLR_EL1.ITD**
Applies to execution at EL0 using AArch32 when EL1 is using AArch64.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H3.2.7 Disabling interrupts while stepping

When using Halting Step, the sequence of entering Debug state, interacting with the debugger, and then exiting Debug state for each instruction reduces the rate at which the PE executes instructions. However, the rate at which certain interrupts, such as timer interrupts, are generated might be fixed by the system. This means it might be necessary to disable interrupts while using Halting Step by setting EDSCR.INTdis, to allow the code being debugged to make forward progress.

### H3.2.8 Syndrome information on Halting Step

Three EDSCR.STATUS encodings record different scenarios for entering Debug state on a Halting Step debug event:

**Halting Step, normal**

An instruction other than a Load-Exclusive instruction was stepped.

**Halting Step, exclusive**

A Load-Exclusive instruction was stepped.

**Halting Step, no syndrome**

The syndrome data is not available.

If the PE enters Debug state due to a Halting Step debug event immediately after stepping an instruction in the active-not-pending state, EDSCR.STATUS is set to either:

- Halting Step, normal, if the stepped instruction was not a Load-Exclusive instruction.
- Halting Step, exclusive, if the stepped instruction was a Load-Exclusive instruction.

If the stepped instruction was a conditional Load-Exclusive instruction that failed its condition code test, EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, normal, or Halting Step, exclusive.

Otherwise the PE enters Debug state without stepping an instruction. This means that the Halting Step state machine enters the active-pending state directly from the inactive state, without going through active-not-pending state. In this case, EDSCR.STATUS is set to Halting Step, no syndrome. This happens when:

- The PE enters directly into the active-pending state on an exception return to Non-secure state from EL3 when Halting is prohibited in Secure state.
- The active-pending state is entered for other reasons. See *Synchronization and the Halting Step state machine on page H3-4870*

In addition, EDSCR.STATUS is set to one of a CONSTRAINED UNPREDICTABLE choice if:

- The instruction being stepped generated a synchronous exception, or a pending asynchronous exception was taken before the instruction was executed.

  In this case EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of:

  — Halting Step, no syndrome, or Halting Step, normal, if the stepped instruction was not a Load-Exclusive instruction.

  — Halting Step, no syndrome, or Halting Step, exclusive, if the stepped instruction was a Load-Exclusive instruction.

- The instruction that was stepped was an exception return instruction or an `ISB`. As these instructions are not in the Load-Exclusive instructions, EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, no syndrome or Halting Step, normal.
- The PE enters directly into the active-pending state on reset because EDECR.SS is set to 1. EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, no syndrome or Halting Step, normal.

In all cases, if EDSCR.STATUS is not set to Halting Step, no syndrome, then it must indicate whether the stepped instruction was a Load-Exclusive instruction by setting EDSCR.STATUS to Halting Step, normal or Halting Step, exclusive.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

In an implementation that always sets EDSCR.STATUS to Halting Step, no syndrome is not compliant.

### H3.2.9 Pseudocode description of Halting Step debug events

There are two pseudocode functions for Halting Step debug events:

- `RunHaltingStep()`. This is called after an instruction has executed and any exception generated by the instruction is taken. It is also called after taking a reset before executing any instructions. That is, reset is treated like an asynchronous exception, even if EDECR.RCE == 1. `RunHaltingStep()` affects the next instruction.
- `CheckHaltingDebugStep()`. This is called before the next instruction is executed. If a step is pending, it generates the debug event.

```
// RunHaltingStep()
// ================

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    // if "exception_generated" == TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exeception.
    // "reset" = TRUE if exiting reset state into the highest EL.
    if reset then assert !Halted();             // Cannot come out of reset halted

    active = EDECR.SS == '1' && !Halted();

    if active && reset then                     // Coming out of reset with EDECR.SS set.
        EDESR.SS = '1';
    elsif active && HaltingAllowed() then
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;

// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H3.3 Halt Instruction debug event

A Halt Instruction debug event is generated when EDSCR.HDE == 1, halting is allowed, and software executes the Halting software breakpoint instruction, HLT.

The pseudocode for Halt Instruction Debug events is described in *HLT* on page C6-531 for A64 and *HLT* on page F6-2685 for A32 and T32.

HLT never generates a debug exception. It is treated as UNDEFINED if EDSCR.HDE == 0, or if halting is prohibited.

**Note**

A debugger can replace a program instruction with a Halt instruction to generate a Halting Software Breakpoint. Debuggers that use the HLT instruction must be aware of the ARMv8-A rules for concurrent modification of executable code, CMODX. The rules for concurrent modification and execution of instructions do not allow one thread of execution or an external debugger to replace an instruction with an HLT instruction when these same instructions are being executed by a different thread of execution. See *Concurrent modification and execution of instructions* on page B2-82.

The T32 HLT instruction is unconditionally executed inside an IT block, even when it is treated as UNDEFINED. The A32 HLT instruction is CONSTRAINED UNPREDICTABLE if the condition code field is not 0b1110, with the set of behaviors the same as for BKPT. See Appendix K1 *Architectural Constraints on UNPREDICTABLE behaviors*.

**Note**

The HLT instruction is part of the external debug solution for ARMv8-A. As such, the presence of the HLT instruction is not indicated in the ID registers. In particular, the AArch32 CP15 register ID_ISAR0.Debug does not indicate the presence of the HLT instruction.

### H3.3.1 HLT instructions as the first instruction in a T32 IT block

In an implementation that supports the ITD control, the architecture permits a combination of one T32 IT instruction and certain other 16-bit T32 instruction to be treated as a single 32-bit instruction when the value of the ITD field that applies to the current Exception level is 1.

The T32 HLT instruction cannot be combined with an IT instruction in this way. In an implementation that supports the ITD control, if the first instruction in an IT block is an HLT instruction, then the behavior of the instruction depends on the value of the applicable ITD field:

- If the value of the ITD field is 1, then the combination is treated as UNDEFINED and an Undefined Instruction exception is generated either by the IT instruction or by the HLT instruction.
- If the value of the ITD field is 0, then the HLT instruction either:
  - Is treated as UNDEFINED and generates an Undefined Instruction exception.
  - Generates an HLT Instruction debug event.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

To set an HLT Instruction debug event on the first instruction of an IT block, debuggers must replace the IT instruction with an HLT instruction to ensure consistent behavior.

The ITD control fields are:

**HSCTLR.ITD** Applies to execution at EL2 when EL2 is using AArch32.

**SCTLR.ITD** Applies to execution at EL0 or EL1 when EL1 is using AArch32.

**SCTLR_EL1.ITD**
Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

**Note**

An HLT instruction is always unconditional, even within an IT block.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H3.4 Exception Catch debug event

Exception Catch debug events:

- Are generated when the corresponding bit in the Exception Catch Control Register, EDECCR, is set to 1 on all entries to a given Exception level. This means:
  - Exceptions taken to the Exception level.
  - Exception returns to the Exception level.
  - Entry to the Exception level due to reset. This is an overlap with a Reset Catch debug event. See *Reset Catch debug events* on page H3-4880.
- Are taken synchronously, after entry to the Exception level.
- Ignore the Execution state of the target Exception level.
- Are ignored if halting is prohibited.

The EDECCR contains two fields:

- One field for Non-secure state.
- One field for Secure state.

Each field contains one bit for each Exception level in that state. Bits corresponding to Exception levels that are not implemented are RES0. See *EDECCR, External Debug Exception Catch Control Register* on page H9-5008.

**Note**

- EDECCR does not replace DBGVCR:
  - DBGVCR is retained in AArch32 state for backwards compatibility.
  - DBGVCR is ignored in AArch64 state and never generates entries to Debug state.
  - DBGVCR cannot be accessed by the external debug interface.
- EDECCR is only visible as OSECCR_EL1 by System Register instructions in AArch64 state, and as DBGOSECCR by CP14 register access instructions in AArch32 state, when the OS Lock is locked to allow software to save and restore it over a powerdown.
- Exception Catch debug events are not disabled when the OS Lock is locked.

For an Exception Catch debug event generated after taking an exception to a trapped Exception level:

- The PE must not fetch instructions from the vector address before entering Debug state, if address translation is disabled in the translation regime at the target Exception level.
- On entering Debug state:
  - The current Exception level is the target Exception level of the exception.
  - The ELR, SPSR, ESR, and other syndrome registers contain information about the exception.
  - DLR contains the exception vector address.

### H3.4.1 Prioritization of Exception Catch debug events

The following rules define the prioritization of Exception Catch debug events:

- It is IMPLEMENTATION DEFINED whether Exception Catch debug events are higher or lower priority than each of Software Step exceptions and Halting Step debug events.
- Exception Catch debug events are higher priority than all synchronous exceptions other than Software Step exceptions.
- Exception Catch debug events are lower priority than Reset Catch debug events.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

As described in *Synchronous exception prioritization for exceptions taken to AArch64* on page D1-1564, an exception trapping form of a Vector Catch debug event might generate a second debug exception as part of the exception entry, before the Exception Catch debug event is taken. See *Vector Catch exceptions* on page D2-1685 or *Vector Catch exceptions* on page G2-3937.

A second unmasked asynchronous exception can be taken before the PE enters Debug state. If this second exception does not generate an Exception Catch debug event, the exception handler executed at the higher Exception level later returns to the trapped Exception level, causing the Exception Catch debug event to be generated again.

See also *Debug state entry and debug event prioritization* on page H2-4823.

### H3.4.2 UNPREDICTABLE generation of Exception Catch debug events

When the PE is executing code at a given Exception level and the corresponding EDECCR bit is 1, it is CONSTRAINED UNPREDICTABLE whether an Exception Catch debug event is generated.

**Note**

It is possible to generate Exception Catch debug events:

- As a trap on all instruction fetches from the trapped Exception level as part of an instruction fetch.
- On entry to the Exception level, as described in *Detailed Halting Step state machine behavior* on page H3-4867.

This is similar to the implementation options allowed for Vector Catch debug events. The architecture does not require that the event is generated following an `ISB` operation executed at the Exception level.

Examples of this are:

- If the debugger writes to EDECCR so that the current Exception level is trapped.
- If the OS restore code writes to OSECCR so that the current Exception level is trapped.
- If the code executing in AArch32 state changes the Exception level or security state other than by an exception return, and the target Exception level is trapped. See *State and mode changes without explicit context synchronization operations* on page G2-3946.

### H3.4.3 Examples of Exception Catch debug events

If EDECCR == `0x20`, meaning that the Exception Catch debug event is enabled for Non-secure EL1, then the following exceptions generate Exception Catch debug events:

- An exception taken from Non-secure EL0 to Non-secure EL1.
- An exception return from EL2 to Non-secure EL1.
- An exception return from EL3 to Non-secure EL1.

For example, on taking a Data Abort exception from Non-secure EL0 to Non-secure EL1, using AArch64:

- ELR_EL1 and SPSR_EL1 are written with the preferred return address and PE state for a return to EL0.
- ESR_EL1 and FAR_EL1 are written with the syndrome information for the exception.
- DLR_EL0 is set to VBAR_EL1 + `0x400`, the synchronous exception vector.
- DSPSR_EL0 is written with the PE state for an exit to EL1.

The following do not generate Exception Catch debug events:

- An exception taken from Non-secure EL0 to EL2 or EL3.
- An exception return from EL2 to Non-secure EL0.
- An exception taken from Secure EL0 to Secure EL1.
- An exception return from EL3 to Secure EL1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H3.4.4 Pseudocode description of Exception Catch debug events

The pseudocode for the CheckExceptionCatch() function is as follows:

```
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch()
    // Called after taking an exception, that is, such that IsSecure() and PSTATE.EL are correct
    // for the exception target.
    base = if IsSecure() then 0 else 4;
    if HaltingAllowed() && EDECCR<UInt(PSTATE.EL) + base> == '1' then
        Halt(DebugHalt_ExceptionCatch);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H3.5 External Debug Request debug event

External Debug Request debug events are asynchronous debug events.

An External Debug Request debug event is generated when signaled by the embedded cross-trigger. See Chapter H5 *The Embedded Cross-Trigger Interface*.

**Note**

ARMv8-A requires the implementation of an embedded cross-trigger.

An implementation might also support IMPLEMENTATION DEFINED ways of generating an External Debug Request debug event.

If an External Debug Request debug event is being asserted at the point where a reset is taken, then the PE enters Debug state before it completes execution of the first instruction following the reset, provided that the state into which the PE resets allows halting.

### H3.5.1 Pseudocode description of External Debug Request debug events

The pseudocode for the `ExternalDebugRequest()` function is as follows:

```
// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt(DebugHalt_EDBGRQ);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H3.6 OS Unlock Catch debug event

An OS Unlock Catch debug event is generated when EDECR.OSUCE == 1 and the state of the OS Lock changes from locked to unlocked.

When the OS Unlock Catch debug event is generated, it is recorded by setting EDESR.OSUC to 1, meaning it immediately becomes pending. However, it is not guaranteed to be taken immediately. See *Synchronization and Halting debug events* on page H3-4882.

OS Unlock Catch debug events are not generated if the OS Lock is unlocked when the PE is in Debug state. See also:

- *Debug behavior when the OS Lock is unlocked* on page H6-4932.
- *EDECR, External Debug Execution Control Register* on page H9-5010.
- *EDESR, External Debug Event Status Register* on page H9-5012.

EDESR.OSUC is cleared to 0 on a Warm reset and on exiting Debug state.

### H3.6.1 Using the OS Unlock Catch debug event

If the debugger attempts to access a debug register when the Core power down domain is completely off or in a low-power state in which the core power domain registers cannot be accessed, and that access returns an error, then the debugger must retry the access. However, if the Core power domain is regularly powered down, this can lead to unreliable debugger behavior.

The debugger can program a Reset Catch debug event to halt the PE when it has powered up, and can program the debug registers from Debug state. However, if the PE boot software restores the debug registers, as described in *Debug OS Save and Restore sequences* on page H6-4930, then newly written values are overwritten by the restore sequence.

The debugger can program an OS Unlock Catch debug event to halt the PE after the restore sequence has completed, and program the debug registers from Debug state.

### H3.6.2 Pseudocode description of OS Unlock Catch debug event

The `CheckOSUnlockCatch()` function is called when the OS Lock is unlocked.

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if EDECR.OSUCE == '1' && !Halted() then EDESR.OSUC = '1';
```

The `CheckPendingOSUnlockCatch()` function is called before an instruction is executed. If an OS Unlock Catch is pending, it generates the debug event.

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H3.7 Reset Catch debug events

A Reset Catch debug event is generated when EDECR.RCE == 1 and the PE exits reset state. When the Reset Catch debug event is generated, it is recorded by setting EDESR.RC to 1.

If halting is allowed when the event is generated, the Reset Catch debug event is taken immediately and synchronously. On entering Debug state, DLR has the address of the reset vector. The PE must not fetch any instructions from memory.

Otherwise, the Reset Catch debug event is pended and taken when halting is allowed. See also:

- *Synchronization and Halting debug events* on page H3-4882.
- *EDECR, External Debug Execution Control Register* on page H9-5010.
- *EDESR, External Debug Event Status Register* on page H9-5012.

This means that EDESR.RC is set to the value of EDECR.RCE on a Warm reset. EDESR.RC is cleared to 0 on exiting Debug state.

### H3.7.1 Pseudocode description of Reset Catch debug event

The `CheckResetCatch()` function is called after reset before executing any instruction.

```
// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if EDECR.RCE == '1' then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

The `CheckPendingResetCatch()` function is called before an instruction is executed. If a Reset Catch is pending, it generates the Reset Catch debug event.

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H3.8 Software Access debug event

When the value of EDSCR.TDA == 1, software access to the following debug AArch64 System registers and AArch32 CP14 registers cause a trap to Debug state:

- The Breakpoint Value Registers, DBGBVR.
- The Breakpoint Control Registers, DBGBCR.
- The Watchpoint Value Registers, DBGWVR.
- The Watchpoint Control Registers, DBGWCR.

However, EDSCR.TDA is ignored if one of:

- The value of OSLSR.OSLK == 1, meaning that the OS Lock is locked.
- Halting is prohibited. See *Halting allowed and halting prohibited* on page H2-4821.
- The register access generates an exception.

**Note**

- DBGPRCR.CORENPDRQ (Core No-powerdown Request), DCC registers, and CLAIM tag bits are also shared, but are deliberately excluded from this list.
- Only accesses as System registers in AArch64 state or CP14 registers in AArch32 state generate a trap. Accesses to the memory-mapped interface by a PE are not trapped.

### H3.8.1 Pseudocode description of Software Access debug event

The pseudocode for `CheckSoftwareAccessToDebugRegisters()` is as follows:

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()

    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H3.9 Synchronization and Halting debug events

The behavior of external debug depends on:

- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.
- The state of the external authentication interface.

This means that any change to these registers or the external authentication interface requires explicit synchronization by a *Context synchronization operation* before the change takes effect. This ensures that for instructions appearing in program order after the change, the change affects the following:

- The generation and behavior of Software debug events. See *Synchronization and debug exceptions* on page D2-1701 for exceptions taken from AArch64 state, or *Synchronization and debug exceptions* on page G2-3945 for exceptions taken from AArch32 state.
- The generation of all Halting debug events.
- Taking a pending Halting debug event or other asynchronous Debug event. See:
  - *Pending Halting debug events*.
  - *Taking Halting debug events asynchronously* on page H3-4883.
- The behavior of the Halting Step state machine. See *Synchronization and the Halting Step state machine* on page H3-4870.

If there is an instruction between the change and the *Context synchronization operation*, it is CONSTRAINED UNPREDICTABLE whether the PE uses the old state or the new state.

For some registers, all read and write accesses that update the register occur in program order, without any additional synchronization, but others require an explicit *Context synchronization operation*. For more information on the synchronization of register updates see:

- *Synchronization requirements for System registers* on page D7-1908.
- *Synchronization of changes to the external debug registers* on page H8-4946.
- *State and mode changes without explicit context synchronization operations* on page G2-3946.

A change on the external authentication interface is typically asynchronous to software and can happen without a *Context synchronization operation*.

External Debug Request debug events must be taken in finite time, without requiring the synchronization of any necessary change to the external authentication interface.

Example H3-3 shows an example of the synchronization requirements.

**Example H3-3 Synchronization requirements**

---

Secure software locks up in a tight loop, so it executes indefinitely without any synchronization operations. An External debug request must be able to break the PE out of that loop. This is a requirement even if **DBGEN** or **SPIDEN** or both are LOW on entry to the loop, meaning that halting is prohibited, and are only asserted HIGH later.

---

### H3.9.1 Pending Halting debug events

A pending Halting debug event is taken when halting becomes allowed. This can happen without a *Context synchronization operation* if:

- The PE enters Non-secure state with `ExternalInvasiveDebugEnabled()` == TRUE, and this is not the result of an exception return. See *State and mode changes without explicit context synchronization operations* on page G2-3946.
- A change on the external authentication interface means halting becomes allowed in the current state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The OS Double Lock status, `DoubleLockStatus()` becomes FALSE. For example, this can happen when software clears OSDLR.DLK to 0 or sets DBGPRCR.COREPDRQ to 1.
- The debug event is an OS Unlock Catch debug event. OS Unlock Catch debug events are generated in a pending state, rather than taken synchronously.

In these cases a pending Halting debug event is taken asynchronously.

### H3.9.2 Taking Halting debug events asynchronously

The ARM architecture does not define when Halting debug events that are taken asynchronously are taken.

Any Halting debug event that is observed as pending in the EDESR before a *Context synchronization operation*, or an External Debug Request debug event that is asserted before a *Context synchronization operation*, is taken and the PE enters Debug state before the first instruction following the *Context synchronization operation* completes its execution. This is only possible if halting is allowed after completion of the *Context synchronization operation*.

If the first instruction after the *Context synchronization operation* generates a synchronous exception, or an asynchronous exception is also pending, then the architecture does not define the order in which the debug event and the exception or exceptions are taken, unless both:

- A Halting Step debug event is pending. EDESR.SS == 1.
- The *Context synchronization operation* is an exception return from a state where halting is prohibited to a state where halting is allowed.

**Note**

This applies to an exception return from Secure state with `ExternalSecureInvasiveDebugEnabled()` == FALSE to Non-secure state with `ExternalInvasiveDebugEnabled()` == TRUE.

In this case the order in which the debug events are handled is specified to avoid a double-step. See *Entering the active-pending state* on page H3-4869.

An External Debug Request debug event that is being asserted when the PE comes out of reset is taken, and the PE enters Debug state before the first instruction after the reset completes its execution, provided that halting is allowed when the PE exits reset state

**Note**

These rules are based on the rules that apply to taking asynchronous exceptions. See *Asynchronous exception types, routing, masking and priorities* on page D1-1569.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Chapter H4
# The Debug Communication Channel and Instruction Transfer Register

This chapter describes communication between a debugger and the implemented debug logic, using the *Debug Communications Channel* (DCC) and the *Instruction Transfer Register* (ITR), and associated control flags. It contains the following sections:


- *Introduction* on page H4-4886.
- *DCC and ITR registers* on page H4-4887.
- *DCC and ITR access modes* on page H4-4889.
- *Flow control of the DCC and ITR registers* on page H4-4893.
- *Synchronization of DCC and ITR accesses* on page H4-4897.
- *Interrupt-driven use of the DCC* on page H4-4902.
- *Pseudocode description of the operation of the DCC and ITR registers* on page H4-4903.


**Note**

Where necessary Table K13-1 on page K13-5708 disambiguates the general register references used in this chapter.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H4.1 Introduction

The *Debug Communications Channe*l, DCC, is a channel for passing data between the PE and an external agent, such as a debugger. The DCC provides a communications channel between:

- An external debugger, described as the *debug host*.
- The debug implementation on the PE, described as the *debug target*.

The DCC can be used:

- As a 32-bit full-duplex channel.
- As a 64-bit half-duplex channel.

The DCC is an essential part of Debug state operation and can also be used in Non-debug state.

The *Instruction Transfer Register*, ITR, passes instructions to the PE to execute in Debug state.

The PE includes flow-control mechanisms for both the DCC and ITR.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H4.2 DCC and ITR registers

The DCC comprises *data transfer registers*, the DTRs, and associated flow-control flags. The data transfer registers are DTRRX and DTRTX.

The ITR comprises a single register, EDITR, and associated flow-control flags.

In AArch64 state, software can access the data transfer registers as:

- A receive and transmit pair for 32-bit full duplex operation:
  - The write-only DBGDTRTX_EL0 register to transmit data.
  - The read-only DBGDTRRX_EL0 register to receive data.
- A single 64-bit read/write register, DBGDTR_EL0, for 64-bit half-duplex operation.
- The read/write OSDTRTX_EL1 and OSDTRRX_EL1 registers for save and restore.

In AArch32 state, software can only access the data transfer registers as:

- A receive and transmit pair, for 32-bit full duplex operation:
  - The write-only DBGDTRTXint register to transmit data.
  - The read-only DBGDTRRXint register to receive data.
- The read/write DBGDTRTXext and DBGDTRRXext registers for save and restore.

The data transfer registers are also accessible by the external debug interface as a pair of 32-bit registers, DBGDTRRX_EL0 and DBGDTRTX_EL0. Both registers are read/write, allowing both 32-bit full-duplex and 64-bit half-duplex operation.

The DCC flow-control flags are EDSCR.{RXfull, TXfull, RXO, TXU}:

- The RXfull and TXfull ready flags are used for flow-control and are visible to software in the debug System registers in DCCSR.
- The RX overrun flag, RXO, and the TX underrun flag, TXU, report flow-control errors.
- The flow-control flags are also accessible by software as simple read/write bits for saving and restoring over a powerdown when the OS Lock is locked in DSCR.
- The flow-control flags are accessible from the external debug interface in EDSCR.

Figure H4-1 on page H4-4888 shows the System register and external debug interface views of the EDSCR and DTR registers in both AArch64 state and AArch32 state. These figures do not include the save and restore views.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

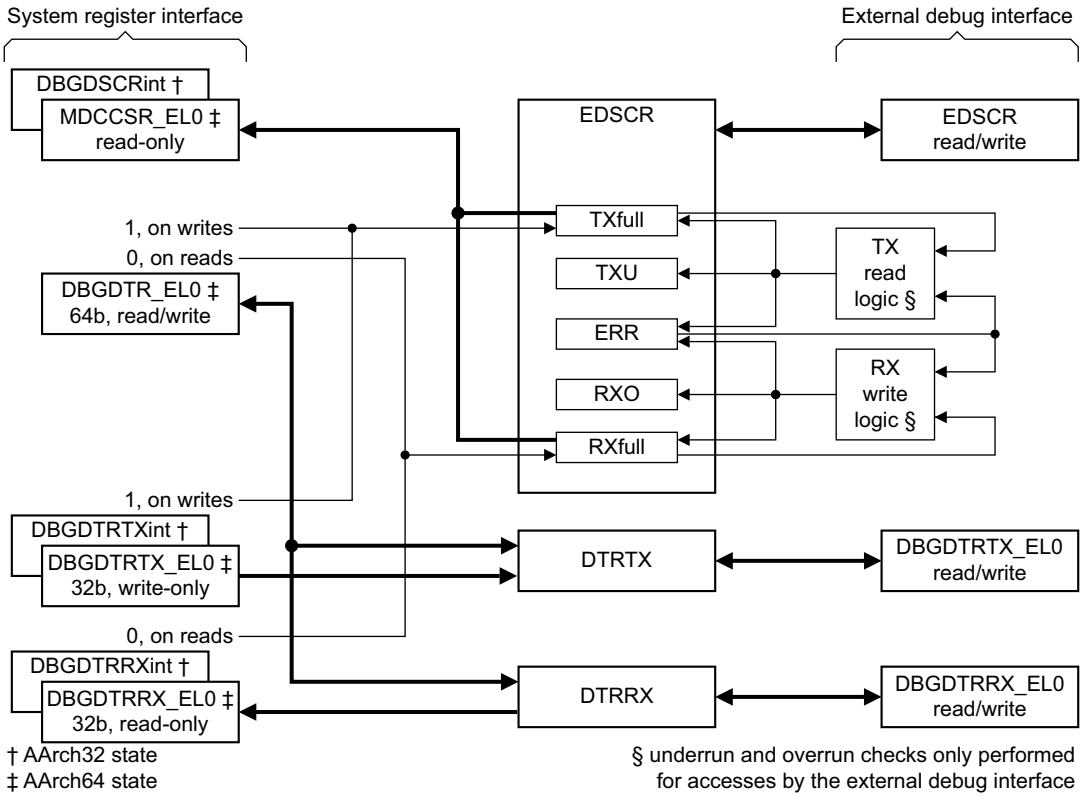



**Figure H4-1 System register and external debug interface views of EDSCR and DTR registers, Normal access mode**

EDITR and the ITR flow-control flags, EDSCR.{ITE, ITO} are accessible only by the external debug interface:

- The EDITR specifies an instruction to execute in Debug state.
- The ITR empty flag, ITE, is used for flow-control.
- The ITR overrun flag, ITO, reports flow-control errors.

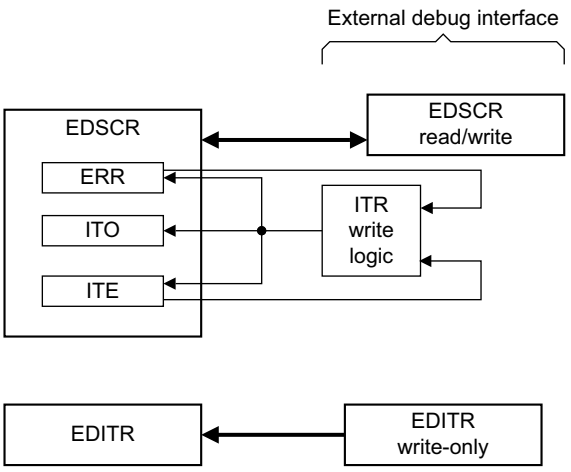



**Figure H4-2 External debug interface views of EDSCR and EDITR registers, Normal access mode**

The sticky overflow flag, EDSCR.ERR, is used by both the DCC and ITR to report flow-control errors.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H4.3 DCC and ITR access modes

The DCC and ITR support two access modes:

- *Normal access mode*, when EDSCR.MA == 0 or the PE is in Non-debug state.
- *Memory access mode* on page H4-4890, when EDSCR.MA == 1 and the PE is in Debug state.

### H4.3.1 Normal access mode

The Normal access mode allows use of the DCC as a communications channel between target and host. It also allows the use of the ITR for issuing instructions to the PE in Debug state.

In Normal access mode, if there is no overrun or underrun, the following occurs:

**For accesses by software:**

- Direct writes to DBGDTRTX update the value in DTRTX and indirectly write 1 to TXfull.
- Direct reads from DBGDTRRX return the value in DTRRX and indirectly write 0 to RXfull.
- In AArch64 state, direct writes to DBGDTR_EL0 update both DTRTX and DTRRX, indirectly write 1 to TXfull, and do not change RXfull:
  - DTRTX is set from bits[31:0] of the transfer register.
  - DTRRX is set from bits[63:32] of the transfer register.
- In AArch64 state, direct reads from DBGDTR_EL0 return the concatenation of DTRRX and DTRTX, indirectly write 0 to RXfull, and do not change TXfull:
  - Bits[31:0] of the transfer register are set from DTRRX.
  - Bits[63:32] of the transfer register are set from DTRTX.

**Note**

For DBGDTR_EL0, the word order is reversed for reads with respect to writes.

Software reads TXfull and RXfull using DCCSR.

**For accesses by the external debug interface:**

- Writes to EDITR trigger the instruction to be executed if the PE is in Debug state:
  - If the PE is in AArch64 state, this is an A64 instruction.
  - If the PE is in AArch32 state, this is a T32 instruction. The T32 instruction is a pair of halfwords where the first halfword is taken from the lower 16-bits, and the second halfword is taken from the upper 16-bits.
- Reads of DBGDTRTX_EL0 return the value in DTRTX and indirectly write 0 to TXfull.
- Writes to DBGDTRTX_EL0 update the value in DTRTX and do not change TXfull.
- Reads of DBGDTRRX_EL0 return the value in DTRRX and do not change RXfull.
- Writes to DBGDTRRX_EL0 update the value in DTRRX and indirectly write 1 to RXfull.

TXfull and RXfull are visible to the external debug interface in EDSCR.

The PE detects overrun and underrun by the external debug interface, and records errors in EDSCR.{TXU, RXO, ITO, ERR}. See *Flow control of the DCC and ITR registers* on page H4-4893.

See also *Synchronization of DCC and ITR accesses* on page H4-4897.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H4.3.2 Memory access mode

When the PE is in Debug state, *Memory access mode* can be selected to accelerate word-aligned block reads or writes of memory by an external debugger. Memory access mode can only be enabled in Debug state, and no instructions can be issued directly by the debugger when in Memory access mode.

If there is no overrun or underrun when in Memory access mode, an access by the external debug interface results in the following:

- External reads from DBGDTRTX_EL0 cause:
  1. The existing value in DTRTX to be returned. This clears EDSCR.TXfull to 0.
  2. The equivalent of `LDR W1,[X0],#4`, if in AArch64 state, or `LDR R1,[R0],#4`, if in AArch32 state, to be executed.
  3. The equivalent of the `MSR DBGDTRTX_EL0,X1` instruction, if in AArch64 state, or the `MCR p14,0,R1,c0,c5,0` instruction, if in AArch32 state, to be executed.
  4. EDSCR.{TXfull, ITE} to be set to {1,1}, and X1 or R1 to be set to an UNKNOWN value.
- External writes to DBGDTRRX_EL0 cause:
  1. The value in DTRRX to be updated. This sets EDSCR.RXfull to 1.
  2. The equivalent of the instruction `MRS X1,DBGDTRRX_EL0`, if in AArch64 state, or `MRC p14,0,R1,c0,c5,0` if in AArch32 state, to be executed.
  3. The equivalent of the instruction `STR W1,[X0],#4`, if in AArch64 state, or `STR R1,[R0],#4`, if in AArch32 state, to be executed.
  4. EDSCR.{RXfull, ITE} to be set to {0,1}, and X1 or R1 to be set to an UNKNOWN value.
- External reads from DBGDTRRX_EL0 return the last value written to DTRRX.
- External writes to EDITR generate an overrun error.

During these accesses, EDSCR.{TXfull, RXfull, ITE} are used for flow control.

The architecture does not require precisely when these flags are set or cleared by the sequence of operations outlined in this section. For example, in the case of an external write to DBGDTRRX_EL0, in AArch64 state, RXfull might be cleared after step 2, or it might not be cleared until after step 3, as an implementation is free to fuse these steps into a single operation. The architecture does require that the flags are set as at step 4 when the PE is ready to accept a further read or write without causing an overrun error or an underrun error.

The process outlined in this section represents a simple sequential execution model of Memory access mode. An implementation is free to pipeline, buffer, and re-order instructions and transactions, as long as the following remain true:

- Data items are transferred into and out of the DTR in order and without loss of data, other than as a result of an overrun or an underrun.
- Data Aborts occur in order.
- The constraints of the memory type are met.
- In the list describing *External reads from DBGDTRTX_EL0*:
  - The `MSR` equivalent operation at step 3 of the sequence reads the value loaded by step 2.
  - If the list is performed in a loop, for all but the first iteration of this list, the value read by step 1 returns the values written by the `MSR` equivalent operation at the previous iteration of step 3.
- In the list describing *External writes to DBGDTRRX_EL0*:
  - The `MRS` equivalent operation at step 2 of the sequence returns the value written at step 1.
  - The `STR` equivalent at step 3 of the sequence writes the value read at step 2.
- If the PE cannot accept a read or write, as applicable, during the sequence, then the flags are updated to indicate an overrun or underrun


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

See *Flow control of the DCC and ITR registers* on page H4-4893 for more information on overrun and underrun.

### Ordering, access sizes and effect on exclusive monitors

For the purposes of memory ordering, access sizes, and effect on the exclusive monitor, accesses in Memory access mode are consistent with Load/Store word instructions executed by the PE.

### Data aborts

If the memory access generates a Data Abort, then:

- The Data Abort exception is taken. See *Exceptions in Debug state* on page H2-4851. In particular, EDSCR.ERR is set to 1. See *Cumulative error flag* on page H4-4895.
- Register R0 retains the address that generated the abort.
- Register R1 is set to an UNKNOWN value.
- EDSCR.TXfull, for a load, or EDSCR.RXfull, for a store, is set to an UNKNOWN value.
- DTRTX, for a load, or DTRRX, for a store, is set to an UNKNOWN value.
- EDSCR.ITE is set to 1.

### Illegal Execution state exception

If PSTATE.IL is set to 1 when EDSCR.MA == 1, then on an external write access to DBGDTRRX_EL0 or an external read from DBGDTRTX_EL0, it is CONSTRAINED UNPREDICTABLE whether the PE:

- Takes an Illegal Execution state exception without performing any operations. In this case:
  - EDSCR.ERR is set to 1, see *Cumulative error flag* on page H4-4895.
  - Register R0 is unchanged.
  - Register R1 is set to an UNKNOWN value.
  - EDSCR.TXfull or EDSCR.RXfull, as applicable, is set to an UNKNOWN value.
  - DTRTX or DTRRX, as applicable, is set an UNKNOWN value.
  - EDSCR.ITE is set to 1.

  See also *Exceptions in Debug state* on page H2-4851.
- Ignores PSTATE.IL.

**Note**

The typical usage model for Memory access mode involves executing instructions in Normal access mode to set up X0 before setting EDSCR.MA to 1. These instructions generate an Illegal state exception if PSTATE.IL is set to 1.

### Alignment constraints

If the address in R0 is not aligned to a multiple of four, the behavior is as follows:

- For each external DTR access a CONSTRAINED UNPREDICTABLE choice of:
  1. The PE makes an unaligned memory access to R0. If alignment checking is enabled for the memory access, this generates an Alignment fault.
  2. The PE makes a memory access to `Align(X[0],4)` in AArch64 state, or `Align(R[0],4)` in AArch32 state.
  3. The PE generates an Alignment fault, regardless of whether alignment checking is enabled.
  4. The PE does nothing.
- Following each memory access, if there is no Data Abort, R0 is updated with an UNKNOWN value.
- For external writes to DBGDTRRX_EL0, if the PE writes to memory, an UNKNOWN value is written.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- For external reads of DBGDTRTX_EL0 an UNKNOWN value is returned.
- The RXfull and TXfull flags are left in an UNKNOWN state, meaning that a DBGDTRTX_EL0 read can trigger a TX underrun, and a DBGDTRTX_EL0 write can trigger an RX overrun.

### H4.3.3 Memory-mapped accesses to the DCC and ITR

Writes to the flags in EDSCR by external debug interface accesses to the DCC and the ITR registers are indirect writes, because they are a side-effect of the access. The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see *Register access permissions for memory-mapped accesses* on page H8-4950.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H4.4 Flow control of the DCC and ITR registers

This sub-section describes the flow-control of the DCC and ITR registers:

- *Ready flags*.
- *Buffering writes to EDITR*.
- *Overrun and underrun flags* on page H4-4894.
- *Cumulative error flag* on page H4-4895.

### H4.4.1 Ready flags

In Normal access mode:

- For the DTR registers there are two ready flags:
  - EDSCR.RXfull == 1 indicates that DBGDTRRX_EL0 contains a valid value that has been written by the external debugger and not yet read by software running on the target.
  - EDSCR.TXfull == 1 indicates that DBGDTRTX_EL0 contains a valid value that has been written by software running on the target and not yet read by an external debugger.
- For the ITR register there is a single ready flag:
  - EDSCR.ITE == 1 indicates that the PE is ready to accept an instruction to the ITR.

    **Note**

    The architecture permits a PE to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, as long as those instructions are discarded if EDSCR.ERR is set, either by an underrun or overrun or by any of the other error conditions described in this architecture, such as an instruction generating an abort.

In Memory access mode:

- EDSCR.{RXfull, ITE} == {0,1} indicates that DBGDTRRX_EL0 is empty and the PE is ready to accept a word external write to DBGDTRRX_EL0.
- EDSCR.{TXfull, ITE} == {1,1} indicates that DBGDTRTX_EL0 is full and the PE is ready to accept a word external read from DBGDTRTX_EL0.

All other values indicate that the PE is not ready, and result in a DTR overrun or underrun error, an ITR overrun error, or both, as defined in *Overrun and underrun flags* on page H4-4894.

EDSCR.{ITE, RXfull, TXfull} shows the status of the ITR and DCC registers. It ignores the question of whether a read or write cannot be accepted because, for example, EDSCR.ERR is set or the software lock is locked for memory-mapped accesses (EDLSR.SLK == 1).

### H4.4.2 Buffering writes to EDITR

The architecture permits a processor to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, provided that:

- Those instructions are discarded if EDSCR.ERR is set to 1, either by an underrun or an overrun, or by any other error conditions described in this architecture, such as an instruction generating an abort.
- The PE maintains the simple sequential execution model with the order of instructions determined by the order in which the PE accepts the EDITR writes. In particular, the buffered instructions must be executed in the Execution state consistent with a simple sequential execution of the instructions, even if one of the previous instructions is a state changing operation, such as `DCPS` or `DRPS`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H4.4.3 Overrun and underrun flags

Each of the ready flags has a corresponding overrun or a corresponding underrun flag. These are sticky status flags that are set if the register is accessed using the external debug interface when the corresponding ready flag is not in the ready state.

If the PE is in Debug state and Memory access mode, the corresponding error flag is also set if the PE is not ready to accept an operation because a previous load or store is still in progress. The sticky status flag remains set until cleared by writing 1 to EDRCR.CSE.

**Note**

The architecture permits a PE to continue to accept and buffer data to write to memory in Memory access mode.

Table H4-1 shows DCC and ITR ready flags and the overrun and underrun flags associated with them.

**Table H4-1 DCC and ITR ready flags and the associated overrun/underrun flags**

| External debug interface access | Overrun/Underrun condition | EDSCR flag |
|---|---|---|
| Write DBGDTRRX_EL0 | `EDSCR.RXfull == '1' \|\| (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0')` | RXO |
| Read DBGDTRTX_EL0 | `EDSCR.TXfull == '0' \|\| (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0')` | TXU |
| Write EDITR | `Halted() && (EDSCR.ITE == '0' \|\| EDSCR.MA == '1')` | ITO |

When an overrun or underrun flag is set to 1, the cumulative error flag, EDSCR.ERR, described in *Cumulative error flag* on page H4-4895, is also set to 1.

In the event of an external write to DBGDTRRX_EL0 or EDITR generating an overrun, or an external read from DBGDTRTX_EL0 generating an underrun:

- For a write, the written value is ignored.
- For a read, an UNKNOWN value is returned.
- EDSCR.TXfull, EDSCR.RXfull or EDSCR.ITE, as applicable, are not updated.

There is no overrun or underrun detection on external reads of DBGDTRRX_EL0 or external writes of DBGDTRTX_EL0.

There is no overrun or underrun detection of direct reads and direct writes of the DTR System registers by software:

- If RXfull == 0, a direct read of DBGDTRRX or DBGDTR_EL0 returns UNKNOWN.
- If TXfull == 1, a direct write of:
  - DBGDTRTX sets DTRTX to UNKNOWN.
  - DBGDTR_EL0 sets DTRRX and DTRTX to UNKNOWN.

See *DCC accesses in Non-debug state* on page H4-4898 for more information.

#### Accessing 64-bit data

In AArch64 state, a software access to the DBGDTR_EL0 register and an external debugger access to both DBGDTRRX_EL0 and DBGDTRTX_EL0 can perform a 64-bit half-duplex operation.

However, there is only overrun and underrun detection on one of the external debug registers. That is:

- If software directly writes a 64-bit value to DBGDTR_EL0, only TXfull is set to 1, meaning:
  - A subsequent external write to DBGDTRRX_EL0 would not be detected as an overrun.
  - If the external debugger reads DBGDTRTX_EL0 first, software might observe MDCCSR_EL0.TXfull == 0 and send a second value before the external debugger reads DBGDTRRX_EL0, leading to an undetected overrun.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- On external writes to both DBGDTRRX_EL0 and DBGDTRTX_EL0 only RXfull is set to 1, meaning:
  - A subsequent direct write of DBGDTRTX_EL0 would not be detected as an overrun.
  - If the external debugger writes to DBGDTRRX_EL0 first, software might observe MDCCSR_EL0.RXfull == 1 and read a full 64-bit value, before the external debugger writes to DBGDTRTX_EL0, leading to an undetected underrun.

To avoid this, debuggers need to be aware of the data size used by software for transfers and ensure that 64-bit data is read or written in the correct order. If the PE is in Non-debug state, this order is as follows:

- The external debugger must check EDSCR.{RXfull, TXfull} before each transfer.
- To receive a 64-bit value from the target, the external debugger must read DBGDTRRX_EL0 before reading DBGDTRTX_EL0.
- To send a 64-bit value to the target, the external debugger must write to DBGDTRTX_EL0 before writing DBGDTRRX_EL0.

Because three accesses are required to transfer 64 bits of data, 64-bit transfers are not recommended for regular communication between host and target. The use of underrun and overrun detection means that only one access is required for 32 bits of data when using 32-bit transfers.

In Debug state, the debugger controls the instructions executed by the PE, so these limitations do not apply. 64-bit transfers provide a means to transfer a 64-bit general register between the host and the target in Debug state.

### H4.4.4 Cumulative error flag

The cumulative error flag, EDSCR.ERR, is set to 1:

- On taking an exception from Debug state.
- On any signaled overrun or underrun in the DCC or ITR.

When EDSCR.ERR == 1:

- External reads of DBGDTRTX_EL0 do not have any side-effects.
- External writes to DBGDTRRX_EL0 are ignored.
- External writes to EDITR are ignored.
- No further instructions can be issued in Debug state. This includes any instructions previously accepted as external writes to EDITR that occur in program order after the instruction or access that caused the error.

This allows a debugger to stream data, or, in Debug state, instructions, to the target without having to:

- Check EDSCR.{RXfull, TXfull, ITE} before each access.
- Check EDSCR.{ITO, RXO, TXU} following each access, for overrun or underrun.
- Check PSTATE or other syndrome registers, or both, for an exception following each instruction executed in Debug state that might generate a synchronous exception.

The cumulative error flag remains set until cleared by writing 1 to EDRCR.CSE. See *EDRCR, External Debug Reserve Control Register* on page H9-5041.

For overruns and underruns, EDSCR.{ITO, RXO, TXU} record the error type.

#### Pseudocode description of clearing the error flag

The pseudocode for the `ClearStickyError()` function is as follows:

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';            // Clear TX underrun flag
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
EDSCR.RXO = '0';            // Clear RX overrun flag
if Halted() then            // in Debug state
    EDSCR.ITO = '0';        // Clear ITR overrun flag
EDSCR.ERR = '0';            // Clear cumulative error flag
return;
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H4.5 Synchronization of DCC and ITR accesses

In addition to the standard synchronization requirements for register accesses, the following subsections describe additional requirements that apply for the DCC and ITR registers:

- *Summary of System register accesses to the DCC*.
- *DCC accesses in Non-debug state* on page H4-4898.

In these sections, accesses by the external debug interface are referred to as external reads and external writes. Accesses to System registers are referred to as direct reads, direct writes, indirect reads, and indirect writes.

**Note**

In *Synchronization requirements for System registers* on page D7-1908 external reads and external writes are described as forms of indirect access. This whole section uses more explicit terminology.

The DTR registers and the DCC flags, TXfull and RXfull, form a communication channel, with one end operating asynchronously to the other. Implementations must respect the ordering of accesses to these registers in order to maintain the correct behavior of the channel.

External reads of, and external writes to DBGDTRRX_EL0 and DBGDTRTX_EL0 are asynchronous to direct reads of, and direct writes to, DBGDTRRX, DBGDTRTX, and in AArch64 state DBGDTR_EL0, made by software using System register access instructions. The direct reads and direct writes indirectly write to the DCC flags. The external reads and external writes indirectly read the DCC flags to check for underrun and overrun.

Throughout this section:

**DCC flags** Means any or all of the following:
- The EDSCR.{RXfull.TXfull} ready flags.
- The EDSCR.RXO overrun flag.
- The EDSCR.TXU underrun flag.
- The EDSCR.ERR cumulative error flag.

**ITR flags** Means any or all of the following:
- The EDSCR.ITE ready flag.
- The EDSCR.ITO overrun flag.
- The EDSCR.ERR cumulative error flag.

### H4.5.1 Summary of System register accesses to the DCC

System register accesses to the DTR registers are direct reads and writes of those registers, as shown in Table H4-2 on page H4-4898. Several of these instructions access the same registers using different encodings.

With the exception of the read and write bits, DBGDTRRX and DBGDTRTX are the same encoding, with exception of the read/write bits, but use different registers. The ARMv8 architecture governs the order of these instructions, as described in *Synchronization requirements for System registers* on page D7-1908. For more details, see the description of the individual register in the relevant chapter, Chapter D7 *AArch64 System Register Descriptions* or Chapter G6 *AArch32 System Register Descriptions*.

Table H4-2 on page H4-4898 shows a summary of System register accesses to the DCC.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table H4-2 Summary of System register accesses to the DCC**

| Operation | OS Lock | AArch64 (MRS/MSR) | AArch32 (MRC/MCR) | Description |
|---|---|---|---|---|
| Read | - | DBGDTRRX_EL0 | DBGDTRRXint | Direct read of DTRRX<br>Indirect write to the DCC flags |
| Write | - | DBGDTRTX_EL0 | DBGDTRTXint | Direct read of DTRTX<br>Indirect write to the DCC flags |
| Read/write | - | DBGDTR_EL0 | - | Direct read/write of both DTRRX and DTRTX<br>Indirect write to the DCC flags |
| Read | - | MDCCSR_EL0 | DBGDSCRint | Direct read of the DCC flags |
| Read/write | - | OSDTRRX_EL1 | DBGDTRRXext | Direct read/write of DTRRX |
| Read/write | - | OSDTRTX_EL1 | DBGDTRTXext | Direct read/write of DTRTX |
| Read | Unlocked | MDSCR_EL1 | DBGDSCRext | Direct read of DCC flags |
| Read/write | Locked | MDSCR_EL1 | DBGDSCRext | Direct read/write of DCC flags |

### H4.5.2 DCC accesses in Non-debug state

In Non-debug state DCC accesses are as described in *Normal access mode* on page H4-4889:

- If a direct read of DCCSR returns RXfull == 1, then a following direct read of DBGDTRRX, or in AArch64 state of DBGDTR_EL0, returns valid data and indirectly writes 0 to DCCSR.RXfull as a side-effect.
- If a direct read of DCCSR returns TXfull == 0, then a following direct write to DBGDTRTX, or in AArch64 state to DBGDTR_EL0, writes the intended value, and indirectly writes 1 to DCCSR.TXfull as a side-effect.

No context synchronization operation is required between these two instructions. Overrun and underrun detection prevents intervening external reads and external writes affecting the outcome of the second instruction.

The indirect write to the DCC flags as part of the DTR access instruction is made atomically with the DTR access.

Because a direct read of DBGDTRRX is an indirect write to DCCSR.RXfull, it must occur in program order with respect to the direct read of DCCSR, meaning it must not return a speculative value for DTTRX that predates the RXfull flag returned by the read of DCCSR. The direct write to DBGDTRTX must not be executed speculatively.

Direct reads of DBGDTRRX, or in AArch64 state DBGDTR_EL0, and DCCSR, must occur in program order with respect to other direct reads of the same register using the same encoding.

The following accesses have an implied order within the atomic access:

- In the simple sequential execution of the program the indirect write of the DCC flags occurs immediately after the direct DTR access.

  **Note**

  For an access to DBGDTR_EL0, this means the indirect write happens after both DBGDTRRX_EL0 and DBGDTRTX_EL0 have been accessed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In the simple sequential execution model, for an external read of DBGDTRTX_EL0 or an external write of DBGDTRRX_EL0:
  - The check of the DCC flags for overrun or underrun occurs immediately before the access.
  - If there is no underrun or overrun, the update of the DCC flags occurs immediately after the access.
  - If there is underrun or overrun, the update of the DCC underrun or overrun flags occurs immediately after the access.

All observers must observe the same order for accesses.

**Note**

These requirements do not create order where order does not otherwise exist. It applies only for ordered accesses.

Without explicit synchronization following external writes and external reads:

- The value written by the external write to DBGDTRRX_EL0 that does not overrun, must be observable to direct reads of DBGDTRRX and DBGDTR_EL0 in finite time.
- The DCC flags that are updated as a side-effect of the external write or external read must be observable:
  - To subsequent external reads of EDSCR.
  - To subsequent external reads of DBGDTRRX_EL0 when checking for underrun.
  - To subsequent external writes to DBGDTRTX_EL0 when checking for overrun.
  - To direct reads of DCCSR in finite time.

However, explicit synchronization is required to guarantee that a direct read of DCCSR returns up-to-date DCC flags. This means that if a signal is received from another agent that indicates that DCCSR must be read, an `ISB` is required to ensure that the direct read of DCCSR occurs after the signal has been received. This also synchronizes the value in DBGDTRRX, if applicable. However, if that signal is an interrupt exception triggered by **COMMIRQ**, **COMMTX**, or **COMMRX**, the exception entry is sufficient synchronization. See *Synchronization of DCC interrupt request signals* on page H4-4901.

Explicit synchronization is required following a direct read or direct write:

- To ensure that a value directly written to DBGDTRTX is observable to external reads of DBGDTRTX_EL0.
- To ensure that a value directly written to DBGDTR_EL0 is observable to external reads of DBGDTRTX_EL0 and DBGDTRRX_EL0.
- To guarantee that the indirect writes to the DCC flags that were a side-effect of the direct read or direct write have occurred, and therefore that the updated values are:
  - Observable to external reads of EDSCR.
  - Observable to external reads of DBGDTRRX_EL0 when checking for underrun.
  - Observable to external writes of DBGDTRTX_EL0 when checking for overrun.
  - Returned by a following direct read of DCCSR.

See also *Memory-mapped accesses to the DCC and ITR* on page H4-4892 and *Synchronization of changes to the external debug registers* on page H8-4946.

**Note**

These ordering rules mean that software:

- Must not read DBGDTRRX without first checking DCCSR.RXfull or if the previously-read value of DCCSR.RXfull is 0.

  It is not sufficient to read both registers and then later decide whether to discard the read value, as there might be an intervening write from the external debug interface.
- Must not write DBGDTRTX without first checking DCCSR.TXfull or if the previously-read value of DCCSR.TXfull is 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The write to DBGDTRTX overwrites the value in DTRTX, and the external debugger might or might not have read this value.

- Must ensure there is an explicit context synchronization operation following a DTR access, even if not immediately returning to read DCCSR again. This synchronization operation can be an exception return.

---

### Derived requirements

The rules for DCC accesses in Non-debug state are as follows:

- Following a direct read of DBGDTRRX when RXfull is 1:
  - If an external write to DBGDTRRX checks the RXfull flag for overrun and observes that the value of RXfull is 0, the value returned by the previous direct read must not be affected by the external write.
  - If an external read of EDSCR returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by a following external write to DBGDTRRX, and the following external write does not overrun.
- Following a direct read of DBGDTR_EL0, when RXfull is 1:
  - If an external write to DBGDTRRX checks the RXfull flag for overrun and observes that the value of RXfull is 0, the value returned by the previous direct read must not be affected by the external write nor by a following direct write to DBGDTRTX.
  - If an external read of EDSCR returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by subsequent external writes to DBGDTRRX and DBGDTRTX in any order, and the following external write of DBGDTRRX will not overrun.
- Following a direct write to DBGDTRTX, when TXfull is 0:
  - If an external read of DBGDTRTX checks the TXfull flag for underrun and observes that the value of TXfull is 1, the value returned by the external read must be the value written by the previous direct write.
  - If an external read of EDSCR returns a TXfull value of 1, then the value returned by a following external read of DBGDTRRX must be the value written by the previous direct read, and the subsequent external read will not underrun.
- Following a direct write to DBGDTR_EL0, when TXfull is 0:
  - If an external read of DBGDTRTX checks the TXfull flag for underrun and observes that the value of TXfull is 1, the values returned by the external read and by a subsequent external read of DBGDTRRX must be the value written by the previous direct write.
  - If an external read of EDSCR returns a TXfull value of 1, then the value returned by subsequent external reads of DBGDTRRX and DBGDTRTX, in any order, must be the value written by the previous direct read, and the subsequent external read of DBGDTRTX does not underrun.
- Following an external read of DBGDTRTX that does not underrun, if a direct read of DCCSR returns a TXfull value of 0, then the value returned by the external read must not be affected by a following direct write to DBGDTRTX.
- Following a first external read DBGDTRRX and a following second external read of DBGDTRTX that does not underrun, if a direct read of DCCSR returns a TXfull value of 0, then the values returned by the external reads must not be affected by a following direct write to DBGDTR_EL0.
- Following an external write to DBGDTRRX that does not overrun, if a direct read of DCCSR returns an RXfull value of 1, then the value returned by a following direct read of DBGDTRRX or DBGDTR_EL0 must be the value written by the previous external write.
- Following a first external write to DBGDTRTX and a following second external write to DBGDTRRX that does not overrun, if a direct read of DCCSR returns an RXfull value of 1, then the value returned by a subsequent direct read of DBGDTR_EL0 must return the values written by the previous external writes.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H4.5.3 Synchronization of DCC interrupt request signals

Following an external read or external write access to the DTR registers, the interrupt request signals, **COMMIRQ**, **COMMTX**, and **COMMRX**, must be updated in finite time without explicit synchronization.

The updated values must be observable to a direct read of DCCSR or DBGDTRRX, or a direct write of DBGDTRTX executed after taking an interrupt exception generated by the interrupt request. The updated values must also be observable to a direct write of DBGDTRTX executed after taking an interrupt exception generated by the interrupt request.

Following a direct read of DBGDTRRX or a direct write to DBGDTRRX, software must execute a context synchronization operation to guarantee the interrupt request signals have been updated in finite time. This synchronization operation can be an exception return.

### H4.5.4 DCC and ITR access in Debug state

In Debug state, stricter observability rules apply for instructions issued through the ITR, to maintain communication between a debugger and the PE, without requiring excessive explicit synchronization.

In Normal access mode, without explicit synchronization:

- A direct read or direct write of the DTR registers by an instruction written to EDITR must be observable to an external write or an external read in finite time:
  - A direct read of DBGDTRRX must be observable to an external write of DBGDTRRX_EL0.
  - A direct read of DBGDTR_EL0 must be observable to an external write of DBGDTRRX_EL0 and DBGDTRTX_EL0.
  - A direct write of DBGDTRTX must be observable to an external read of DBGDTRTX_EL0.
  - A direct write of DBGDTR_EL0 must be observable to an external read of DBGDTRRX_EL0 and DBGDTRTX_EL0.

  This includes the indirect write to the DCC flags that occurs atomically with the access as described in *DCC accesses in Non-debug state* on page H4-4898.

  The subsequent external write or external read must observe either the old or the new values of both the DTR contents and DCC flags. If the old values are observed, this typically results in overrun or underrun, assuming the old values of the DCC flags indicate an overrun or underrun condition, as would normally be the case.

  This means the debugger can observe the direct read or direct write without explicit synchronization and without explicitly testing the DCC flags in EDSCR, because it can rely on overrun and underrun tests.

- External reads of DBGDTRTX_EL0 that do not underrun and external writes to DBGDTRRX_EL0 that do not overrun must be observable to an instruction subsequently written to EDITR on completion of the first external access. This includes the indirect write to the DCC flags.

  This means that without explicit synchronization and without the need to first check the DCC flags in DCCSR:
  - If the instruction is a direct read of DBGDTRRX, it observes the external write.
  - If the instruction is a direct write of DBGDTRTX, it observes the external read.
- Writes to EDITR that do not overrun commit an instruction for execution immediately. The instruction must complete execution in finite time without requiring any further operation by the debugger.

- After an external write to the EDITR, the ITR flags that are updated as a side effect of that write must be observable by:
  - An external read of the EDSCR that follows the external write to the EDITR.
  - When checking for overrun, another external write to the EDITR that follows the original external write to the EDITR.

In Memory access mode, these requirements shift to the instructions implicitly executed by external reads and external writes of the DTR registers, as described in *Memory access mode* on page H4-4890.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H4.6 Interrupt-driven use of the DCC

ARM recommends implementations provide a level-sensitive DCC interrupt request through the IMPLEMENTATION DEFINED interrupt controller as a private peripheral interrupt for the originating PE.

**Note**

- In addition to connection to the interrupt controller ARM also recommends **COMMIRQ**, **COMMTX**, and **COMMRX** signals that might be implemented for use by any legacy system peripherals.
- GICv3 reserves a private peripheral interrupt number for the **COMMIRQ** interrupt.

The DCCINT register provides a first level of interrupt masking within the PE, meaning only a single interrupt source, **COMMIRQ**, is needed at the interrupt controller.

See also *Synchronization of DCC interrupt request signals* on page H4-4901.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H4.7 Pseudocode description of the operation of the DCC and ITR registers

The basic operation of the DCC and ITR registers is shown by the following pseudocode functions. These functions do not cover the behavior when OSLSR.OSLK == 1, meaning that the OS lock is locked.

The definition of the DTR Registers is:

```
bits(32) DTRRX;
bits(32) DTRTX;
```

The pseudocode for the DBGDTR_EL0() function is as follows:

```
// DBGDTR_EL0[] (write)
// ===================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ===================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```

The pseudocode for the DBGDTRRX_EL0() function is as follows:

```
// DBGDTRRX_EL0[] (external write)
// ===============================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xD5330501<31:0>);               // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401<31:0>);               // A64 "STR W1,[X0],#4"
            X[1] = bits(64) UNKNOWN;
        else
            ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
            R[1] = bits(32) UNKNOWN;

        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(32) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
            assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_EL0[] (external read)
// ==============================

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;
```

The pseudocode for the DBGDTRTX_EL0() function is as follows:

```
// DBGDTRTX_EL0[] (external read)
// ==============================
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then           // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN

    EDSCR.TXfull = '0';

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(32) UNKNOWN;
        else
            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';

        if !UsingAArch32() then
            X[1] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// ===============================

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write
    DTRTX = value;
    return;
```

The pseudocode for the EDITR() function is as follows:

```
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x088.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if !Halted() then return;                           // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';              // Overrun condition: block write
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return;
```

The pseudocode for the `CheckForDCCInterrupts()` function is as follows:

```
// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
               (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Chapter H5
# The Embedded Cross-Trigger Interface

This chapter describes the embedded cross-trigger interface. It contains the following sections:


- *About the Embedded Cross-Trigger (ECT)* on page H5-4908.
- *Basic operation on the ECT* on page H5-4910.
- *Cross-triggers on a PE in an ARMv8 implementation* on page H5-4914.
- *Description and allocation of CTI triggers* on page H5-4915.
- *CTI registers programmers' model* on page H5-4919.
- *Examples* on page H5-4920.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H5.1 About the Embedded Cross-Trigger (ECT)

The *Embedded Cross-Trigger*, ECT, allows a debugger to:

- Send trigger events to a PE. For example, this might be done to halt the PE.
- Send a trigger event to one or more PEs when a trigger event occurs on another PE. For example, this might be done to halt all PEs when one individual PE halts.

Figure H5-1 shows the logical structure of an ECT.

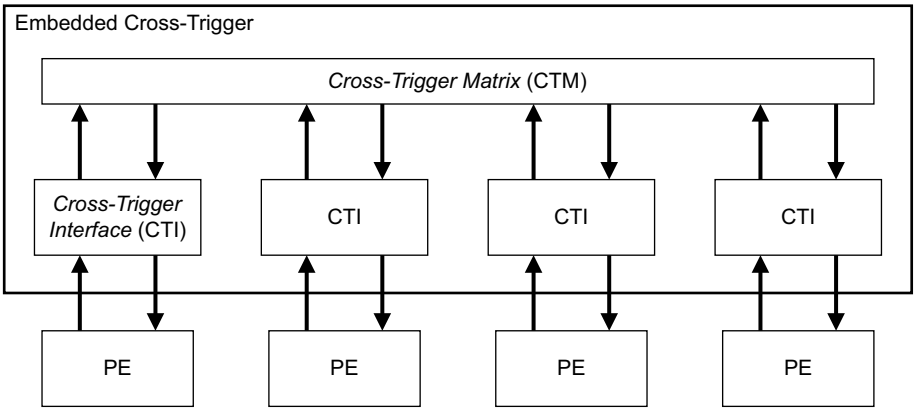



**Figure H5-1 Structure of an embedded cross-trigger**

The ECT can deliver many types of trigger events, which are described in the following sections:

- *Debug request trigger event* on page H5-4916.
- *Restart request trigger event* on page H5-4916.
- *Cross-halt trigger event* on page H5-4916.
- *Performance Monitors overflow trigger event* on page H5-4917.
- *Generic trace external input trigger events* on page H5-4917.
- *Generic trace external output trigger events* on page H5-4917.
- *Generic CTI interrupt trigger event* on page H5-4917.

An ARMv8-A implementation must:

- Include a cross-trigger interface, CTI.
- Implement at least the input and output triggers defined in this architecture.

See *Cross-triggers on a PE in an ARMv8 implementation* on page H5-4914.

In addition, ARM recommends that the cross-trigger includes:

- The ability to route trigger events between Trace extensions:
  - These typically have advanced event triggering logic.
- An output trigger to the interrupt controller.

**Note**

The ECT and CTI must only signal trigger events for external debugging. They must not route software events, such as interrupts. For example, the Performance Monitors overflow input trigger is provided to allow entry to Debug state on a counter overflow, and the output trigger to the interrupt controller is provided to generally allow events from the external debug sub-system to be routed to a software agent. However, the combination of the two must not be used as a mechanism to route Performance Monitors overflows to an interrupt controller.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H5.1.1 Implementation with a CoreSight CTI

For details of the recommended connections in an ARMv8-A implementation, see Appendix K2 *Recommended External Debug Interface*. See also *CoreSight™ SoC Technical Reference Manual*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H5.2 Basic operation on the ECT

The ECT comprises a Cross-Trigger Matrix, CTM, and one Cross-Trigger Interface, CTI, for each PE. The CTM passes events between the CTI blocks over channels. The CTM can have a maximum of 32 channels.

The main interfaces of the cross-trigger interface, CTI, are:

- The input triggers:
  - These are trigger event inputs from the PE to the CTI.
- The output triggers:
  - These are trigger event outputs from the CTI to the PE.
- The input channels:
  - These are channel event inputs from the cross-trigger matrix, CTM, to the CTI.
- The output channels:
  - These are channel event outputs from the CTI to the CTM.

Each CTI block has:

- Up to 32 input triggers that come from the PE:
  - The input triggers are numbered 0-31.
- Up to 32 output triggers that go to the PE:
  - The output triggers are numbered 0-31.

Figure H5-2 on page H5-4911 shows the logical internal structure of a CTI.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

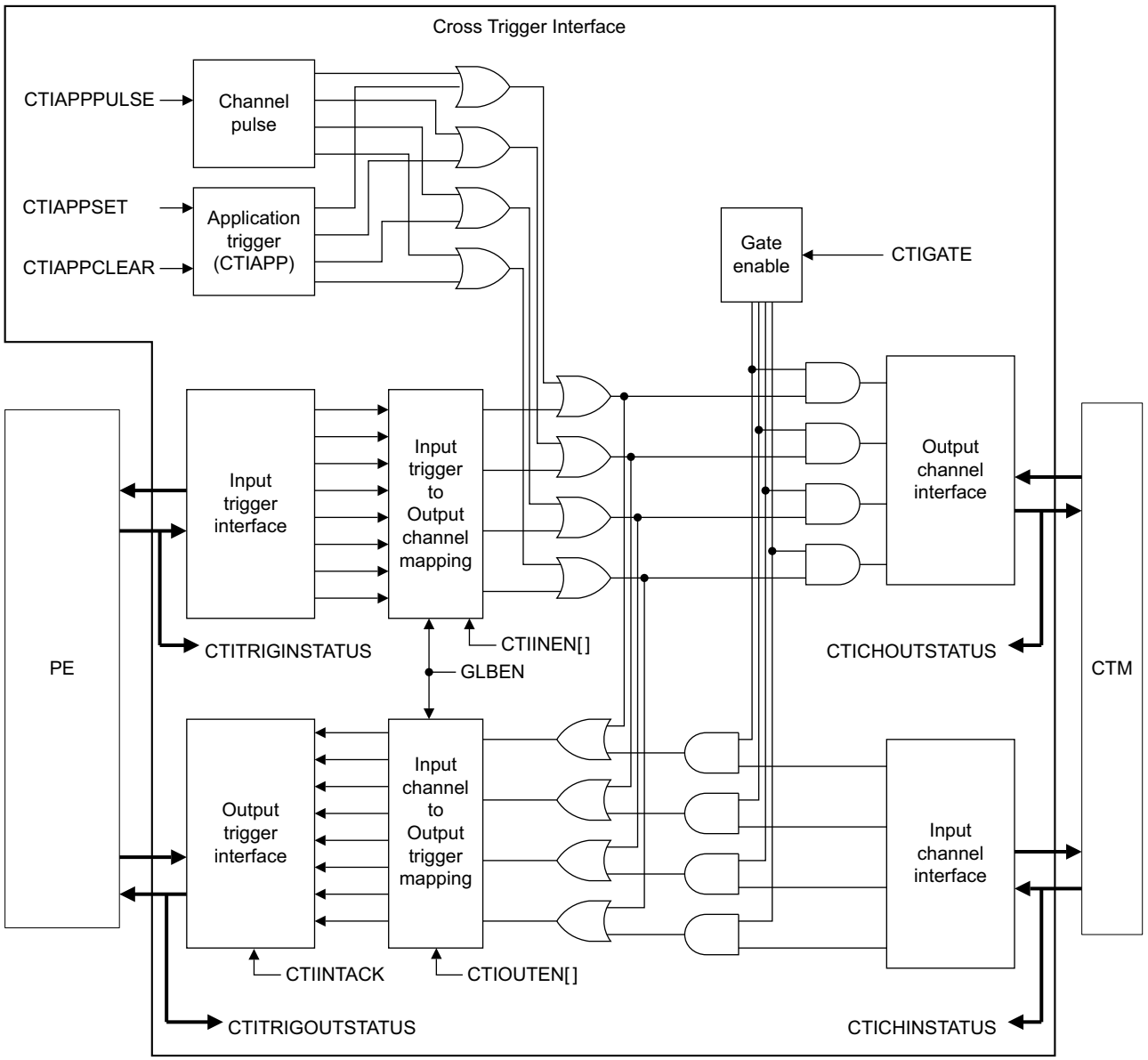



**Figure H5-2 Structure of a cross-trigger interface**

**Note**

- The number of triggers in IMPLEMENTATION DEFINED. Figure H5-2 shows eight input and eight output triggers.
- The number of channels is IMPLEMENTATION DEFINED. Figure H5-2 shows four channels.
- In Figure H5-2 the input channel gate function is a CTIv2 feature.

When the CTI receives an input trigger event, this generates channel events on one or more internal channels, according to the *mapping function* defined by the *Input trigger→output channel mapping registers*, CTIINEN<n>.

The CTI also contains an *application trigger* and *channel pulse* to allow a debugger to create channel events directly on internal channels by writing to the CTI control registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Channel events on each internal channel are passed to a corresponding output channel that is controlled by a *channel gate*. The channel gate can block propagation of channel events from an internal channel to an output channel.

The output channels from a CTI are combined, using a logical OR function, with the output channels from all other CTIs to form the input channels on other CTIs. The input channels of this CTI are the logical OR of the output channels on all other CTIs. This is the *cross-trigger matrix*, CTM. Therefore, the number of input channels must equal the number of output channels.

**Note**

The number of input triggers and output triggers is not required to be the same.

The internal channels form an internal cross-trigger matrix within the CTI. This delivers events directly from the input triggers to the output triggers. Therefore the number of internal channels is the same as the number of input and output channels on the external CTM, and there is a direct mapping between the two.

Channel events received on each input channel are passed to the corresponding internal channel. It is IMPLEMENTATION DEFINED whether the cross-trigger gate also blocks propagation of channel events from input channels to internal channels.

When the CTI receives a channel event on an internal channel this generates trigger events on one or more output triggers, according to the mapping function defined by the *Input channel → output trigger mapping registers*, CTIOUTEN<n>.

The CTI contains the input and output trigger interfaces to the PE and the interface of the cross-trigger matrix. The architecture does not define the signal protocol used on the trigger interfaces, and:

- It is IMPLEMENTATION DEFINED whether the CTI supports multicycle input trigger events.
- It is IMPLEMENTATION DEFINED whether the CTM supports multicycle channel events.

See *Multicycle events*.

However, an output trigger is asserted until acknowledged. The output trigger can be:

- Self-acknowledging. This means that no further action is required from the debugger.
- Acknowledged by the debugger writing 1 to the corresponding bit of CTIINTACK.

The time taken to propagate a trigger event from the first PE, through its CTI, across the CTM to another CTI, and thereby to a second PE is IMPLEMENTATION DEFINED.

**Note**

ARM recommends that this path is not longer than the shortest software communication path between those PEs. This is because if the first PE halts, the Cross-halt trigger event can propagate through the ECT and halt the second PE without causing software on the second PE to malfunction because the first PE is in Debug state and is not responding.

### H5.2.1 Multicycle events

A multicycle event is one with a continuous state that might persist over many cycles, as opposed to a discrete event. A typical implementation of a multicycle event is a level-based signal interface, whereas a discrete event might be implemented as a pulse signal or message.

CTI support for multicycle trigger events is IMPLEMENTATION DEFINED. Use of multicycle trigger events is deprecated. Of the architecturally defined input trigger events, the Performance Monitors overflow trigger event and Generic trace external output trigger events can be multicycle input triggers.

CTM support for multicycle channel events is IMPLEMENTATION DEFINED. A CTM that does not support multicycle channel events cannot propagate a multicycle trigger event between CTIs.

**Note**

A full ECT might comprise a mix of CTIs, some of which can support multicycle trigger events. In bridging these components, multicycle channel events become single channel events at the boundary between the CTIs.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### An ECT that supports multicycle trigger events

When an ECT supports multicycle trigger events, an input trigger event to the CTI continuously asserts channel events on all output channels mapped to it until either:

- The input trigger event is removed.
- The channel mapping function is disabled.

This means that an input trigger that is asserted for multiple cycles causes any channels that are mapped to it to become active for multiple cycles. Consequently, any output triggers mapped from that channel are asserted for multiple cycles.

**Note**

The output trigger remains asserted for at least as long as the channel remains active. This means that even if the output trigger is acknowledged, it remains asserted until the channel deactivates.

The CTI does not guarantee that these events have precisely the same duration, as the triggers and channels can cross between clock domains.

CTIAPPSET and CTIAPPCLEAR can set a channel active for multiple cycles. CTIAPPPULSE generates a single channel event. CTICHINSTATUS and CTICHOUTSTATUS can report whether a channel is active.

### An ECT that does not support multicycle trigger events

When an ECT does not support multicycle trigger events, an input trigger event to the CTI generates a single channel event on all output channels mapped to it, regardless of how long the input trigger event is asserted.

This means that an input trigger event that is asserted for multiple cycles generates a single channel event on any channels mapped to it. Consequently any self-acknowledging output triggers mapped from those channels are single trigger events.

**Note**

A single event is typically a single cycle, but there is no guarantee that this is always the case.

CTIAPPSET and CTIAPPCLEAR can only generate a single channel event. CTIAPPPULSE generates a single channel event. If the ECT does not support multicycle channel events, use of CTIAPPSET and CTIAPPCLEAR is deprecated, and the debugger must only use CTIAPPPULSE. CTICHINSTATUS and CTICHOUTSTATUS must be treated as UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H5.3 Cross-triggers on a PE in an ARMv8 implementation

An ARMv8 PE must include a cross-trigger interface, and the implementation must include at least the input and output triggers defined in this architecture. The number of channels in the cross-trigger matrix is IMPLEMENTATION DEFINED, but there must be a minimum of three. Software can read CTIDEVID.NUMCHAN to discover the number of implemented channels.

The CTM must connect to all PEs in the same Inner Shareability domain as the ARMv8-A PE, but can also connect to additional PEs. ARM strongly recommends that the CTM connects all PEs implementing a CTI in the system. This includes ARMv7-A PEs and other PEs that can be connected using a CoreSight CTI module.

**Note**

In a uniprocessor system the CTM is OPTIONAL. The CTM might be connected other CTI modules for non-PEs, such as triggers for system visibility components. ARM recommends that the CTM is implemented.

Any CTI connected to a PE that is not an ARMv8-A PE must implement at least:

- The Debug request trigger event.
- The Restart trigger event.
- The Cross-halt trigger event.

For more information about the CTI, see the *CoreSight ™ SoC Technical Reference Manual*. ARMv8-A refines the generic CTI by defining roles for each of the implemented input and output triggers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H5.4 Description and allocation of CTI triggers

Table H5-1 shows the output trigger events defined by the architecture and the related trigger numbers.

**Table H5-1 Allocation of CTI output trigger events**

| Number | Source | Destination | Event description |
|---|---|---|---|
| 0 | CTI | PE | *Debug request trigger event* on page H5-4916 |
| 1 | CTI | PE | *Restart request trigger event* on page H5-4916 |
| 2 | CTI | IRQ controller | *Generic CTI interrupt trigger event* on page H5-4917 |
| 3 | - | - | Reserved |
| 4 - 7 | CTI | Trace extension | OPTIONAL *Generic trace external input trigger events* on page H5-4917 |

**Note**

Output triggers from the CTI are inputs to other blocks.

Table H5-2 shows the input trigger events defined by the architecture and the related trigger numbers.

**Table H5-2 Allocation of CTI input trigger events**

| Number | Source | Destination | Event description |
|---|---|---|---|
| 0 | PE | CTI | *Cross-halt trigger event* on page H5-4916 |
| 1 | PE | CTI | *Performance Monitors overflow trigger event* on page H5-4917 |
| 2 - 3 | - | - | Reserved |
| 4 - 7 | Trace extension | CTI | OPTIONAL *Generic trace external output trigger events* on page H5-4917 |

**Note**

Input triggers to the CTI are outputs from other blocks.

Table H5-1 and Table H5-2 show the minimum set of trigger events defined by the architecture. However:

- The Generic trace external input and output trigger events are only required if the OPTIONAL Trace extension is implemented. If the OPTIONAL Trace extension is not implemented, these trigger events are reserved.
- Support for the generic CTI interrupt trigger event is IMPLEMENTATION DEFINED because details of interrupt handling in the system, including any interrupt controllers, are IMPLEMENTATION DEFINED. Details regarding how the CTI interrupt is connected to an interrupt controller and its allocated interrupt number lie outside the scope of the architecture. ARM strongly recommends that implementations provide a means to generate interrupts based on external debug events.
- The other trigger events are required by the architecture.

An ARMv8-A implementation can extend the CTI with additional triggers. These start with the number eight.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H5.4.1 Debug request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to force the PE into Debug state. The trigger event is asserted until acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to CTIINTACK[0].

**Note**

A debugger must poll CTITRIGOUTSTATUS[0] until it reads as 0, to confirm that the output trigger has been deasserted before generating any event that must be ordered after the write to CTIINTACK.

If the PE is already in Debug state, the PE ignores the trigger event, but the CTI continues to assert it until it is removed by the debugger. See also *External Debug Request debug event* on page H3-4878.

### H5.4.2 Restart request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to request the PE to exit Debug state. If the PE is in Non-debug state, the request is ignored and dropped by the CTI.

If a Restart request trigger event is received at or about the same time as the PE enters Debug state, it is CONSTRAINED UNPREDICTABLE whether:

- The restart is ignored. In this case the PE enters Debug state and remains in Debug state.
- The PE enters Debug state and then immediately restarts.

Debuggers must program the CTI to send Restart request trigger events only to PEs that are halted. To enable the PE to disambiguate discrete Restart request trigger events, after sending a Restart request trigger event, the debugger must confirm that the PE has restarted and halted before sending another Restart request trigger event. Debuggers can use EDPRSR.{SDR, HALTED} to determine the Execution state of the PE.

**Note**

Before generating a Restart request trigger event for a PE, a debugger must ensure any Debug request trigger event targeting that PE is cleared. *Debug request trigger event* describes how to do this.

The trigger event is self-acknowledging, meaning that the debugger requires no further action to remove the trigger event. See also *Exiting Debug state* on page H2-4858.

### H5.4.3 Cross-halt trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted by a PE when it is entering Debug state.

**Note**

To reduce the latency of halting, ARM recommends that an implementation issues the Cross-halt trigger event early in the committed process of entering Debug state. This means that there is no requirement to wait until all aspects of entry to Debug state have completed before issuing the trigger event. Speculative emission of Cross-halt trigger events is not allowed. The Cross-halt trigger event must not be issued early enough for a subsequent Debug request trigger event, that might be derived from the Cross-halt trigger event, to be recorded in the EDSCR.STATUS field. This applies to Debug request trigger events that are acting as inputs to the PE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H5.4.4 Performance Monitors overflow trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted each time the PE asserts a new Performance Monitors counter overflow interrupt request. See Chapter D5 *The Performance Monitors Extension*.

If the CTI supports multicycle trigger events, then the trigger event remains asserted until the overflow is cleared by a write to PMOVSCLR_EL0. Otherwise, the trigger event is not asserted again until the overflow is cleared by a write to PMOVSCLR_EL0.

**Note**

- This does not replace the recommended connection of Performance Monitors overflow trigger event to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI.
- Events can be counted when `ExternalNoninvasiveDebugEnabled()` == FALSE, and, in Secure state, when `ExternalSecureNoninvasiveDebugEnabled()` == FALSE. Secure software must be aware that overflow trigger events are nevertheless visible to the CTI.

### H5.4.5 Generic trace external input trigger events

These are output trigger events from the CTI, and input trigger events to the OPTIONAL Trace extension, that are used in conjunction with the Generic trace external output trigger events to pass trigger events between:

- The PE and the OPTIONAL Trace extension.
- The OPTIONAL Trace extension and any other component attached to the CTM, including other Trace extensions.

There are four Generic trace external input trigger events.

The trigger events are self-acknowledging. This means that the debugger does not have to take any further action to to remove the events.

### H5.4.6 Generic trace external output trigger events

These are input trigger events to the CTI, and output trigger events from the OPTIONAL Trace extension, used in conjunction with the Generic trace external input trigger events to pass trigger events between:

- The PE and the OPTIONAL Trace extension.
- The OPTIONAL Trace extension and any other component attached to the CTM, including other Trace extensions.

There are four Generic trace external output trigger events.

### H5.4.7 Generic CTI interrupt trigger event

This is an output trigger event from the CTI, and an input to an IMPLEMENTATION DEFINED interrupt controller, and can transfer trigger events from the PE, Trace extension, or any other component attached to the CTI and CTM to software as an interrupt. The Generic CTI interrupt trigger event must be connected to the interrupt controller as an interrupt that can target the originating PE.

**Note**

- ARM recommends that the Generic CTI interrupt trigger event is a private peripheral interrupt, but implementations might instead make this trigger event available as a shared peripheral interrupt or a local peripheral interrupt.
- GICv3 reserves a private peripheral interrupt number for this interrupt.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

It is IMPLEMENTATION DEFINED whether this trigger event is:

- Self-acknowledging. This means that the debugger is not required to take any further action, and that the interrupt controller must treat the trigger event as a pulse or edge-sensitive interrupt.
- Acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to CTIINTACK[2]. This means that the interrupt controller must treat the trigger event as a level-sensitive interrupt.

ARM recommends that the Generic CTI interrupt trigger event is a self-acknowledging trigger event.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H5.5 CTI registers programmers' model

The CTI registers programmers' model is described in Chapter H8 *About the External Debug Registers*. The following sections contain information specific to the CTI:

- *External debug register resets* on page H8-4963.
- *External debug interface register access permissions* on page H8-4952.
- *Cross-trigger interface registers* on page H8-4961.
- The individual register descriptions in *Cross-Trigger Interface registers* on page H9-5054.

See also *Memory-mapped accesses to the external debug interface* on page H8-4950.

### H5.5.1 CTI reset

An External Debug reset resets the CTI. See *External debug register resets* on page H8-4963 for details of CTI register resets. All CTI output triggers and output channels are deasserted on an External Debug reset.

### H5.5.2 CTI authentication

The CTI ignores the state of the IMPLEMENTATION DEFINED authentication interface. This means that:

- CTITRIGINSTATUS shows the status of the input triggers and CTICHINSTATUS shows the status of the input channels, regardless of the value of `ExternalNoninvasiveDebugEnabled()`.

  **Note**

  The PE does not generate the Cross-halt trigger event and the Trace extension does not generate Generic trace external output trigger events when `ExternalNoninvasiveDebugEnabled()` == FALSE. However, the PE can generate Performance Monitors overflow trigger events.

- The CTI can generate external triggers regardless of the value of `ExternalInvasiveDebugEnabled()`.

  **Note**

  The PE ignores Debug request and Restart request trigger events when `ExternalInvasiveDebugEnabled()` == FALSE. The Trace extension ignores Generic trace external input trigger events when `ExternalNoninvasiveDebugEnabled()` == FALSE. The behavior of Generic CTI interrupt requests is part of the IMPLEMENTATION DEFINED handling of these interrupts, but it is permissible for an interrupt controller to receive these requests even when `ExternalInvasiveDebugEnabled()` == FALSE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H5.6 Examples

The CTI is fully programmable and allows for flexible cross-triggering of events within a PE and between PEs in a multiprocessor system. For example:

- The Cross-halt trigger event and the Debug request trigger event can be used for cross-triggering in a multiprocessor system.
- The Cross-halt trigger event and the Generic interrupt trigger event can be used for event-driven debugging in a multiprocessor system.
- The Performance Monitors overflow trigger event and the Debug request trigger event can force entry to Debug state on overflow of a Performance Monitors event counter, for event-driven profiling.

  **Note**

  This does not replace the recommended connection of Performance Monitors overflow trigger events to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI. ARM recommends that the Performance Monitors overflow signal is directly available as a local interrupt source.

- The Generic trace external input and Generic trace external output trigger events can pass trace events into and out of the event logic of the Trace extension. They can do this:
  - To pass trace events between Trace extensions.
  - In conjunction with the Performance Monitors overflow trigger event, to couple the Performance Monitors to the Trace extension.
  - In conjunction with the Debug request trigger event, to trigger entry to Debug state on a trace event.
  - In conjunction with other CTIs, to signal a trace trigger event onto a CoreSight trace interconnect.

The following sections describe some examples in more detail:

- *Halting a single PE*.
- *Halting all PEs in a group when any one PE halts* on page H5-4921.
- *Synchronously restarting a group of PEs* on page H5-4921.
- *Halting a single PE on Performance Monitors overflow* on page H5-4922.

**Example H5-1 Halting a single PE**

To halt a single PE, set:

1. CTIGATE[0] to 0, so that the CTI does not pass channel events on internal channel 0 to the CTM.
2. CTIOUTEN0[0] to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 0.

   **Note**

   The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instance of CTIOUTEN<n> for which <n> is 0.

3. CTIAPPPULSE[0] to 1, to generate a channel event on channel 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to CTIINTACK[0], before restarting the PE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Example H5-2 Halting all PEs in a group when any one PE halts**

To program a group of PEs so that when one PE in the group halts, all of the PEs in that group halt, set the following registers for each PE in the group:

1. CTIGATE[2] to 1, so that each CTI passes channel events on internal channel 2 to the CTM.
2. CTIINEN0[2] to 1,so that each CTI generates a channel event on channel 2 in response to a Cross-halt trigger event.
3. CTIOUTEN0[2] to 1, so that each CTI generates a Debug request trigger event in response to an channel event on channel 2.

   **Note**

   The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instances of CTIINEN<n> and CTIOUTEN<n> for which <n> is 0.

When a PE has halted, clear the Debug request trigger event by writing a value of 1to CTIINTACK[0], before restarting the PE.

**Example H5-3 Synchronously restarting a group of PEs**

To restart a group of PEs, for each PE in the group:

1. If the PE was halted because of a Debug request trigger event, the debugger must ensure the trigger event is deasserted. It can do this by:
   a. Writing 1 to CTIINTACK[0] to clear the Debug request trigger event.
   b. Polling CTITRIGOUTSTATUS[0], until it reads as 0, to confirm that the trigger event has been deasserted.
2. Set CTIGATE[1] to 1, so that each CTI passes channel events on internal channel 1 to the CTM.
3. Set CTIOUTEN1[1] to 1, so that each CTI generates a Restart request trigger event in response to a channel event on channel 1.

   **Note**

   This example must use the instance of CTIOUTEN<n> for which <n> is 1.

4. Set CTIAPPPULSE[1] to 1 on any one PE in the group, to generate a channel event on channel 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Example H5-4 Halting a single PE on Performance Monitors overflow**

To halt a single PE on a Performance Monitors overflow set:

1. CTIGATE[3] to 0, so that the CTI does not pass channel events on internal channel 3 to the CTM.
2. CTIINEN1[3] to 1, so that the CTI generates a channel event on channel 3 in response to a Performance Monitors overflow trigger event.

   **Note**

   This step of this example must use the instance of CTIINEN<n> for which <n> is 1.

3. CTIOUTEN0[3] to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 3.

   **Note**

   This step of this example must use the instance of CTIOUTEN<n> for which <n> is 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to CTIINTACK[0], before restarting the PE. Clear the overflow status by writing to PMOVSCLR_EL0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter H6
# Debug Reset and Powerdown Support

This chapter describes the reset and powerdown support in the Debug architecture. It contains the following sections:


- *About Debug over powerdown* on page H6-4924.
- *Power domains and debug* on page H6-4925.
- *Core power domain power states* on page H6-4926.
- *Emulating low-power states* on page H6-4928.
- *Debug OS Save and Restore sequences* on page H6-4930.
- *Reset and debug* on page H6-4935.


**Note**

Where necessary, Table K13-1 on page K13-5708 disambiguates the general register references used in this chapter.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H6.1 About Debug over powerdown

Debug over powerdown is a facility for an operating system to save and restore the PE state on behalf of a self-hosted or external debugger or both.

For external debug over powerdown, the architecture defines the OS Lock, OS Double Lock, and the logical split of the hardware on which a PE executes into the *Core power domain* and the *Debug power domain*. See:

- *Power domains and debug* on page H6-4925.
- *Core power domain power states* on page H6-4926.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H6.2 Power domains and debug

The external debug component of ARMv8-A has two logical power domains, each with its own reset:

- The debug power domain contains the interface between the PE and the external debugger, and is powered up whenever an external debugger is connected to the SoC. It remains powered up while the external debugger is connected. Registers in this domain are reset by an external debug reset.
- The core power domain contains the rest of the PE, and is allowed to power up and power down independently of the Debug power domain.

The core power domain contains several types of registers:

- Non-debug logic refers to all registers and logic that are not associated with debug.
- Self-hosted debug logic refers to registers and logic associated solely with the self-hosted debug aspects of the architecture.
- Shared debug logic refers to registers and logic associated with both the self-hosted and external debug aspects of the architecture.
- External debug logic refers to registers and logic associated solely with the external debug aspects of the architecture.

**Note**

- The model of two logical power domains has an impact on the reset and access permission requirements of the debug programmers' model.
- The power domains are described as logical because the architecture defines the requirements but does not require two physical power domains. Any power domain split that meets the requirements of the programmers' model is a valid implementation.

Figure H6-1 shows the recommended power domain split. The signals **DBGPWRUPREQ**, **DBGNOPWRDWN**, and **DBGPWRDUP** shown in Figure H6-1 provide an interface between the power controller and the PE debug logic that is in the debug power domain. They are part of the recommended interface. See Appendix K2 *Recommended External Debug Interface*.

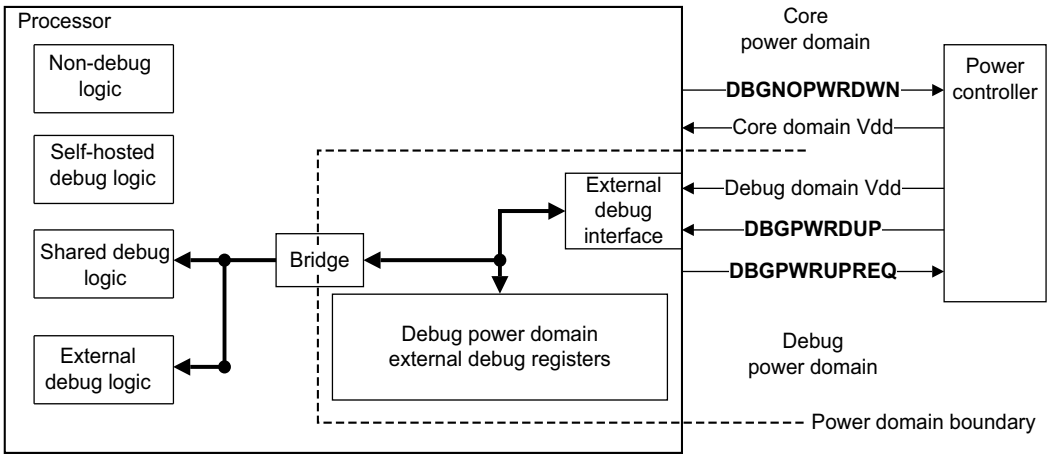



**Figure H6-1 Recommended power domain split between core and debug power domains**


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H6.3 Core power domain power states

The ARM architecture does not define the power states of the PE as these are not normally visible to software. However, they are visible to the external debugger. The Debug architecture uses a four logical power states model for the core power domain. The four logical power states are as follows:

**Normal** The core power domain is fully powered up and the debug registers are accessible.

**Standby** The core power domain is on, but there are measures to reduce energy consumption. In a typical implementation, the PE enters standby by executing a `WFI` or `WFE` instruction, and exits on a wake-up event. There can be other IMPLEMENTATION DEFINED measures the OS can take to enter standby.

The PE preserves the PE state, including the debug logic state. Changing from standby to normal operation does not involve a reset of the PE.

Standby is the least invasive OS energy saving state. Standby implies only that the PE is unavailable and does not clear any debug settings. For standby, the Debug architecture requires only the following:

- An External Debug Request debug event is a wake-up event when halting is allowed. This means that the PE must exit standby to handle the debug event. If the PE executed a `WFE` or a `WFI` instruction to enter standby, then it retires that instruction,
- If the external debug interface is accessed, the PE must respond to that access. ARM recommends that, if the PE executed a `WFI` or `WFE` instruction to enter standby, then it does not retire that instruction.

Standby is transparent, meaning that to software and to an external debugger it is indistinguishable from normal operation.

**Retention** The OS takes some measures, including IMPLEMENTATION DEFINED code sequences and registers, to reduce energy consumption. The PE state, including debug settings, is preserved in low-power structures, allowing the core power domain to be at least partially turned off.

Changing from low-power retention to normal operation does not involve a reset of the PE. The saved PE state is restored on changing from low-power retention state to normal operation. If software has to use an IMPLEMENTATION DEFINED code sequence before entering, or after leaving, a retention state, this is referred to as a *software-visible retention state*. It is IMPLEMENTATION DEFINED whether the value of DBGPRCR.CORENPDRQ is set to the value of EDPRCR.COREPURQ on leaving the software-visible retention state.

External Debug Request debug events stay pending and debug registers in the core power domain cannot be accessed.

**Note**

- This model of retention does not include implementations where the PE exits the state in response to a debug register access. From the Debug architecture perspective, implementations like this are forms of standby.

**Powerdown** The OS takes some measures to reduce energy consumption by turning the core power domain off. These measures must include the OS saving any PE state, including the debug settings, that must be preserved over powerdown. Changing from powerdown to normal operation must include:

- A Cold reset of the PE after the power level has been restored.
- The OS restoring the saved PE state.

External Debug Request debug events stay pending and debug registers in the core power domain cannot be accessed.

The Debug architecture uses a simpler two states model for the Debug power domain. The two states are:

**Off** The debug power domain is turned off.

**On** The debug power domain is turned on.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The available power states, including the cross-product of core power domain and debug power domain power states is IMPLEMENTATION DEFINED. Implementations are not required to implement all of these states and might include additional states. These additional states must appear to the debugger as one of the logical power states defined by this model. The control of power states is IMPLEMENTATION DEFINED.

**Note**

As a result, it is IMPLEMENTATION DEFINED whether it is possible for the debug power domain to be on when the core power domain is off.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H6.4 Emulating low-power states

The control registers DBGPRCR.CORENPDRQ and EDPRCR.COREPURQ provide an interface between the power controller and the PE. They typically map directly to signals in the recommended external debug interface. With this interface the external debugger can request the power controller to:

- Emulate states where the core power domain is completely off or in a low-power state where the core power domain registers cannot be accessed. This simplifies the requirements on software by sacrificing entirely realistic behavior.
- Restore full power to the core power domain.

EDPRSR.{SPD, PU} indicates the core power domain power state. For more information see:

- *DBGPRCR_EL1, Debug Power Control Register* on page D7-2185 and *DBGPRCR, Debug Power Control Register* on page G6-4704.
- *EDPRCR, External Debug Power/Reset Control Register* on page H9-5030.
- *EDPRSR, External Debug Processor Status Register* on page H9-5033.
- Appendix K2 *Recommended External Debug Interface*.

The measures to emulate powerdown are IMPLEMENTATION DEFINED. The ability of the debugger to access the state of the PE and the system might be limited as a result of the measures adopted.

In an emulated powerdown state, the debugger must be able to access all debug registers in both the debug power domain and the core power domain as if the core power domain were on. That is, the debugger must be able to read and write to such registers without receiving errors. This allows an external debugger to debug the powerup sequence. To stop the OS Double Lock preventing access to debug registers when powerdown is being emulated, `DoubleLockStatus() == FALSE` when DBGPRCR.CORENPDRQ == 1.

Otherwise, the behavior of the PE in emulated powerdown must be similar to that in a real powerdown state. In particular, the PE must not respond to other system stimuli, such as interrupts.

Example H6-1 and Example H6-2 are examples of two approaches to emulating powerdown.

**Example H6-1 An example of emulating powerdown**

---

The PE is held in Standby state, isolated from any system stimuli. It is IMPLEMENTATION DEFINED whether the PE can respond to debug stimuli such as an External Debug Request debug event.

If the PE can enter Debug state, then the external debugger is able to use the ITR to execute instructions, such as loads and stores. This causes the external debugger to interact with the system. If the external debugger restarts the PE, the PE leaves Standby state and restarts fetching instructions from memory.

---

**Example H6-2 Another example of emulating powerdown**

---

The PE is held in Warm reset. This limits the ability of an external debugger to access the resources of the PE. For example, the PE cannot be put into Debug state.

---

On exit from emulated powerdown the PE is reset. However, the debug registers that are only reset by a Cold reset must not be reset. Typically this means that a Warm reset is substituted for the Cold reset.

**Note**

- Warm reset and Cold reset have different effects apart from resetting the debug registers. In particular, RMR_ELx is reset by a Cold reset and controls the reset state on a Warm reset. This means that if a Cold reset is substituted by a Warm reset, the behavior of the reset code might be different.
- The timing effects of powering down are typically not factored in the powerdown emulation. Examples of these timing effects are clock and voltage stabilization.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Emulation does not model the state lost during powerdown, meaning that it might mask errors in the state storage and recovery routines.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H6.5 Debug OS Save and Restore sequences

In ARMv8-A, the following registers provide the OS Save and restore mechanism:

- The *OS Lock Access Register*, OSLAR, locks the OS Lock to restrict access to debug registers before starting an OS Save sequence, and unlocks the OS Lock after an OS Restore sequence.
- The *OS Lock Status Register*, OSLSR, shows the status of the OS Lock.
- The *External Debug Execution Control Register*, EDECR, can be configured to generate a debug event when the OS Lock is unlocked.
- The *OS Double Lock Register*, OSDLR, locks out an external debug interface entirely. This is only used immediately before a powerdown sequence.

See also:

- *Reset and debug* on page H6-4935
- Appendix K8 *Example OS Save and Restore Sequences*

### H6.5.1 Debug registers to save over powerdown

Table H6-1 shows the different requirements for self-hosted debug over powerdown and external debug over powerdown:

- The column labeled Self-hosted lists registers that software must preserve over powerdown so that it can support self-hosted debug over powerdown. This does not require use of the OS Save and Restore mechanism.
- The column labeled External lists registers that software must preserve over powerdown so that it can support external debug over powerdown. This requires use of the OS Save and Restore mechanism:
  - Some external debug registers are not normally accessible to software executing on the PE. Additional debug registers are provided that give software the required access to save and restore these external debug registers when OSLSR.OSLK is locked. These registers include OSECCR, OSDTRRX, and OSDTRTX.
- Some registers might only present in some implementations, or might not be accessible at all Exception levels or in Non-secure state. DBGVCR32_EL2 and SDER32_EL3 are only required to support AArch32.

Table H6-1 does not include registers for the OPTIONAL Trace and Performance Monitor extensions.

**Table H6-1 Debug registers to save over powerdown**

| Register in AArch64 | Register in AArch32 | Self-hosted | External |
|---|---|---|---|
| MDSCR_EL1 | DBGDSCRext | Yes | Yes[a] |
| DBGBVR<n>_EL1 | DBGBVR<n> | Yes | Yes |
| DBGBCR<n>_EL1 | DBGBCR<n> | Yes | Yes |
| DBGWVR<n>_EL1 | DBGWVR<n> | Yes | Yes |
| DBGWCR<n>_EL1 | DBGWCR<n> | Yes | Yes |
| DBGVCR32_EL2 | DBGVCR | Yes | - |
| MDCR_EL2 | HDCR | Yes | - |
| SDER32_EL3 | SDER | Yes | - |
| MDCR_EL3 | SDCR | Yes[b] | - |
| MDCCINT_EL1 | DBGDCCINT | - | Yes[b] |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table H6-1 Debug registers to save over powerdown (continued)**

| Register in AArch64 | Register in AArch32 | Self-hosted | External |
|---|---|---|---|
| DBGCLAIMSET_EL1<br>DBGCLAIMCLR_EL1 | DBGCLAIMSET,<br>DBGCLAIMCLR | - | Yes[c] |
| OSECCR_EL1 | DBGOSECCR | - | Yes[ab] |
| OSDTRRX_EL1<br>OSDTRTX_EL1 | DBGDTRRXext<br>DBGDTRTXext | - | Yes |

a. The OS Lock must be locked to save and restore for external debug. When the OS Lock is locked, DSCR is part of the software save and restore mechanism for external debug. It provides a mechanism for an operating system to access some fields of EDSCR that are otherwise read-only or not visible to software. This allows the operating system to save and restore these settings over a powerdown for the external debugger.

b. This register is new in ARMv8-A. Sequences written for ARMv7 do not preserve the register over powerdown.

c. Read DBGCLAIMCLR to save, write DBGCLAIMSET to restore.

### H6.5.2 OS Save sequence

To preserve the debug logic state over a powerdown, the state must be saved to nonvolatile storage. This means the OS Save sequence must:

1. Lock the OS Lock by:
   - Writing the key value `0xC5ACCE55` to the DBGOSLAR in AArch32 state.
   - Writing 1 to OSLAR_EL1.OSLK in AArch64 state.
2. Execute an `ISB` instruction.
3. Walk through the debug registers listed in *Debug registers to save over powerdown* on page H6-4930 and save the values to the nonvolatile storage.

Before removing power from the core power domain, software must:

1. Lock the OS Double Lock by writing 1 to OSDLR_EL1.DLK.
2. Execute a context synchronization operation.

### H6.5.3 OS Restore sequence

After a powerdown, the OS Restore sequence must perform the following steps to restore the debug logic state from the non-volatile storage:

1. Lock the OS Lock, as described in *OS Save sequence*. The OS Lock is generally locked by the Cold reset, but this step ensures that it is locked.
2. Execute an `ISB` instruction.
3. To ensure that, if an external debugger clears the OS Lock before the end of this sequence, no debug exceptions are generated:
   - Write 0 to MDSCR_EL1 if executing in AArch64 state.
   - Write 0 to DBGDSCRext if executing in AArch32 state.
4. Walk through the debug registers listed in *Debug registers to save over powerdown* on page H6-4930, and restore the values from the nonvolatile storage. The last register to be restored must be:
   - MDSCR_EL1 if executing in AArch64 state.
   - DBGDSCRext if executing in AArch32 state.
5. Execute an `ISB` instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

6. Unlock the OS Lock by:
   - Writing any non-key value to DBGOSLAR if executing in AArch32 state.
   - Writing 0 to OSLAR_EL1.OSLK if executing in AArch64 state.
7. Execute a context synchronization operation.

**Note**

The OS Restore sequence overwrites the debug registers with the values that were saved. If there are valid values in these registers immediately before the restore sequence, then those values are lost.

### H6.5.4 Debug behavior when the OS Lock is locked

The main purpose of the OS Lock is to prevent updates to debug registers during an OS Save or OS Restore operation. The OS Lock is locked on a Cold reset.

When the OS Lock is locked:

- Access to debug registers through the System register interface is mainly unchanged except that:
  - Certain registers are read and written without side-effects.
  - Fields in DSCR and OSECCR that are normally read-only become read/write.

  This allows the state to be saved or restored. For more information, see the relevant register description in Chapter H9 *External Debug Register Descriptions*.
- Access to debug registers by the external debug interface is restricted to prevent an external debugger modifying the registers that are being saved or restored. For more information see *External debug interface register access permissions summary* on page H8-4954.
- Debug exceptions, other than Software Breakpoint Instruction exceptions are not generated.

The OS Lock has no effect on Software Breakpoint Instruction debug events and Halting debug events.

### H6.5.5 Debug behavior when the OS Lock is unlocked

When the OS Lock is unlocked, an OS Unlock Catch debug event is generated if EDECR.OUCE is set to 1. See *OS Unlock Catch debug event* on page H3-4879.

### H6.5.6 Debug behavior when the OS Double Lock is locked

The OS Double Lock is locked immediately before a powerdown sequence. The OS Double Lock ensures that it is safe to remove core power by forcing the debug interfaces to be quiescent.

When `DoubleLockStatus()` == TRUE:

- The external debug interface only has restricted access to the debug registers, so that it is quiescent before removing power. See *External debug interface register access permissions summary* on page H8-4954.
- Debug exceptions, other than Software Breakpoint Instruction exceptions, are not generated.
- Halting is prohibited. See *Halting allowed and halting prohibited* on page H2-4821.

  **Note**

  Pending Halting debug events might be lost when core power is removed.
- No asynchronous debug events are `WFI` or `WFE` wake-up events.

Software must synchronize the update to OSDLR before it indicates to the system that core power can be removed. The interface between the PE and its power controller is IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Typically software indicates that core power can be removed by entering the Wait For Interrupt state. This means that software must explicitly synchronize the OSDLR update before issuing the `WFI` instruction.

OSDLR.DLK is ignored and `DoubleLockStatus()` == FALSE if either:

- The PE is in Debug state.
- DBGPRCR.CORENPDRQ is set to 1.

**Note**

It is possible to enter Debug state with OSDLR.DLK set to 1. This is because a context synchronization operation is required to ensure the OS Double Lock is locked, meaning that Debug state might be entered before the OSDLR update is synchronized.

Because OSDLR.DLK is ignored when DBGPRCR.CORENPDRQ is set to 1, and an external debugger can write to DBGPRCR.CORENPDRQ, software must not rely on using the OS Double Lock to disable debug exceptions or to prohibit halting, or both. ARM deprecates use of the OS Double Lock for these purposes, and instead recommends that software:

- Uses the OS Lock to disable debug exceptions during save or restore sequences.
- Uses the debug authentication interface to prohibit halting and external debug access to debug registers at times other than immediately prior to removing power.

As the purpose of the OS Double Lock is to ensure that it is safe to remove core power, it is important to avoid race conditions that defeat this purpose. ARM recommends that:

- Once the write to OSDLR.DLK has been synchronized by a *Context synchronization operation* and `DoubleLockStatus()` == TRUE, a PE must:
  - Not allow a debug event generated before the *Context synchronization operation* to cause an entry to Debug state or act as a wake-up event for a `WFI` or `WFE` instruction after the context synchronization operation has completed.
  - Complete any external debug access started before the *Context synchronization operation* by the time the context synchronization operation completes.

    **Note**

    A debug register access might be in progress when software sets OSDLR.DLK to 1. An implementation must not permit the synchronization of locking the OS Double Lock to stall indefinitely while waiting for that access to complete. This means that any debug register access that is in progress when software sets OSDLR.DLK to 1 must complete or return an error in finite time.

- If a write to DBGPRCR or EDPRCR made when OSDLR.DLK == 1 changes DBGPRCR.CORENPDRQ or EDPRCR.CORENPDRQ from 1 to 0, meaning `DoubleLockStatus()` changes from FALSE to TRUE, then before signaling to the system that the CORENPDRQ field has been cleared and emulation of powerdown is no longer requested, meaning the system can remove core power, the PE must ensure that all the requirements for `DoubleLockStatus()` == TRUE listed in this section are met.

In the standard OS Save sequence, the OS Lock is locked before the OS Double Lock is locked. This means that writes to CORENPDRQ are ignored by the time the OS Double Lock is locked. However, if `DoubleLockStatus()` == FALSE, an external debugger can clear the OS Lock at any time, and then write to EDPRCR.

The pseudocode for the `DoubleLockStatus()` function is as follows:

```
// DoubleLockStatus()
// ==================
// Returns the value of EDPRSR.DLK.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()

    if ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H6.6 Reset and debug

All registers in the Core power domain are reset either by a Warm reset or a Cold reset, as described in *Reset* on page D1-1533, including external debug logic registers.

All registers in the Debug power domain are reset by an External Debug reset.

Figure H6-2 shows this reset scheme. The following three reset signals are an example implementation of the reset scheme:

- **CORERESET**, which must be asserted for a Warm reset.
- **CPUPORESET**, which must be asserted for a Cold reset.
- **PRESETDBG**, which must be asserted for an External Debug reset.

As shown in the figure, the external debug logic is split between the Debug power domain and the Core power domain.

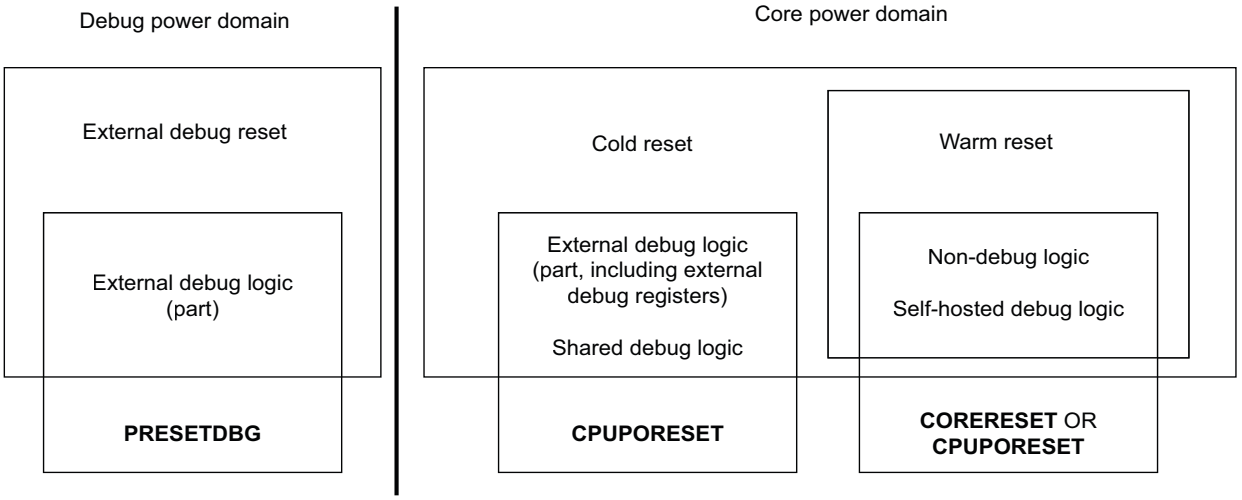



**Figure H6-2 Power and reset domains**

For more information about power domains and power states, see *Power domains and debug* on page H6-4925.

When power is first applied to the Debug power domain, **PRESETDBG** must be asserted.

When power is first applied to the Core power domain, **CPUPORESET** must be asserted.

**Note**

In this scheme, logic in the Warm reset domain is reset by asserting either **CORERESET** or **CPUPORESET**. This implies a particular implementation style that permits these approaches.

**CPUPORESET** is not normally asserted on moving from a low-power state, where power has not been removed, to a full-power state. This can occur, for example, on exiting a low-power retention state. See also *Emulating low-power states* on page H6-4928 and *EDPRSR, External Debug Processor Status Register* on page H9-5033.

### H6.6.1 External debug interface accesses to registers in reset

If a reset signal is asserted and the external debug interface:

- Writes a register, or indirectly writes a register or register field as a side-effect of an access:
  - Then, if the register or register field is reset by that reset signal, it is CONSTRAINED UNPREDICTABLE whether the register or register field takes the reset value or the value written. The reset value might be UNKNOWN.
  - Otherwise the register or register field takes the value that is written.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Reads a register, or indirectly reads a register or register field, as part of an access:
  - Then, if the register or register field is reset by that reset signal, the value returned in UNKNOWN.
  - Otherwise, the value of the register or register field is returned.

It is IMPLEMENTATION DEFINED whether any register can be accessed when External Debug reset is being asserted. The result of these accesses is IMPLEMENTATION DEFINED.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Chapter H7
# The Sample-based Profiling Extension

This chapter describes the Sample-based Profiling Extension, that is an OPTIONAL extension to the ARMv8 architecture. The extension provides a non-invasive external debug component.

**Note**

This form of the Sample-based Profiling Extension is OPTIONAL. ARM recommends that if extension EDPCSR is not implemented that an alternative IMPLEMENTATION DEFINED form of Sample-based Profiling is implemented.

It contains the following section:


- *Sample-based profiling* on page H7-4938.




*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H7.1 Sample-based profiling

Sample-based Profiling is an OPTIONAL extension to the architecture. It provides a mechanism for coarse-grained profiling of software executing on the PE by an external debugger, without changing the behavior of that software. The following sections describe this extension:

- *The implemented Sample-based profiling registers*.
- *Reads of the External Debug Program Counter Sampling Registers*.
- *Reads of the External Debug Virtual Context Sample Register* on page H7-4939.
- *Accuracy of sampling* on page H7-4939.
- *Sample-based Profiling and security* on page H7-4940.
- *Pseudocode description of Sample-based Profiling* on page H7-4940.

### H7.1.1 The implemented Sample-based profiling registers

An implementation that includes the Sample-based Profiling Extension implements the following external debug registers:

- EDPCSR is a 64-bit read-only register that contains a sampled program counter value. As external debug register accesses are atomic only at word granularity, EDPCSR is split into two registers: EDPCSRhi and EDPCSRlo. See *Reads of the External Debug Program Counter Sampling Registers*.
- EDCIDSR is a read-only register that contains the sampled value of CONTEXTIDR_EL1 indirectly read on reading EDPCSRlo.

  **Note**

  If EL3 is implemented and using AArch32, then CONTEXTIDR is a Banked register and EDCIDSR samples the current Banked copy of CONTEXTIDR.

- EDVIDSR is a read-only register that contains sampled values captured on reading EDPSRlo. If neither EL3 nor EL2 are implemented, EDVIDSR is not implemented.

### H7.1.2 Reads of the External Debug Program Counter Sampling Registers

A read of the EDPCSRlo normally has the side-effect of indirectly writing to EDCIDSR, EDVIDSR, and EDPCSRhi. When EDPCSRlo is read, the bottom 32 bits of a program counter sample are returned. The top 32 bits are captured in EDPCSRhi and can be read later. However:

- If the PE is in Debug state, or Sample-based Profiling is prohibited, EDPCSRlo reads as `0xFFFFFFFF` and EDCIDSR, EDVIDSR and EDPCSRhi become UNKNOWN. See *Sample-based Profiling and security* on page H7-4940.
- If the PE is in Reset state, the sampled value is UNKNOWN and EDCIDSR, EDVIDSR and EDPCSRhi become UNKNOWN.
- If no instruction has been retired since the PE left Reset state, Debug state, or a state where Sample-based Profiling is prohibited, the sampled value is UNKNOWN and EDCIDSR, EDVIDSR and EDPCSRhi become UNKNOWN.
- The indirect writes to EDCIDSR, EDVIDSR, and EDPCSRhi might not occur for a memory-mapped access to the external debug interface. For more information, see *Memory-mapped accesses to the external debug interface* on page H8-4950.

The value written to EDCIDSR is an indirect read of CONTEXTIDR_EL1. This means that it is CONSTRAINED UNPREDICTABLE whether EDCIDSR is set to the original or new value if a read of the EDPCSRlo samples:

- An instruction that writes to CONTEXTIDR_EL1.
- The next context synchronization operation.
- Any instruction executed between these two instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

In ARMv7 Sample-based Profiling Extension, an offset was applied to the sampled program counter value and this offset and the instruction set state indicated in bits [1:0] of the sampled value. In the ARMv8 Sample-based Profiling Extension, the sampled value is the address of an instruction that has executed, with no offset and no indication of the instruction set state.

### H7.1.3 Reads of the External Debug Virtual Context Sample Register

A read of the EDVIDSR contains sampled values captured on reading EDPSRlo, where:

- EDVIDSR.NS indicates the Security state associated with the most recent EDPCSR sample.
- EDVIDSR.E2 indicates whether the most recent EDPCSR sample was associated with EL2. If EDVIDSR.NS == 0, this bit is 0.
- EDVIDSR.E3 indicates whether the most recent EDPCSR sample was associated with EL3 using AArch64. If EDVIDSR.NS == 1 or the PE was in AArch32 state when EDPCSRlo was read, this bit is 0.
- EDVIDSR.HV indicates whether EDPCSRhi is valid, that is, bits [63:32] of the most recent program counter sample are nonzero.

  **Note**

  — EDVIDSR.HV == 1 does not mean that EDPCSRhi != 0. EDVIDSR.HV == 0 is a hint that EDPCSRhi does not need to be read.

  — Tools must take care to avoid skewing sampled data by over-sampling code for which EDVIDSR.HV == 0.

- EDVIDSR.VMID indicates the value of the VTTBR_EL2.VMID register associated with the most recent EDPCSRlo sample. If EDVIDSR.NS == 0 or EDVIDSR.E2 == 1, this field is RAZ.

If EL2 is not implemented, EDVIDSR.E2 and EDVIDSR.VMID are RES0.

If EL3 is not implemented, EDVIDSR.E3 is RES0, and EDVIDSR.NS has a fixed read-only value.

### H7.1.4 Accuracy of sampling

Sample-based Profiling is provided as a mechanism for tools to populate a statistical model of the performance of software executing on the PE. The statistical data returned by random sampling of EDPCSR, EDCIDSR, and EDVIDSR must allow such statistical modeling.

It must be possible to sample references to branch targets. It is IMPLEMENTATION DEFINED whether references to other instructions can be sampled. The branch target for a conditional branch instruction that fails its condition check is the instruction that follows the conditional branch instruction. The branch target for an exception is the exception vector address.

Under normal operating conditions, the whole sample, EDPCSR, EDCIDSR, and EDVIDSR, must reference an instruction that was committed for execution, including its context, and must not reference instructions that are fetched but not committed for execution.

To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the sampled data is acceptable. ARM does not define *a reasonable degree of inaccuracy* but recommends the following guidelines:

- In exceptional circumstances, such as a change in Security state or other boundary condition, it is acceptable for the sample to represent an instruction that was not committed for execution.
- Under very unusual non-repeating pathological cases the sample can represent an instruction that was not committed for execution. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in sampling is very unlikely.

See also *Non-invasive behavior* on page D5-1859.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H7.1.5 Sample-based Profiling and security

Sample-based Profiling is a non-invasive external debug component, which is controlled by an IMPLEMENTATION DEFINED authentication interface. Sample-based Profiling is prohibited unless both:

- Allowed by the IMPLEMENTATION DEFINED authentication interface `ExternalNoninvasiveDebugEnabled()`
- Any one of:
  - Executing in Non-secure state.
  - EL3 is not implemented.
  - EL3 is implemented, executing in Secure state, and allowed by the IMPLEMENTATION DEFINED authentication interface `ExternalSecureNoninvasiveDebugEnabled()`.
  - EL3 is implemented, EL3 or EL1 is using AArch32, executing at EL0 in Secure state, and SDER32_EL3.SUNIDEN == 1.

The state of IMPLEMENTATION DEFINED authentication interface is visible through DBGAUTHSTATUS_EL1.

See also Appendix K2 *Recommended External Debug Interface*.

### H7.1.6 Pseudocode description of Sample-based Profiling

`PCSample()` records a PC sample for the EDPCSR and associated registers.

```
type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    bit ns,
    bits(32) contextidr,
    bits(8) vmid
)

PCSample pc_sample;
// CreatePCSample()
// ================

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.
    enabled = (if IsSecure() then ExternalSecureNoninvasiveDebugEnabled()
                             else ExternalNoninvasiveDebugEnabled());

    pc_sample.valid = enabled && !Halted();
    pc_sample.pc = ThisInstrAddr();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ns = if IsSecure() then '0' else '1';
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1;
    if HaveEL(EL2) && !IsSecure() then
        pc_sample.vmid = if ELUsingAArch32(EL2) then VTTBR.VMID else VTTBR_EL2.VMID;

    return;

// EDPCSRlo[] (read)
// =================

bits(32) EDPCSRlo[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
update = !memory_mapped || EDLSR.SLK == '0';        // Software locked: no side-effects

if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
        EDCIDSR = pc_sample.contextidr;
        EDVIDSR.VMID = (if HaveEL(EL2) && pc_sample.ns == '1' && pc_sample.el IN {EL1,EL0}
                        then pc_sample.vmid else Zeros(8));
        EDVIDSR.NS = pc_sample.ns;
        EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
        EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
        // The conditions for setting HV are not specified if PCSRhi is zero.
        // An example implementation may be "pc_sample.rw".
        EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
else
    sample = Ones(32);
    if update then
        EDPCSRhi = bits(32) UNKNOWN;
        EDCIDSR = bits(32) UNKNOWN;
        EDVIDSR = (bits(4) UNKNOWN):Zeros(20):(bits(8) UNKNOWN);

return sample;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Chapter H8
# About the External Debug Registers

This chapter provides some additional information about the external debug registers. It contains the following sections:


- *Relationship between external debug and System registers* on page H8-4944.
- *Supported access sizes* on page H8-4945.
- *Synchronization of changes to the external debug registers* on page H8-4946.
- *Memory-mapped accesses to the external debug interface* on page H8-4950.
- *External debug interface register access permissions* on page H8-4952.
- *External debug interface registers* on page H8-4956.
- *Cross-trigger interface registers* on page H8-4961.
- *External debug register resets* on page H8-4963.


**Note**

Where necessary Table K13-1 on page K13-5708 disambiguates the general register references used in this chapter.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H8.1 Relationship between external debug and System registers

Table H8-1 shows the relationship between external debug registers and System registers. Where no relationship exists, the registers are not listed.

**Table H8-1 Equivalence between external debug and System registers**

| | System register | | |
|---|---|---|---|
| **External debug register** | **AArch64** | **AArch32** | **Notes** |
| DBGDTRRX_EL0 | DBGDTRRX_EL0 | DBGDTRRXint | See also *Summary of System register accesses to the DCC* on page H4-4897 |
| DBGDTRTX_EL0 | DBGDTRTX_EL0 | DBGDTRTXint | |
| OSLAR_EL1 | OSLAR_EL1 | DBGOSLAR | - |
| DBGBVR<n>_EL1[31:0]<br>DBGBVR<n>_EL1[63:32] | DBGBVR<n>_EL1[31:0]<br>DBGBVR<n>_EL1[63:32] | DBGBVR<n><br>DBGBXVR<n> | - |
| DBGBCR<n>_EL1 | DBGBCR<n>_EL1 | DBGBCR<n> | - |
| DBGWVR<n>_EL1[31:0]<br>DBGWVR<n>_EL1[63:32] | DBGWVR<n>_EL1[31:0]<br>DBGWVR<n>_EL1[63:32] | DBGWVR<n> | - |
| DBGWCR<n>_EL1 | DBGWCR<n>_EL1 | DBGWCR<n> | - |
| DBGCLAIMSET_EL1 | DBGCLAIMSET_EL1 | DBGCLAIMSET | - |
| DBGCLAIMCLR_EL1 | DBGCLAIMCLR_EL1 | DBGCLAIMCLR | - |
| DBGAUTHSTATUS_EL1 | DBGAUTHSTATUS_EL1 | DBGAUTHSTATUS | Read-only |
| EDSCR | MDSCR_EL1 | DBGDSCRext | Only some fields map |
| EDECCR | OSECCR_EL1 | DBGOSECCR | Applies when the OS Lock is locked. |
| MIDR_EL1 | MIDR_EL1 | MIDR | Read-only copies of Processor ID Registers |
| EDDEVAFF0<br>EDDEVAFF1 | MPIDR_EL1[31:0][a]<br>MPIDR_EL1[63:32][a] | MPIDR | Read-only copies of system ID registers |

a. This is a word of a 64-bit register.

In addition:

- EDSCR.{TXfull, RXfull} are read-only aliases for DCCSR.{TXfull, RXfull}.
- EDPRCR.CORENPDRQ is a read/write alias for DBGPRCR.CORENPDRQ.
- EDPRSR.OSLK is a read-only alias for OSLSR.OSLK.
- EDPRSR.DLK is a read-only function of OSLSR.DLK.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H8.2 Supported access sizes

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the debug registers, Performance Monitor registers, and CTI registers, implementations must support:

- Word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations.
- Doubleword-aligned 64-bit accesses to access 64-bit registers mapped to a doubleword-aligned pair of adjacent 32-bit locations.

**Note**

This means that a system implementing the debug registers using a 32-bit bus, such as a AMBA APB3, with a wider system interconnect must implement a bridge between the system and the debug bus that can split 64-bit accesses.

All registers are only single-copy atomic at word granularity. This means that for 64-bit accesses to a 64-bit register, the system might generate a pair of 32-bit accesses. The order in which the two halves are accessed is not specified.

The following accesses are not supported:

- Byte.
- Halfword.
- Unaligned word. These accesses are not word single-copy atomic.
- Unaligned doubleword. These accesses are not doubleword single-copy atomic.
- Doubleword accesses to a pair of 32-bit locations that are not a doubleword-aligned pair forming a 64-bit register.
- Quadword or higher.
- Exclusive accesses.

For each of these access types, it is CONSTRAINED UNPREDICTABLE whether:

- The access generates an external abort or not.
- The defined side-effects of a read occur or not. A read returns UNKNOWN data.
- A write is ignored or sets the accessed register or registers to UNKNOWN.

For accesses from the external debug interface, the size of an access is determined by the interface. For an access from an ADIv5-compliant Memory Access Port, MEM-AP, this is specified by the MEM-AP CSW register.

See *Access sizes for memory-mapped accesses* on page H8-4951.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H8.3 Synchronization of changes to the external debug registers

This section describes the synchronization requirements for the external debug interface.

For more information on how these requirements affect debug, see:

- *Synchronization and debug exceptions* on page D2-1701 for exceptions taken from AArch64 state, or *Synchronization and debug exceptions* on page G2-3945 for exceptions taken from AArch32 state.
- *Synchronization and Halting debug events* on page H3-4882.
- *Synchronization of DCC and ITR accesses* on page H4-4897.

This section refers to accesses from the external debug interface as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

**Note**

*Synchronization requirements for System registers* on page D7-1908 defines direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads, and external writes as indirect writes.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. With the exception of DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, and DBGWVR<n>_EL1, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

*Synchronization of DCC and ITR accesses* on page H4-4897 describes the synchronization requirements for the DCC and ITR.

Changes to the IMPLEMENTATION DEFINED authentication interface are external writes to the authentication status registers by the master of the authentication interface. See *Synchronization and the authentication interface* on page H8-4947.

Explicit synchronization is not required for an external read or an external write by an external agent to be observable to a following external read or external write by that agent to the same register using the same address, and so is never required for registers that are accessible only in the external debug interface.

Some registers are guaranteed to be observable to all observers in finite time, without explicit synchronization. For more information, see *Synchronization requirements for System registers* on page D7-1908. Otherwise, explicit synchronization is normally required following an external write to any register for that write to be observable by:

- A direct access.
- An indirect read by an instruction.
- An external read of the register using a different address.

This means that an external write by an external agent is guaranteed to have an effect on subsequent instructions executed by the PE only if all of the following are true:

- The write has completed.
- The PE has executed a context synchronization operation.
- The context synchronization operation was executed after the write completed.

The order and synchronization of direct reads and direct writes of System registers is defined by *Synchronization requirements for System registers* on page D7-1908.

The external agent must be able to guarantee completion of a write. For example by:

- Marking the memory as Device-nGnRnE and executing a `DSB` barrier, if the system supports this property.
- Reading back the value written.
- Some guaranteed property of the connection between the PE and the external agent.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

For an external Debug Access Port, this is an IMPLEMENTATION DEFINED property. For a CoreSight system using APB-AP to access a debug APB, a write is guaranteed to complete before the APB-AP allows a second APB transaction to complete.

The external agent and PE can guarantee ordering by, for example, passing messages in an ordered way with respect to the external write and the context synchronization operation, and relying on the memory ordering rules provided by the memory model.

External reads and external writes complete in the order in which they arrive at the PE. For accesses to different register locations the external agent must create this order by:

- Marking the memory as Device-nGnRnE or Device-nGnRE.
- Using the appropriate memory barriers.
- Some guaranteed property of the connection between the PE and the external agent.

**Note**

For an external Debug Access Port, this is an IMPLEMENTATION DEFINED property. For a CoreSight system using APB-AP to access a debug APB, accesses complete in order.

However, the external agent cannot force synchronization of completed writes without halting the PE. Executing an `ISB` instruction, either in Debug state or in Non-debug state, and exiting from Debug state forces synchronization. If the PE is in Debug state, executing an `ISB` instruction is guaranteed to explicitly synchronize any external reads, external writes, and changes to the authentication interface that are ordered before the external write to EDITR.

For any given observer, external writes to the following register groups are guaranteed to be observable in the same order in which they complete:

- The breakpoint registers, DBGBCR<n>_EL1 and DBGBVR<n>_EL1.
- The watchpoint registers, DBGWCR<n>_EL1 and DBGWVR<n>_EL1.

This guarantee only applies to external writes to registers within one of these groups. There is no guarantee regarding the ordering of the observability of external writes within these groups with respect to external writes to registers, for example EDSCR, or between breakpoints and watchpoints, including watchpoints linked to context matching breakpoints.

**Note**

This means that a debugger can rely on the external writes to be observed in the same order in which they complete. It does not mean that a debugger can rely on the external writes being observed in finite time.

In a simple sequential execution an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

If two or more interfaces simultaneously access a register, the behavior must be as if the accesses occurred atomically and in any order. This is described in *Examples of the synchronization of changes to the external debug registers* on page H8-4948.

Some registers have the property that for certain bits a write of 0 is ignored and a write of 1 has an effect. This means that simultaneous writes must be merged. Registers that have this property and support both external debug and System register access include DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, PMCR_EL0.{C,P}, PMOVSSET_EL0, PMOVSCLR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, and PMSWINC_EL0. This last register is OPTIONAL and deprecated in the external debug interface.

### H8.3.1 Synchronization and the authentication interface

Changes to the authentication interface are indirect writes to the Authentication Status registers by the master of the authentication interface. For each of these Authentication Status registers, it is IMPLEMENTATION DEFINED whether a change on the authentication interface is guaranteed to be observable to an external debug interface read of the register only after a context synchronization operation or in finite time.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For DBGAUTHSTATUS_EL1, a change on the authentication interface is guaranteed to be observable to a System register read of DBGAUTHSTATUS_EL1 only after a context synchronization operation.

### H8.3.2 Examples of the synchronization of changes to the external debug registers

Example H8-1, Example H8-2, and Example H8-3 show the synchronization of changes to the external debug registers.

**Example H8-1 Order of synchronization of Breakpoint and Watchpoint register writes**

Initially DBGBVR<n>_EL1 is `0x8000` and DBGBCR<n>_EL1 is `0x0181`. This means that a breakpoint is enabled on the halfword T32 instruction at address `0x8000`.

A sequence of external writes occurs in the following order:

1. `0x0000` is written to DBGBCR<n>_EL1, disabling the breakpoint.
2. `0x9000` is written to DBGBVR<n>_EL1[31:0].
3. `0x0061` is written to DBGBCR<n>_EL1, enabling a breakpoint on the halfword at address `0x9002`.

The external writes must be observable to indirect reads in the same order as the external writes complete. This means that at no point is there a breakpoint enabled on either of the halfwords at address `0x8002` and `0x9000`.

Similarly a breakpoint or watchpoint must be disabled:

- If both halves of a 64-bit address have to be updated.
- If any of the DBGBCR<n>_EL1 or DBGWCR<n>_EL1 fields are modified at the same time as updating the address.

**Example H8-2 Simultaneous accesses to DTR registers**

Initially EDSCR.{TXfull, TXU, ERR} are 0. Then:

- `0x0DCCDA7A` is directly written to DBGDTRTX_EL0 by an `MSR` instruction.
- DBGDTRTX_EL0 is indirectly read by the external debug interface.

These accesses might happen at the same time and in any order.

If the direct write of `0x0DCCDA7A` to DBGDTRTX_EL0 is handled first, then:

- The external debug interface read of DBGDTRTX_EL0 clears EDSCR.TXfull to 0.
- EDSCR.{TXU, ERR} are unchanged.
- The external debug interface read returns `0x0DCCDA7A`.

If the indirect read of DBGDTRTX_EL0 by the external debug interface is handled first, then:

- The external debug interface read of DBGDTRTX_EL0 causes an underrun and as a result EDSCR.{TXU, ERR} are both set to 1.
- The external debug interface returns an UNKNOWN value.
- Writing `0x0DCCDA7A` to DBGDTRTX_EL0 sets DTRTX to `0x0DCCDA7A` and EDSCR.TXfull to 1.

**Example H8-3 Simultaneous writes to CLAIM registers**

Initially all CLAIM tag bits are 0. Then:

- `0x01` is written to DBGCLAIMSET_EL1 by a direct write, followed by an explicit context synchronization operation.
- `0x02` is written to DBGCLAIMSET_EL1 by an external write.

These events might happen at the same time and in either order.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

After this:

- DBGCLAIMCLR_EL1 is read by a direct read.
- DBGCLAIMCLR_EL1 is read by an external read.

In this case, a direct read can return either `0x01` or `0x03`, and the external read can return either `0x02` or `0x03`.

The only permitted final result for the CLAIM tags is the value `0x03`, because this would be the result regardless of whether `0x01` or `0x02` is written first. This is because the external write is guaranteed to be observable to a direct read in finite time. See *Synchronization requirements for System registers* on page D7-1908.

It is not possible for a direct read to return `0x01` and the external read to return `0x02`, because the writes to DBGCLAIMCLR_EL1 are serialized.

In the following scenario, there is only one permitted result. Both observers observe the value `0x03`, and then, at the same time, two writes occur:

- `0x04` is written to DBGCLAIMSET_EL1 by a direct write, followed by an explicit context synchronization operation.
- `0x01` is written to DBGCLAIMCLR_EL1 by an external write.

In this case only permitted final result for the CLAIM tags is the value `0x06`.

---


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H8.4 Memory-mapped accesses to the external debug interface

Support for memory-mapped access to the external debug interface is OPTIONAL.

If the external debug interface is CoreSight compliant, then an OPTIONAL Software Lock can be implemented for memory-mapped accesses to each component. The Software Locks are controlled by EDLSR and EDLAR, PMLSR and PMLAR, and CTILSR and CTILAR. See *Management registers and CoreSight compliance* on page K2-5523.

With the exception of these registers and the effect of the Software Lock, the behavior of the memory-mapped accesses is the same as for other accesses to the external debug interface.

**Note**

The recommended memory-mapped accesses to the external debug interface are not compatible with the memory-mapped interface defined in ARMv7. In particular:

- The memory map is different.
- Memory-mapped accesses do not behave differently to Debug Access Port accesses when OSLSR.OSLK == 1, meaning that the OS Lock is locked.

### H8.4.1 Register access permissions for memory-mapped accesses

It is IMPLEMENTATION DEFINED whether unprivileged memory-mapped accesses are allowed. Privileged software is responsible for controlling memory-mapped accesses using the MMU.

If memory-mapped accesses are made through an ADIv5 interface, the Debug Access Port can block the access using **DBGSWENABLE**. This is outside the scope of the ARMv8-A architecture. See *ARM® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2*.

#### Effect of the OPTIONAL Software Lock on memory-mapped access

For memory-mapped accesses, if other controls permit access to a register, the OPTIONAL Software Lock is implemented, and EDLSR.SLK, PMLSR.SLK, or CTILSR.SLK is set to 1, meaning the Software Lock is locked, then with the exception of the LAR itself:

- If other controls permit access to a register, then writes are ignored. That is:
  - Read/write (RW) registers become read-only (RO).
  - Write-only (WO) registers become write-ignored (WI).
- Reads and writes have no side-effects. A side-effect is where a direct read or a direct write of a register creates an indirect write of the same or another register. When the Software Lock is locked, the indirect write does not occur.
- Writes to EDLAR, PMLAR, and CTILAR are unaffected.

This behavior must also apply to all IMPLEMENTATION DEFINED registers.

For example, if EDLSR.SLK is set to 1:

- EDSCR.{TXfull, TXU, ERR} are unchanged by a memory-mapped read from DBGDTRTX_EL0.
- EDSCR.{RXfull, RXO, ERR} are unchanged by a memory-mapped write to DBGDTRRX_EL0 that is ignored.
- EDSCR.{ITE, ITO, ERR} are unchanged by a memory-mapped write to EDITR that is ignored.
- OSLSR.OSLK is unchanged by a memory-mapped write to OSLAR_EL1 that is ignored.
- EDPCSR[63:32], EDCIDSR, and EDVIDSR are unchanged by a memory-mapped read from EDPCSR[31:0].


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Updating EDVIDSR, EDCIDSR, and EDPCSRhi are side-effects of reading EDPCSRlo, such that these registers contain the matching context for EDPCSRlo. The process that updates EDPCSRlo with PC samples is not a side-effect of the access. Reads of EDPCSRlo made when the Software Lock is locked can be used to profile software.

- EDPRSR.{SDR, SPMAD, SDAD, SR, SPD} are unchanged by a memory-mapped read from EDPRSR.
- EDPRSR.SDAD is not set if an error response is returned due to a memory-mapped read or write of any debug register as the result of the value of the EDAD field.
- The CLAIM tags are unchanged by memory-mapped writes to DBGCLAIMSET_EL1 and DBGCLAIMCLR_EL1 which are ignored.

Similarly, if PMLSR.SLK is set to 1, then EDPRSR.SPMAD is not set if an error response is returned to a memory-mapped read or write of any Performance Monitors register due to the value of the EPMAD field.

#### Behavior of a not permitted memory-mapped access

Where the architecture requires that an external debug interface access generates an error response, a memory-mapped access must also generate an error response. However, it is IMPLEMENTATION DEFINED how the error response is handled, as this depends on the system.

ARM recommends that the error is returned as either:

- A synchronous external Data Abort.
- An SError interrupt.

### H8.4.2 Synchronization of memory-mapped accesses to external debug registers

The synchronization requirements for memory-mapped accesses to the external debug interface is described in *Synchronization of changes to the external debug registers* on page H8-4946.

The synchronization requirements between different routes to the external debug interface, that is, between Debug Access Port accesses and memory-mapped accesses are IMPLEMENTATION DEFINED.

### H8.4.3 Access sizes for memory-mapped accesses

For memory-mapped accesses from a PE that complies with an ARM architecture, the single-copy atomicity rules for the instruction, the type of instruction, and the type of memory accessed, determine the size of the access made by an instruction. Example H8-4 shows this.

**Example H8-4 Access sizes for memory-mapped accesses**

Two Load Doubleword instructions made to consecutive doubleword-aligned locations generate a pair of single-copy atomic doubleword reads. However, if the accesses are made to Normal memory or Device-GRE memory they might appear as a single quadword access that is not supported by the peripheral.

ARMv8 does not require the size of each element accessed by a multi-register load or store instruction to be identifiable by the memory system beyond the PE. Any memory-mapped access to a debug register is defined to be beyond the PE.

Software must use a Device-nGRE or stronger memory-type and use only single register load and store instructions to create memory accesses that are supported by the peripheral. For more information, see *Memory types and attributes* on page B2-93.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H8.5 External debug interface register access permissions

Some external accesses to debug registers and Performance Monitor registers are not permitted and return an error response if:

- The Core power domain is powered down or is in low-power state where the registers cannot be accessed.
- OSLSR.OSLK == 1. The OS Lock is locked.
- `DoubleLockStatus()` == TRUE. The OS Double Lock is locked, that is, EDPRSR.DLK == 1.
- Access by the external debug interface is disabled by the authentication interface or secure monitor.

Not all registers are affected in all of these cases. For details, see *External debug interface register access permissions summary* on page H8-4954.

**Note**

OSLSR.OSLK is visible through EDPRSR.

### H8.5.1 External debug over powerdown and locks

Accessing registers using the external debug interface is not possible when the Debug power domain is off. In this case all accesses return an error.

External accesses to debug and Performance Monitors registers in the Core power domain are not permitted and return an error response if:

- The Core power domain is off or in low-power state where the registers cannot be accessed.
- OSLSR.OSLK == 1, meaning that the OS Lock is locked. This allows software to prevent external debugger modification of the registers while it saves and restores them over powerdown.

  **Note**

  In this case OSLAR_EL1 can be accessed, meaning an external debugger can override this lock.

- `DoubleLockStatus()` == TRUE. This means that the OS Double Lock is locked and EDPRSR.DLK == 1. The OS Double Lock ensures that it is safe to remove Core power by forcing the debug interface to be quiescent.

See also *Debug registers to save over powerdown* on page H6-4930.

### H8.5.2 External access disabled

Accesses are further controlled by the external authentication interface. An untrusted external debugger cannot program the breakpoint and watchpoint registers to generate spurious debug exceptions. If external invasive debugging is not enabled, these external accesses to the registers are disabled. If EL3 is implemented, then SDCR provides additional external access disable controls for those registers if Secure external invasive debugging is disabled.

The disable applies to:

- *DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15* on page H9-4973.
- *DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15* on page H9-4970.
- *DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15* on page H9-4987.
- *DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15* on page H9-4984.

The external debug interface cannot access these registers if either:

- External debugging is not enabled. `ExternalInvasiveDebugEnabled()` == FALSE.
- Secure external debugging is not enabled, meaning `ExternalSecureInvasiveDebugEnabled()` == FALSE, and any of the following:
  - EL3 is not implemented and the implemented Security state is Secure state.
  - EL3 is implemented and SDCR.EDAD == 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The following pseudocode outlines the AllowExternalDebugAccess() function.

```
// AllowExternalDebugAccess()
// ==========================
// Returns the status of EDPRSR.EDAD.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if ExternalInvasiveDebugEnabled() then
        if ExternalSecureInvasiveDebugEnabled() then
            return TRUE;
        elsif HaveEL(EL3) then
            return (if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD) == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

PEs might also provide an OPTIONAL external debug interface to the Performance Monitor registers. The authentication interface and SDCR provide similar external access disable controls for those registers.

The external debug interface cannot access the Performance Monitor registers if either:

- External non-invasive debug is not enabled. ExternalNoninvasiveDebugEnabled() == FALSE.
- Secure external non-invasive debugging is not enabled, ExternalSecureNoninvasiveDebugEnabled() == FALSE, and any of:
  - EL3 is not implemented and the implemented Security state is Secure state.
  - EL3 is implemented and SDCR.EPMAD == 1.

The following pseudocode outlines the AllowExternalPMUAccess() function.

```
// AllowExternalPMUAccess()
// ========================
// Returns the status of EDPRSR.EPMAD.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if ExternalNoninvasiveDebugEnabled() then
        if ExternalSecureNoninvasiveDebugEnabled() then
            return TRUE;
        elsif HaveEL(EL3) then
            return (if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD) == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

**Note**

- ARM recommends that secure software that is not making use of debug hardware does not lock out the external debug interface.
- ARMv8-A does not provide the equivalent control over access to Trace extension registers.

### H8.5.3 Behavior of a not permitted access

For an external debug interface access by a Debug Access Port, the Debug Access Port receives the error response and must signal this to the external debugger. For an ADIv5 implementation of a Debug Access Port, the error sets a sticky error flag in the Debug Access Port that the debugger can poll, and that suppresses further accesses until it is explicitly cleared.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When an error is returned because external access is disabled, and this is the highest priority error condition, a sticky error flag in EDPRSR is indirectly written to 1 as a side-effect of the access:

- For a debug register access when `AllowExternalDebugAccess()` == FALSE, EDPRSR.SDAD is indirectly written to 1.
- For Performance Monitor register access when `AllowExternalPMUAccess()` == FALSE, EDPRSR.SPMAD is indirectly written to 1.

The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see *Register access permissions for memory-mapped accesses* on page H8-4950.

If no error is returned, or the error is returned because of a higher priority error condition, the flag in EDPRSR is unchanged.

See also *Behavior of a not permitted memory-mapped access* on page H8-4951.

For more information, see *ARM® Debug Interface Architecture Specification*.

### H8.5.4 Trapping software access to debug registers

When EDSCR.TDA == 1, software access to the breakpoint and watchpoint registers generate a Halting debug event and entry to Debug state. For more information see *Software Access debug event* on page H3-4881.

### H8.5.5 External debug interface register access permissions summary

For accesses to:

- IMPLEMENTATION DEFINED registers, see *IMPLEMENTATION DEFINED registers*.
- OPTIONAL registers for CoreSight compliance, see *OPTIONAL CoreSight management registers*.
- Reserved, unallocated, or unimplemented registers, writes to read-only registers, and reads of write-only registers, see *Reserved and unallocated registers*.

For all other external debug interface, CTI, and Performance Monitor registers, Table H8-3 on page H8-4959, Table H8-4 on page H8-4961 and Table I2-1 on page I2-5113, show the response of the PE to accesses by the external debug interface.

### H8.5.6 IMPLEMENTATION DEFINED registers

For debug registers, Performance Monitors registers, CTI registers, IMPLEMENTATION DEFINED register access permissions are IMPLEMENTATION DEFINED. The power domain in which these registers are implemented is also IMPLEMENTATION DEFINED.

If OPTIONAL memory-mapped access to the external debug interface is supported, there are additional constraints on memory-mapped accesses to registers. These constraints must also apply to IMPLEMENTATION DEFINED registers. In particular, if the OPTIONAL Software Lock is locked, writes are ignored and accesses have no side-effects. For more information see *Register access permissions for memory-mapped accesses* on page H8-4950.

### H8.5.7 OPTIONAL CoreSight management registers

Compliance with CoreSight architecture requires additional registers in the range `0xF00 - 0xFFC` that are always accessible. See *Management registers and CoreSight compliance* on page K2-5523.

### H8.5.8 Reserved and unallocated registers

The following information relates to certain types of reserved accesses:

- Reads and writes of unallocated locations. These accesses are reserved for the architecture.
- Reads and writes of locations for features that are not implemented, including:
  - OPTIONAL features that are not implemented.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

— Breakpoints and watchpoints that are not implemented.
— Performance Monitors counters that are not implemented.
— CTI triggers that are not implemented.

These accesses are reserved.

- Reads of WO locations. These accesses are reserved for the architecture.
- Writes to RO locations. These accesses are reserved for the architecture.

Reserved accesses normally RAZ/WI. However, software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

**Note**

Reads of WO and writes to RO refers to the default access permissions for a register. For example, when the SLK field is set, meaning that the relevant registers become RO, a memory-mapped write to a RW register is ignored, and not treated as a reserved access.

The following reserved registers are RES0 in all conditions, other than when debug power is off:

- If the implementation is CoreSight architecture compliant, all reserved registers in the range `0xF00 - 0xFFC`. See *Management register access permissions* on page K2-5524.
- All unallocated Processor ID Registers. That is, unallocated debug registers in the range `0xD00-0xDFC`.
- All reserved CTI registers.

Otherwise, the architecture defines that:

1. If debug power is off, all register accesses, including reserved accesses, return an error.
2. For reserved debug registers and Performance Monitors registers, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0, when any of the following hold:

   **Off** The Core power domain is either completely off or in a low-power state in which the Core power domain registers cannot be accessed.

   **DLK** `DoubleLockStatus()` == TRUE. The OS Double Lock is locked, that is, EDPRSR.DLK == 1.

   **OSLK** OSLSR.OSLK == 1. The OS Lock is locked.

3. In addition, for reserved debug registers in the address ranges `0x400 - 0x4FC` and `0x800 - 0x8FC`, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

   **EDAD** `AllowExternalDebugAccess()` == FALSE. External debug is disabled.

   **Note**

   See also *Behavior of a not permitted access* on page H8-4953.

4. In addition, for reserved Performance Monitors registers in the address ranges `0x000 - 0x0FC` and `0x400 - 0x47C`, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

   **EPMAD** `AllowExternalPMUAccess()` == FALSE. External Performance Monitor access is disabled.

   **Note**

   See also *Behavior of a not permitted access* on page H8-4953.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H8.6 External debug interface registers

The external debug interface register map is described by:

- *Performance Monitors memory-mapped register views* on page I3-5119.
- *Cross-trigger interface registers* on page H8-4961.
- Table H8-2.

**Table H8-2 External debug interface register map**

| Offset | Mnemonic | Register, or additional information |
|---|---|---|
| 0x020 | EDESR | *EDESR, External Debug Event Status Register* on page H9-5012 |
| 0x024 | EDECR | *EDECR, External Debug Execution Control Register* on page H9-5010 |
| 0x030<br>0x034 | EDWAR[31:0]<br>EDWAR[63:32] | *EDWAR, External Debug Watchpoint Address Register* on page H9-5050 |
| 0x080 | DBGDTRRX_EL0 | Chapter H4 *The Debug Communication Channel and Instruction Transfer Register* |
| 0x084 | EDITR | *EDITR, External Debug Instruction Transfer Register* on page H9-5016 |
| 0x088 | EDSCR | *EDSCR, External Debug Status and Control Register* on page H9-5043 |
| 0x08C | DBGDTRTX_EL0 | Chapter H4 *The Debug Communication Channel and Instruction Transfer Register* |
| 0x090 | EDRCR | *EDRCR, External Debug Reserve Control Register* on page H9-5041 |
| 0x094 | EDACR | *EDACR, External Debug Auxiliary Control Register* on page H9-4991 |
| 0x098 | EDECCR | *EDECCR, External Debug Exception Catch Control Register* on page H9-5008 |
| 0x0A0 | EDPCSRlo[a] | *EDPCSR, External Debug Program Counter Sample Register* on page H9-5021 |
| 0x0A4 | EDCIDSR[a] | *EDCIDSR, External Debug Context ID Sample Register* on page H9-4996 |
| 0x0A8 | EDVIDSR[a] | *EDVIDSR, External Debug Virtual Context Sample Register* on page H9-5048 |
| 0x0AC | EDPCSRhi[a] | *EDPCSR, External Debug Program Counter Sample Register* on page H9-5021 |
| 0x0300 | OSLAR_EL1 | *OSLAR_EL1, OS Lock Access Register* on page H9-5053 |
| 0x0310 | EDPRCR | *EDPRCR, External Debug Power/Reset Control Register* on page H9-5030 |
| 0x0314 | EDPRSR | *EDPRSR, External Debug Processor Status Register* on page H9-5033 |
| 0x0400+16×n<br>0x0404+16×n | DBGBVR<n>_EL1[31:0][bc]<br>DBGBVR<n>_EL1[63:32][bc] | *DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15* on page H9-4973 |
| 0x0408+16×n | DBGBCR<n>_EL1 | *DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15* on page H9-4970 |
| 0x800+16<br>0x804+16×n | DBGWVR<n>_EL1[31:0][bc]<br>DBGWVR<n>_EL1[63:32][bc] | *DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15* on page H9-4987 |
| 0x808+16×n | DBGWCR<n>_EL1[c] | *DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15* on page H9-4984 |
| 0xC00-0xCFC | IMPLEMENTATION DEFINED | - |
| 0xD00 | MIDR_EL1 | Main ID register |
| 0xD04-0xD1C | - | Reserved, RES0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table H8-2 External debug interface register map (continued)**

| Offset | Mnemonic | Register, or additional information |
|---|---|---|
| 0xD20 | EDPFR[31:0] | External Debug Processor Feature Register 0 |
| 0xD24 | EDPFR[63:32] | |
| 0xD28 | EDDFR[31:0] | External Debug Feature Register 0 |
| 0xD2C | EDDFR[63:32] | |
| 0xD30 | Reserved, see next column | Previously defined as Instruction Set Attribute Register 0 bits[31:0]. Behavior is:<br>**Bits[31:20]** RES0.<br>**Bits[19:4]** UNKNOWN.<br>**Bits[3:0]** RES0. |
| 0xD34 | RES0 | Previously defined as Instruction Set Attribute Register 0 bits[63:32] |
| 0xD38 | UNKNOWN | Previously defined as Memory Model Feature Register 0 |
| 0xD3C | RES0 | |
| 0xD40-0xDFC | RES0 | Reserved, RES0 |
| 0xD60 | EDAA32PFR | External Debug AArch32 Processor Feature Register |
| 0xE80-EFC | IMPLEMENTATION DEFINED | - |
| 0xF00-E8C | Management registers | *Management registers and CoreSight compliance* on page K2-5523 |
| 0xFA0 | DBGCLAIMSET_EL1 | *DBGCLAIMSET_EL1, Debug CLAIM Tag Set register* on page H9-4978 |
| 0xFA4 | DBGCLAIMCLR_EL1 | *DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register* on page H9-4976 |
| 0xFA8 | EDDEVAFF0 | *EDDEVAFF0, External Debug Device Affinity register 0* on page H9-4997 |
| 0xFAC | EDDEVAFF1 | *EDDEVAFF1, External Debug Device Affinity register 1* on page H9-4998 |
| 0xFB0-FB4 | Management registers | *Management registers and CoreSight compliance* on page K2-5523 |
| 0xFB8 | DBGAUTHSTATUS_EL1 | *DBGAUTHSTATUS_EL1, Debug Authentication Status register* on page H9-4968 |
| 0xFC0 | EDDEVID2 | *EDDEVID, External Debug Device ID register 0* on page H9-5001 |
| 0xFC4 | EDDEVID1 | *EDDEVID1, External Debug Device ID register 1* on page H9-5003 |
| 0xFC8 | EDDEVID | *EDDEVID2, External Debug Device ID register 2* on page H9-5004 |
| 0xFD0-FFC | Management registers | *Management registers and CoreSight compliance* on page K2-5523 |

a. Only if the OPTIONAL Sample-based Profiling Extension is implemented.

b. A 64-bit register mapped to a pair of 32-bit locations. Doubleword accesses to this register are not guaranteed to be 64-bit single copy atomic. See *Supported access sizes* on page H8-4945. Software must ensure a breakpoint or watchpoint is disabled before altering the value register.

c. Implemented breakpoints and watchpoints only. *n* is the breakpoint or the watchpoint number.

**Note**

All other locations are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H8.6.1 Access permissions for the External debug interface registers

Table H8-3 on page H8-4959 shows the access permissions for the external debug interface registers in an ARMv8-A Debug implementation. The terms are defined as follows:

**Domain** This describes the power domain in which the register is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.

**Conditions** This lists the conditions under which the access is attempted.

To determine the access permissions for a register, read these columns from left to right, and stop at first column which lists the condition as being true.

The conditions are:

**Off** EDPRSR.PU == 0. The Core power domain is completely off, or in low-power state. In these cases the Core power domain registers cannot be accessed.

**Note**

If debug power is off, then all external debug interface accesses return an error.

**DLK** `DoubleLockStatus()` == TRUE. The OS Double Lock is locked, that is, EDPRSR.DLK == 1.

**OSLK** OSLSR.OSLK == 1. The OS Lock is locked.

**EDAD** `AllowExternalDebugAccess()` == FALSE. External debug access is disabled. See also *Behavior of a not permitted access* on page H8-4953.

**EPMAD** `AllowExternalPMUAccess()` == FALSE. Access to the external Performance Monitors is disabled. See also *Behavior of a not permitted access* on page H8-4953.

**Default** This provides the default access permissions, if there are no conditions that prevent access to the register.

**SLK** This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked. See *Register access permissions for memory-mapped accesses* on page H8-4950. For all other accesses, this column is ignored.

The access permissions are:

**-** This means that the default access permission applies. See the Default column, or the SLK column, if applicable.

**RO** This means that the register or field is read-only.

**RW** This means that the register or field is read/write. Individual fields within the register might be RO. See the relevant register description for details.

**RC** This means that the bit clears to 0 after a read.

**(SE)** This means that accesses to this register have indirect write side-effects. A side-effect occurs when a direct read or a direct write of a register creates an indirect write to the same register or to another register.

**WO** This means that the register or field is write-only.

**WI** This means that the register or field ignores writes.

**IMP DEF** This means that the access permissions are IMPLEMENTATION DEFINED.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If OPTIONAL memory-mapped access to the external debug interface is supported, there might be additional constraints on memory-mapped accesses. See *Register access permissions for memory-mapped accesses* on page H8-4950.

**Table H8-3 Access permissions for the external debug interface registers**

| | | | Conditions (priority from left to right) | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Offset** | **Register** | **Domain** | **Off** | **DLK** | **OSLK** | **EDAD** | **Default** | **SLK** |
| 0x020 | EDESR | Core | Error | Error | - | - | RW | RO |
| 0x024 | EDECR | Debug | - | - | - | - | RW | RO |
| 0x030<br>0x034 | EDWAR[31:0]<br>EDWAR[63:32] | Core | Error | Error | Error | - | RO | - |
| 0x080 | DBGDTRRX_EL0 | Core | Error | Error | Error | - | RW | RO |
| 0x084 | EDITR | Core | Error | Error | Error | - | WO | WI |
| 0x088 | EDSCR | Core | Error | Error | Error | - | RW | RO |
| 0x08C | DBGDTRTX_EL0 | Core | Error | Error | Error | - | RW | RO |
| 0x090 | EDRCR | Core | Error | Error | Error | - | WO | WI |
| 0x094 | EDACR | IMP DEF | IMP DEF | IMP DEF | IMP DEF | - | RW | RO |
| 0x098 | EDECCR | Core | Error | Error | Error | - | RW | RO |
| 0x0A0 | EDPCSR[31:0][a] | Core | Error | Error | Error | - | RO | RO |
| 0x0A4 | EDCIDSR[a] | Core | Error | Error | Error | - | RO | RO |
| 0x0A8 | EDVIDSR[a] | Core | Error | Error | Error | - | RO | RO |
| 0x0AC | EDPCSR[63:32][a] | Core | Error | Error | Error | - | RO | RO |
| 0x0300 | OSLAR_EL1 | Core | Error | Error | - | - | WO | WI |
| 0x0310[b] | EDPRCR | See register field descriptions for information | | | | | | |
| 0x0314[c] | EDPRSR | See register field descriptions for information | | | | | | |
| 0x0400+16×n<br>0x0404+16×n | DBGBVR<n>_EL1[31:0][d]<br>DBGBVR<n>_EL1[63:32][d] | Core<br>Core | Error<br>Error | Error<br>Error | Error<br>Error | Error<br>Error | RW<br>RW | RO<br>RO |
| 0x0408+16×n | DBGBCR<n>_EL1[d] | Core | Error | Error | Error | Error | RW | RO |
| 0x800+16×n<br>0x804+16×n | DBGWVR<n>_EL1[31:0][d]<br>DBGWVR<n>_EL1[63:32][d] | Core<br>Core | Error<br>Error | Error<br>Error | Error<br>Error | Error<br>Error | RW<br>RW | RO<br>RO |
| 0x808+16×n | DBGWCR<n>_EL1[d] | Core | Error | Error | Error | Error | RW | RO |
| 0xD00 | MIDR_EL1 | Debug | - | - | - | - | RO | RO |
| 0xD20 | ID_AA64PFR0_EL1[31:0] | Debug | - | - | - | - | RO | RO |
| 0xD24 | ID_AA64PFR0_EL1[63:32] | Debug | - | - | - | - | RO | RO |
| 0xD28 | ID_AA64DFR0_EL1[31:0] | Debug | - | - | - | - | RO | RO |
| 0xD2C | ID_AA64DFR0_EL1[63:32] | Debug | - | - | - | - | RO | RO |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table H8-3 Access permissions for the external debug interface registers (continued)**

| | | | Conditions (priority from left to right) | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Offset** | **Register** | **Domain** | **Off** | **DLK** | **OSLK** | **EDAD** | **Default** | **SLK** |
| 0xD30 | ID_AA64ISAR0_EL1[31:0] | Debug | - | - | - | - | RO | RO |
| 0xD34 | ID_AA64ISAR0_EL1[63:32] | Debug | - | - | - | - | RO | RO |
| 0xD38 | ID_AA64MMFR0_EL1[31:0] | Debug | - | - | - | - | RO | RO |
| 0xD3C | ID_AA64MMFR0_EL1[63:32] | Debug | - | - | - | - | RO | RO |
| 0xD40 | ID_AA64PFR1_EL1[31:0] | Debug | - | - | - | - | RO | RO |
| 0xD44 | ID_AA64PFR1_EL1[63:32] | Debug | - | - | - | - | RO | RO |
| 0xD48 | ID_AA64DFR1_EL1[31:0] | Debug | - | - | - | - | RO | RO |
| 0xD4C | ID_AA64DFR1_EL1[63:32] | Debug | - | - | - | - | RO | RO |
| 0xD50 | ID_AA64ISAR1_EL1[31:0] | Debug | - | - | - | - | RO | RO |
| 0xD54 | ID_AA64ISAR1_EL1[63:32] | Debug | - | - | - | - | RO | RO |
| 0xD58 | ID_AA64MMFR1_EL1[31:0] | Debug | - | - | - | - | RO | RO |
| 0xD5C | ID_AA64MMFR1_EL1[63:32] | Debug | - | - | - | - | RO | RO |
| 0xFA0 | DBGCLAIMSET_EL1 | Core | Error | Error | Error | - | RW | RO |
| 0xFA4 | DBGCLAIMCLR_EL1 | Core | Error | Error | Error | - | RW | RO |
| 0xFA8 | EDDEVAFF0 | Debug | - | - | - | - | RO | RO |
| 0xFAC | EDDEVAFF1 | Debug | - | - | - | - | RO | RO |
| 0xFB8 | DBGAUTHSTATUS_EL1 | Debug | - | - | - | - | RO | RO |
| 0xFC0 | EDDEVID2 | Debug | - | - | - | - | RO | RO |
| 0xFC4 | EDDEVID1 | Debug | - | - | - | - | RO | RO |
| 0xFC8 | EDDEVID | Debug | - | - | - | - | RO | RO |

a. Only if the Sample-based profiling Extension is implemented.
b. Some control bits are in the Core power domain. These bits ignore writes when Core power domain registers cannot be accessed as shown.
c. Some status bits are fetched from the Core power domain. These bits read UNKNOWN when Core power domain registers cannot be accessed as shown.
d. Implemented breakpoints and watchpoints only. *n* is the breakpoint or watchpoint number.

For the reset values for the external debug interface registers, see Table H8-6 on page H8-4963.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H8.7 Cross-trigger interface registers

The embedded Cross-trigger Interface, CTI, is located within its own block of the external debug memory map. There must be one such block for each PE.

If the CTI of a PE does not implement the CTIDEVAFF0 or CTIDEVAFF1 registers it must be located 64KB above the debug registers in the external debug interface.

Table H8-4 shows the CTI register map.

**Table H8-4 Cross-trigger interface map**

| Offset | Mnemonic | Location of further details |
|---|---|---|
| 0x000 | CTICONTROL | *CTICONTROL, CTI Control register* on page H9-5069 |
| 0x010 | CTIINTACK | *CTIINTACK, CTI Output Trigger Acknowledge register* on page H9-5081 |
| 0x014 | CTIAPPSET | *CTIAPPSET, CTI Application Trigger Set register* on page H9-5058 |
| 0x018 | CTIAPPCLEAR | *CTIAPPCLEAR, CTI Application Trigger Clear register* on page H9-5056 |
| 0x01C | CTIAPPPULSE | *CTIAPPPULSE, CTI Application Pulse register* on page H9-5057 |
| 0x020+4×n | CTIINEN<n>[a] | *CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31* on page H9-5080 |
| 0x0A0+4×n | CTIOUTEN<n>[a] | *CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31* on page H9-5087 |
| 0x130 | CTITRIGINSTATUS | *CTITRIGINSTATUS, CTI Trigger In Status register* on page H9-5093 |
| 0x134 | CTITRIGOUTSTATUS | *CTITRIGOUTSTATUS, CTI Trigger Out Status register* on page H9-5094 |
| 0x138 | CTICHINSTATUS | *CTICHINSTATUS, CTI Channel In Status register* on page H9-5061 |
| 0x13C | CTICHOUTSTATUS | *CTICHOUTSTATUS, CTI Channel Out Status register* on page H9-5062 |
| 0x140 | CTIGATE | *CTIGATE, CTI Channel Gate Enable register* on page H9-5079 |
| 0x144 | ASICCTL | *ASICCTL, CTI External Multiplexer Control register* on page H9-5055 |
| 0xE80 - 0xEFC | IMPLEMENTATION DEFINED | IMPLEMENTATION DEFINED. See *Management registers and CoreSight compliance* on page K2-5523 |
| 0xF00 - 0xFBC | Management registers | *Management registers and CoreSight compliance* on page K2-5523 |
| 0xFC0 | CTIDEVID2 | *CTIDEVID2, CTI Device ID register 2* on page H9-5077 |
| 0xFC4 | CTIDEVID1 | *CTIDEVID1, CTI Device ID register 1* on page H9-5076 |
| 0xFC8 | CTIDEVID | *CTIDEVID, CTI Device ID register 0* on page H9-5074 |
| 0xFD0 - 0xFFC | Management registers | *Management registers and CoreSight compliance* on page K2-5523 |

a. Implemented triggers, including triggers that are not connected, only. *n* is the trigger number.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table H8-5 shows the access permissions for the CTI registers in an ARMv8-A Debug implementation. For a definition of the terms used, see *External debug interface registers* on page H8-4956.

**Table H8-5 Access permissions for the CTI registers**

| | | | Conditions (priority from left to right) | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Offset** | **Register** | **Domain** | **Off** | **DLK** | **OSLK** | **EDAD** | **Default** | **SLK** |
| 0x000 | CTICONTROL | Debug | - | - | - | - | RW | RO |
| 0x010 | CTIINTACK | Debug | - | - | - | - | WO | WI |
| 0x014 | CTIAPPSET | Debug | - | - | - | - | RW | RO |
| 0x018 | CTIAPPCLEAR | Debug | - | - | - | - | WO | WI |
| 0x01C | CTIAPPPULSE | Debug | - | - | - | - | WO | WI |
| 0x020+4×n | CTIINEN<n>[a] | Debug | - | - | - | - | RW | RO |
| 0x0A0+4×n | CTIOUTEN<n> | Debug | - | - | - | - | RW | RO |
| 0x130 | CTITRIGINSTATUS | Debug | - | - | - | - | RO | RO |
| 0x134 | CTITRIGOUTSTATUS | Debug | - | - | - | - | RO | RO |
| 0x138 | CTICHINSTATUS | Debug | - | - | - | - | RO | RO |
| 0x13C | CTICHOUTSTATUS | Debug | - | - | - | - | RO | RO |
| 0x140 | CTIGATE | Debug | - | - | - | - | RW | RO |
| 0xFC0 | CTIDEVID2 | Debug | - | - | - | - | RO | RO |
| 0xFC4 | CTIDEVID1 | Debug | - | - | - | - | RO | RO |
| 0xFC8 | CTIDEVID | Debug | - | - | - | - | RO | RO |

a. Implemented triggers only (including triggers that are not connected). *n* is the trigger number.

For the reset values of the CTI registers, see Table H8-7 on page H8-4964.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H8.8 External debug register resets

Each register or field has a defined reset domain:

- Registers and fields in the Warm reset domain are also reset by a Cold reset and unchanged by an External Debug reset that is not coincident with a Cold reset or a Warm reset.
- Registers and fields in the Cold reset domain are unchanged by a Warm reset or an External Debug reset that is not coincident with a Cold reset.
- Registers and fields in the External Debug reset domain are unchanged by a Cold reset or a Warm reset that is not coincident with an External Debug reset.

Table H8-6 and Table H8-7 on page H8-4964 show the external debug register and CTI register resets. For other debug registers and Performance Monitors registers, see *Management register resets* on page K2-5528 and *Power domains and Performance Monitors registers reset* on page I2-5114.

**Note**

By reference to Figure H6-2 on page H6-4935 the power domain can be deduced from the reset domain. Table K2-7 on page K2-5528 also shows reset power domains.

Table H8-6 and Table H8-7 on page H8-4964 do not include:

- Read-only identification registers, such as Processor ID Registers and PMCFGR, that have a fixed value from reset.
- Read-only status registers, such as EDSCR.RW, that are evaluated each time the register is read and that have no meaningful reset value.
- Write-only registers, such as EDRCR, that only have an effect on writes, and have no meaningful reset value.
- Read/write registers, such as breakpoint and watchpoint registers, and EDPRCR.CORENPDRQ, that alias other registers. The reset values are described by the descriptions of those other registers.
- IMPLEMENTATION DEFINED registers. The reset values and reset domains of these registers are also IMPLEMENTATION DEFINED and might be UNKNOWN.

All other fields in the registers are set to an IMPLEMENTATION DEFINED value, that can be UNKNOWN. The register is in the specified reset domain.

**Note**

An IMPLEMENTATION DEFINED reset value, which can be UNKNOWN, means that hardware is not required to reset the register on the specified reset, but software must not rely on the register being preserved over reset.

**Table H8-6 Summary of external debug register resets, debug registers**

| Register | Reset domain | Field | Value | Description |
|---|---|---|---|---|
| EDESR | Warm | SS | EDECR.SS | Halting Step debug event pending |
| | | RC | EDECR.RCE | Reset Catch debug event pending |
| | | OSUC | 0 | OS Unlock Catch debug event pending |
| EDECR | External debug | SS | 0 | Halting Step debug event enable |
| | | RCE | 0 | Reset Catch debug event enable |
| | | OSUCE | 0 | OS Unlock Catch debug event enable |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table H8-6 Summary of external debug register resets, debug registers (continued)**

| Register | Reset domain | Field | Value | Description |
|---|---|---|---|---|
| EDWAR | Cold | - | - | All fields |
| EDSCR | Cold | RXfull | 0 | DTRRX register full |
| | | TXfull | 0 | DTRTX register full |
| | | RXO | 0 | DTRRX overrun |
| | | TXU | 0 | DTRTX underrun |
| | | INTdis | 0 | Interrupt disable |
| | | TDA | 0 | Trap debug register accesses to Debug state |
| | | MA | 0 | Memory access mode in Debug state |
| | | HDE | 0 | Halting debug mode enable |
| | | ERR | 0 | Cumulative error flag |
| EDECCR | Cold | NSE[2:1] | 0b00 | Coarse-grained Non-secure exception catch |
| | | SE[3,1] | 0b00 | Coarse-grained Secure exception catch |
| EDPCSR | Cold | - | - | All fields |
| EDCIDSR | Cold | - | - | All fields |
| EDVIDSR | Cold | - | - | All fields |
| EDPRCR | External debug | COREPURQ[a] | 0 | Core powerup request |
| EDPRSR | Warm | SDR | - | Sticky debug restart |
| | Cold | SPMAD | 0 | Sticky EPMAD error |
| | | SDAD | 0 | Sticky EDAD error |
| | Warm | SR | 1 | Sticky reset status |
| | Cold | SPD | 1 | Sticky powerdown status |

a. On a cold reset into AArch64 state, DBGPRCR_EL1.CORENPRDRQ resets to the value of EDPRCR.COREPURQ. On a cold reset into AArch32 state, DBGPRCR.CORENPRDRQ resets to the value of EDPRCR.COREPURQ.If an External Debug reset and a Cold reset coincide, both EDPRCR.COREPURQ and the CORENPRDRQ field of the appropriate System register are reset to 0.

Table H8-7 shows the reset values for the CTI registers

**Table H8-7 Summary of external debug register resets, CTI registers**

| Register | Reset domain | Field | Value | Description |
|---|---|---|---|---|
| CTICONTROL | External debug | GLBEN | 0 | CTI global enable |
| CTIAPPSET | External debug | - | - | All fields |
| CTIINEN<n> | External debug | - | - | All fields |
| CTIOUTEN<n> | External debug | - | - | All fields |
| CTIGATE | External debug | - | - | All fields |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter H9
# External Debug Register Descriptions

This chapter provides a description of the external debug registers.

It contains the following sections:


- *Introduction* on page H9-4966.
- *Debug registers* on page H9-4967.
- *Cross-Trigger Interface registers* on page H9-5054.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H9.1 Introduction

This section lists the registers that are accessible through the external debug interface.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## H9.2 Debug registers

This section lists the Debug registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.1 DBGAUTHSTATUS_EL1, Debug Authentication Status register

The DBGAUTHSTATUS_EL1 characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

External register DBGAUTHSTATUS_EL1 is architecturally mapped to AArch64 System register DBGAUTHSTATUS_EL1.

External register DBGAUTHSTATUS_EL1 is architecturally mapped to AArch32 System register DBGAUTHSTATUS.

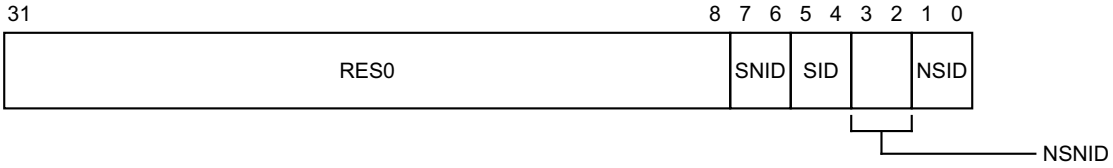
DBGAUTHSTATUS_EL1 is in the Debug power domain.

**Attributes**

DBGAUTHSTATUS_EL1 is a 32-bit register.

## Field descriptions

The DBGAUTHSTATUS_EL1 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**SNID, bits [7:6]**

Secure non-invasive debug. Possible values of this field are:

`00` Not implemented. EL3 is not implemented and the implemented Security state is Non-Secure state.

`10` Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.

`11` Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

Other values are reserved.

**SID, bits [5:4]**

Secure invasive debug. Possible values of this field are:

`00` Not implemented. EL3 is not implemented and the implemented Security state is Non-Secure state.

`10` Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.

`11` Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Other values are reserved.

**NSNID, bits [3:2]**

Non-secure non-invasive debug. Possible values of this field are:

00 Not implemented. EL3 is not implemented and the implemented Security state is Secure state.

10 Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.

11 Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

Other values are reserved.

**NSID, bits [1:0]**

Non-secure invasive debug. Possible values of this field are:

00 Not implemented. EL3 is not implemented and the implemented Security state is Secure state.

10 Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.

11 Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.

Other values are reserved.

### Accessing the DBGAUTHSTATUS_EL1:

DBGAUTHSTATUS_EL1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xFB8 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.2 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n>_EL1 characteristics are:

**Purpose**

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EDAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

When the E field is zero, all the other fields in the register are ignored.

**Configurations**

External register DBGBCR<n>_EL1 is architecturally mapped to AArch64 System register DBGBCR<n>_EL1.

External register DBGBCR<n>_EL1 is architecturally mapped to AArch32 System register DBGBCR<n>.

DBGBCR<n>_EL1 is in the Core power domain.

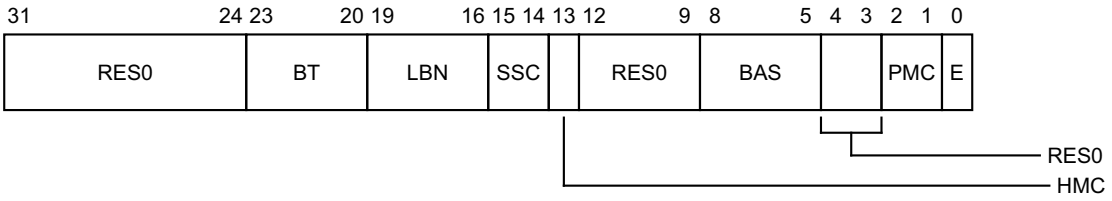
If breakpoint n is not implemented then this register is unallocated.

**Attributes**

DBGBCR<n>_EL1 is a 32-bit register.

## Field descriptions

The DBGBCR<n>_EL1 bit assignments are:




**Bits [31:24]**

Reserved, RES0.

**BT, bits [23:20]**

Breakpoint Type. Possible values are:

`0000` Unlinked instruction address match.

`0001` Linked instruction address match.

`0010` Unlinked context ID match.

`0011` Linked context ID match

`0100` Unlinked instruction address mismatch.

`0101` Linked instruction address mismatch.

`1000` Unlinked VMID match.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`1001` Linked VMID match.

`1010` Unlinked VMID and context ID match.

`1011` Linked VMID and context ID match.

The field breaks down as follows:

- BT[3:1]: Base type.

  `000` Match address. DBGBVR<n>_EL1 is the address of an instruction.

  `010` Mismatch address. Behaves as type `000` if in an AArch64 translation, or if Halting debug is enabled and halting is allowed. Otherwise, DBGBVR<n>_EL1 is the address of an instruction to be stepped.

  `001` Match context ID. DBGBVR<n>_EL1.ContextID is a context ID.

  `100` Match VMID. DBGBVR<n>_EL1.VMID is a VMID.

  `101` Match VMID and context ID. DBGBVR<n>_EL1.ContextID is a context ID, and DBGBVR<n>_EL1.VMID is a VMID.

- BT[0]: Enable linking.

Values `0b011x` and `0b11xx` are always reserved. Constraints on breakpoint programming mean other values are reserved under some conditions. For more information, including the effect of programming this field to a reserved value, see *Reserved DBGBCR<n>_EL1.BT values* on page D2-1663.

**LBN, bits [19:16]**

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of DBGBCR<n>_EL1.E is 0.

**SSC, bits [15:14]**

Security state control. Determines the Security states under which a breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-1664.

**HMC, bit [13]**

Higher mode control. Determines the debug perspective for deciding when a breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see DBGBCR<n>_EL1.SSC description.

**Bits [12:9]**

Reserved, RES0.

**BAS, bits [8:5]**

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state. In an AArch64-only implementation, this field is reserved, RES1. Otherwise:

- BAS[2] and BAS[0] are read/write.
- BAS[3] and BAS[1] are read-only copies of BAS[2] and BAS[0] respectively.

The values `0011` and `1100` are only supported if AArch32 is supported at any Exception level.

The permitted values depend on the breakpoint type.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For Address match breakpoints in either AArch32 or AArch64 state, the permitted values are:

| BAS | Match instruction at | Constraint for debuggers |
|---|---|---|
| 0011 | DBGBVR<n>_EL1 | Use for T32 instructions. |
| 1100 | DBGBVR<n>_EL1+2 | Use for T32 instructions. |
| 1111 | DBGBVR<n>_EL1 | Use for A64 and A32 instructions. |

0000 is reserved and must behave as if the breakpoint is disabled or map to a permitted value.

For more information, see *Using the BAS field in Address Match breakpoints* on page G2-3909.

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

| BAS | Step instruction at | Constraint for debuggers |
|---|---|---|
| 0000 | - | Use for a match anywhere breakpoint. |
| 0011 | DBGBVR<n>_EL1 | Use for stepping T32 instructions. |
| 1100 | DBGBVR<n>_EL1+2 | Use for stepping T32 instructions. |
| 1111 | DBGBVR<n>_EL1 | Use for stepping A64 and A32 instructions. |

For more information, see *Using the BAS field in Address Match breakpoints* on page G2-3909.

For Context matching breakpoints, this field is RES1 and ignored.

**Bits [4:3]**

Reserved, RES0.

**PMC, bits [2:1]**

Privilege mode control. Determines the Exception level or levels at which a breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>_EL1.SSC description.

**E, bit [0]**

Enable breakpoint DBGBVR<n>_EL1. Possible values are:

0 Breakpoint disabled.

1 Breakpoint enabled.

## Accessing the DBGBCR<n>_EL1:

DBGBCR<n>_EL1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x408 + 16n |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n>_EL1 characteristics are:

**Purpose**

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register DBGBCR<n>_EL1.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EDAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register DBGBVR<n>_EL1 is architecturally mapped to AArch64 System register DBGBVR<n>_EL1.

External register DBGBVR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGBVR<n>.

External register DBGBVR<n>_EL1[63:32] is architecturally mapped to AArch32 System register DBGBXVR<n>.

DBGBVR<n>_EL1 is in the Core power domain.

If breakpoint n is not implemented then this register is unallocated.

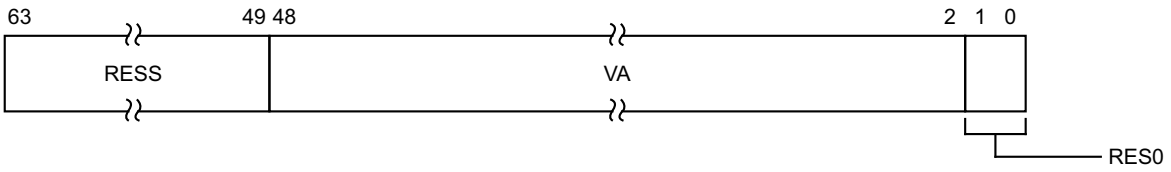
**Attributes**

DBGBVR<n>_EL1 is a 64-bit register.

## Field descriptions

The DBGBVR<n>_EL1 bit assignments are:

***When DBGBCR<n>_EL1.BT==0b0x0x:***




**RESS, bits [63:49]**

Reserved, Sign extended. Hardware and software must treat this field as RES0 if bit[48] is 0 or RES0, and as RES1 if bit[48] is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of bit [48], meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

**VA, bits [48:2]**

If the address is being matched in an AArch64 stage 1 translation regime, this field contains bits[48:2] of the address for comparison.

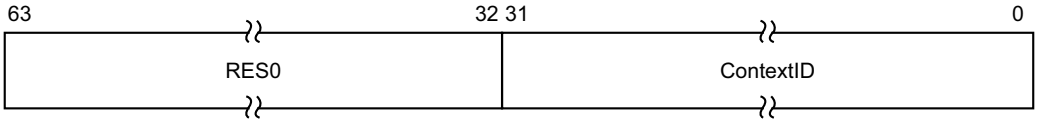

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the address is being matched in an AArch32 stage 1 translation regime, the first 16 bits of this field are RES0, and the rest of the field contains bits[31:2] of the address for comparison.

**Bits [1:0]**

Reserved, RES0.

***When DBGBCR<n>_EL1.BT==0b0x1x:***



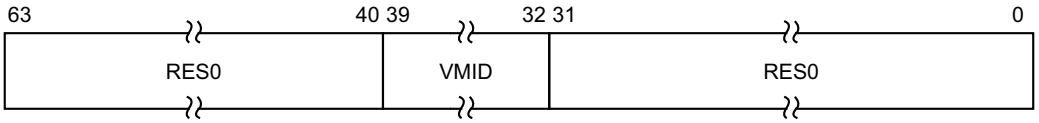

**Bits [63:32]**

Reserved, RES0.

**ContextID, bits [31:0]**

Context ID value for comparison.

***When DBGBCR<n>_EL1.BT==0b1x0x and EL2 implemented:***




**Bits [63:40]**

Reserved, RES0.

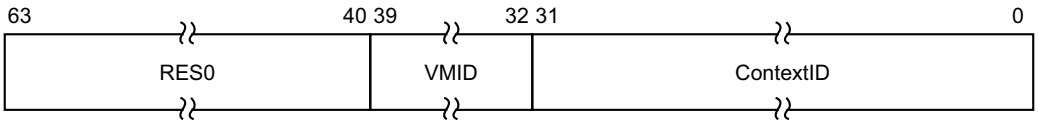
**VMID, bits [39:32]**

VMID value for comparison.

**Bits [31:0]**

Reserved, RES0.

***When DBGBCR<n>_EL1.BT==0x1x1x and EL2 implemented:***




**Bits [63:40]**

Reserved, RES0.

**VMID, bits [39:32]**

VMID value for comparison.

**ContextID, bits [31:0]**

Context ID value for comparison.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the DBGBVR<n>_EL1:

DBGBVR<n>_EL1[31:0] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0x400` + 16n |

DBGBVR<n>_EL1[63:32] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0x404` + 16n |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.2.4 DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register

The DBGCLAIMCLR_EL1 characteristics are:

**Purpose**

Used by software to read the values of the CLAIM tag bits, and to clear these bits to 0.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| Error | Error | Error | RO | RW |

The architecture does not define any functionality for the CLAIM tag bits.

Used in conjunction with the DBGCLAIMSET_EL1 register.

**Configurations**

External register DBGCLAIMCLR_EL1 is architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1.

External register DBGCLAIMCLR_EL1 is architecturally mapped to AArch32 System register DBGCLAIMCLR.

DBGCLAIMCLR_EL1 is in the Core power domain.

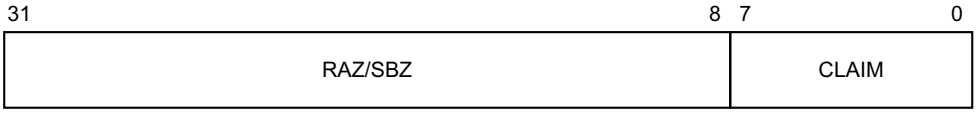
An implementation must include 8 CLAIM tag bits.

**Attributes**

DBGCLAIMCLR_EL1 is a 32-bit register.

#### Field descriptions

The DBGCLAIMCLR_EL1 bit assignments are:




**Bits [31:8]**

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

**CLAIM, bits [7:0]**

Read or clear CLAIM tag bits. Reading this field returns the current value of the CLAIM tag bits.

Writing a 1 to one of these bits clears the corresponding CLAIM tag bit to 0. This is an indirect write to the CLAIM tag bits. A single write operation can clear multiple CLAIM tag bits to 0.

Writing 0 to one of these bits has no effect.

A cold reset clears the CLAIM tag bits to 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the DBGCLAIMCLR_EL1:

DBGCLAIMCLR_EL1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xFA4 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.5 DBGCLAIMSET_EL1, Debug CLAIM Tag Set register

The DBGCLAIMSET_EL1 characteristics are:

**Purpose**

Used by software to set the CLAIM tag bits to 1.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| Error | Error | Error | RO | RW |

The architecture does not define any functionality for the CLAIM tag bits.

Used in conjunction with the DBGCLAIMCLR_EL1 register.

**Configurations**

External register DBGCLAIMSET_EL1 is architecturally mapped to AArch64 System register DBGCLAIMSET_EL1.

External register DBGCLAIMSET_EL1 is architecturally mapped to AArch32 System register DBGCLAIMSET.

DBGCLAIMSET_EL1 is in the Core power domain.

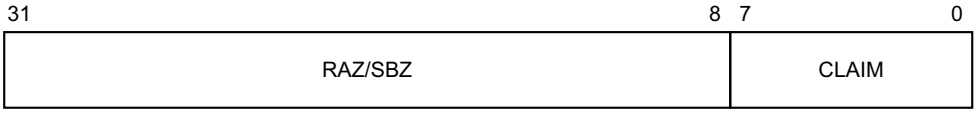
An implementation must include 8 CLAIM tag bits.

**Attributes**

DBGCLAIMSET_EL1 is a 32-bit register.

## Field descriptions

The DBGCLAIMSET_EL1 bit assignments are:




**Bits [31:8]**

Reserved, RAZ/SBZ. Software can rely on these bits reading as zero, and must use a should-be-zero policy on writes. Implementations must ignore writes.

**CLAIM, bits [7:0]**

Set CLAIM tag bits. RAO.

Writing a 1 to one of these bits sets the corresponding CLAIM tag bit to 1. This is an indirect write to the CLAIM tag bits. A single write operation can set multiple CLAIM tag bits to 1.

Writing 0 to one of these bits has no effect.

A cold reset clears the CLAIM tag bits to 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the DBGCLAIMSET_EL1:

DBGCLAIMSET_EL1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xFA0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.6 DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

**Purpose**

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. It is a component of the Debug Communications Channel.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| Error | Error | Error | RO | RW |

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

**Configurations**

External register DBGDTRRX_EL0 is architecturally mapped to AArch64 System register DBGDTRRX_EL0.

External register DBGDTRRX_EL0 is architecturally mapped to AArch32 System register DBGDTRRXint.

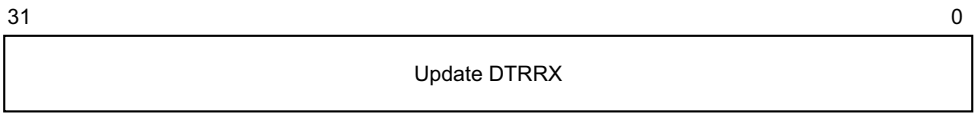
DBGDTRRX_EL0 is in the Core power domain.

**Attributes**

DBGDTRRX_EL0 is a 32-bit register.

## Field descriptions

The DBGDTRRX_EL0 bit assignments are:




**Bits [31:0]**

Update DTRRX.

If RXfull is set to 0, then writes to this register update the value in DTRRX and set RXfull to 1.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the DBGDTRRX_EL0:

DBGDTRRX_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x080 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.7 DBGDTRTX_EL0, Debug Data Transfer Register, Transmit

The DBGDTRTX_EL0 characteristics are:

**Purpose**

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. It is a component of the Debug Communication Channel.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| Error | Error | Error | RO | RW |

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

**Configurations**

External register DBGDTRTX_EL0 is architecturally mapped to AArch64 System register DBGDTRTX_EL0.

External register DBGDTRTX_EL0 is architecturally mapped to AArch32 System register DBGDTRTXint.

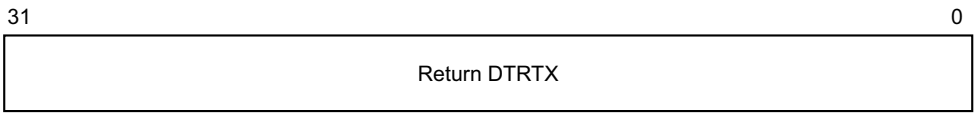
DBGDTRTX_EL0 is in the Core power domain.

**Attributes**

DBGDTRTX_EL0 is a 32-bit register.

## Field descriptions

The DBGDTRTX_EL0 bit assignments are:




**Bits [31:0]**

Return DTRTX.

If TXfull is set to 1, then reads of this register return the value in DTRTX and clear TXfull to 0.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the DBGDTRTX_EL0:

DBGDTRTX_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x08C |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.2.8 DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15

The DBGWCR<n>_EL1 characteristics are:

**Purpose**

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EDAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

When the E field is zero, all the other fields in the register are ignored.

**Configurations**

External register DBGWCR<n>_EL1 is architecturally mapped to AArch64 System register DBGWCR<n>_EL1.

External register DBGWCR<n>_EL1 is architecturally mapped to AArch32 System register DBGWCR<n>.

DBGWCR<n>_EL1 is in the Core power domain.

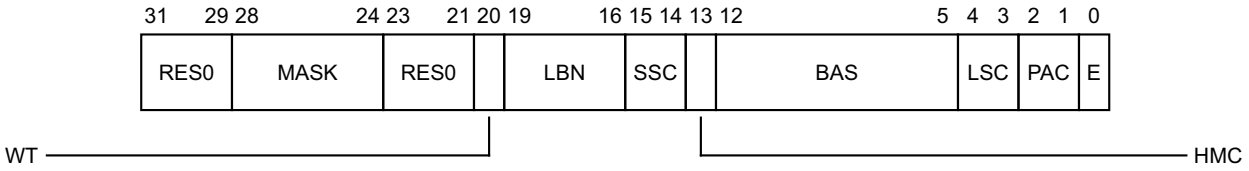
If breakpoint n is not implemented then this register is unallocated.

**Attributes**

DBGWCR<n>_EL1 is a 32-bit register.

## Field descriptions

The DBGWCR<n>_EL1 bit assignments are:




**Bits [31:29]**

Reserved, RES0.

**MASK, bits [28:24]**

Address mask. Only objects up to 2GB can be watched using a single mask.

`00000` No mask.

`00001` Reserved.

`00010` Reserved.

If programmed with a reserved value, a watchpoint must behave as if either:

- MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCRn_EL1.
- The watchpoint is disabled.

Software must not rely on this property because the behavior of reserved values might change in a future revision of the architecture.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Other values mask the corresponding number of address bits, from 0b00011 masking 3 address bits (0x00000007 mask for address) to 0b11111 masking 31 address bits (0x7FFFFFFF mask for address).

**Bits [23:21]**

Reserved, RES0.

**WT, bit [20]**

Watchpoint type. Possible values are:

0 Unlinked data address match.

1 Linked data address match.

**LBN, bits [19:16]**

Linked breakpoint number. For Linked data address watchpoints, this specifies the index of the Context-matching breakpoint linked to.

**SSC, bits [15:14]**

Security state control. Determines the Security states under which a watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

**HMC, bit [13]**

Higher mode control. Determines the debug perspective for deciding when a watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

**BAS, bits [12:5]**

Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n>_EL1 is being watched.

| BAS | Description |
|---|---|
| xxxxxxx1 | Match byte at DBGWVR<n>_EL1 |
| xxxxxx1x | Match byte at DBGWVR<n>_EL1+1 |
| xxxxx1xx | Match byte at DBGWVR<n>_EL1+2 |
| xxxx1xxx | Match byte at DBGWVR<n>_EL1+3 |

In cases where DBGWVR<n>_EL1 addresses a double-word:

| BAS | Description, if DBGWVR<n>_EL1[2] == 0 |
|---|---|
| xxx1xxxx | Match byte at DBGWVR<n>_EL1+4 |
| xx1xxxxx | Match byte at DBGWVR<n>_EL1+5 |
| x1xxxxxx | Match byte at DBGWVR<n>_EL1+6 |
| 1xxxxxxx | Match byte at DBGWVR<n>_EL1+7 |

If DBGWVR<n>_EL1[2] == 1, only BAS[3:0] is used. ARM deprecates setting DBGWVR<n>_EL1[2] == 1.

The valid values for BAS are 0b00000000, or a binary number all of whose set bits are contiguous. All other values are reserved and must not be used by software.

If BAS is zero, no bytes are watched by this watchpoint.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**LSC, bits [4:3]**

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

`01` Match instructions that load from a watchpointed address.

`10` Match instructions that store to a watchpointed address.

`11` Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

**PAC, bits [2:1]**

Privilege of access control. Determines the Exception level or levels at which a watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

**E, bit [0]**

Enable watchpoint n. Possible values are:

`0` Watchpoint disabled.

`1` Watchpoint enabled.

### Accessing the DBGWCR<n>_EL1:

DBGWCR<n>_EL1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0x808` + 16n |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.9 DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15

The DBGWVR<n>_EL1 characteristics are:

**Purpose**

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>_EL1.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EDAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register DBGWVR<n>_EL1 is architecturally mapped to AArch64 System register DBGWVR<n>_EL1.

External register DBGWVR<n>_EL1[31:0] is architecturally mapped to AArch32 System register DBGWVR<n>.

DBGWVR<n>_EL1 is in the Core power domain.

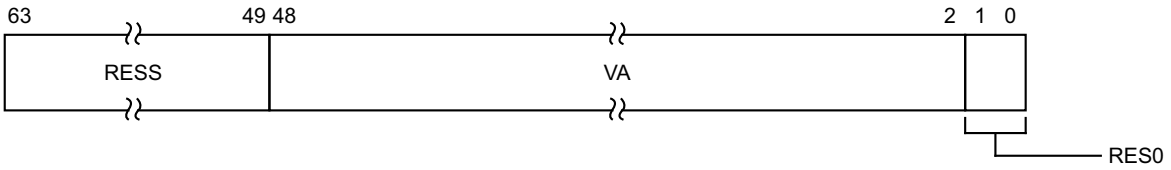
If breakpoint n is not implemented then this register is unallocated.

**Attributes**

DBGWVR<n>_EL1 is a 64-bit register.

## Field descriptions

The DBGWVR<n>_EL1 bit assignments are:




**RESS, bits [63:49]**

Reserved, Sign extended. Hardware and software must treat this field as RES0 if bit[48] is 0 or RES0, and as RES1 if bit[48] is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of bit [48], meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

**VA, bits [48:2]**

Bits[48:2] of the address value for comparison.

ARM deprecates setting DBGWVR<n>_EL1[2] == 1.

**Bits [1:0]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the DBGWVR<n>_EL1:

DBGWVR<n>_EL1[31:0] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x800 + 16n |

DBGWVR<n>_EL1[63:32] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x804 + 16n |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.10 EDAA32PFR, External Debug AArch32 Processor Feature Register

The EDAA32PFR characteristics are:

**Purpose**

Provides additional information about implemented PE features in AArch32.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDAA32PFR is in the Debug power domain.

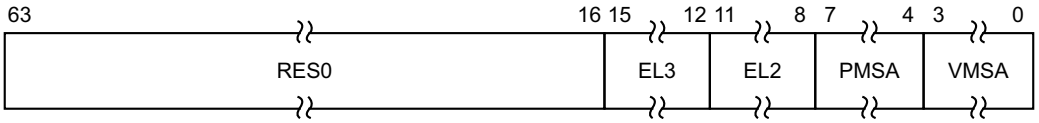
EDAA32PFR is only accessible in an AArch32-only implementation. If AArch64 is supported at any Exception level, EDAA32PFR is RES0.

**Attributes**

EDAA32PFR is a 64-bit register.

## Field descriptions

The EDAA32PFR bit assignments are:




**Bits [63:16]**

Reserved, RES0.

**EL3, bits [15:12]**

AArch32 EL3 Exception level handling. Only valid if EDPFR.EL3 is `0000`. Defined values are:

`0000` EL3 is not implemented.

`0001` EL3 can be executed in AArch32 state only.

All other values are reserved.

**Note**

EDPFR.{EL1, EL0} indicate whether EL1 and EL0 can only be executed in AArch32 state.

**EL2, bits [11:8]**

AArch32 EL2 Exception level handling. Only valid if EDPFR.EL2 is `0000`. Defined values are:

`0000` EL2 is not implemented.

`0001` EL2 can be executed in AArch32 state only.

All other values are reserved.

**Note**

EDPFR.{EL1, EL0} indicate whether EL1 and EL0 can only be executed in AArch32 state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**PMSA, bits [7:4]**

Indicates support for an ARMv8-R PMSA. Defined values are:

`0000` PMSA not supported.

`0100` Support for an ARMv8-R PMSAv8.

All other values are reserved. In ARMv8-A, the only permitted value is `0000`.

**VMSA, bits [3:0]**

Indicates support for an ARMv8-R VMSA. Defined values are:

`0000` VMSA not supported.

`0101` Support for an ARMv8-R VMSAv8-32.

All other values are reserved. In ARMv8-A, the only permitted value is `0000`.

### Accessing the EDAA32PFR:

EDAA32PFR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xD60` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.11 EDACR, External Debug Auxiliary Control Register

The EDACR characteristics are:

**Purpose**

Allows implementations to support IMPLEMENTATION DEFINED controls.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| IMP DEF | IMP DEF | IMP DEF | RO | RW |

**Configurations**

It is IMPLEMENTATION DEFINED whether EDACR is in the Core power domain or in the Debug power domain.

**Attributes**

EDACR is a 32-bit register.

#### Field descriptions

The EDACR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the EDACR:

EDACR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x094 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.12 EDCIDR0, External Debug Component Identification Register 0

The EDCIDR0 characteristics are:

**Purpose**

Provides information to identify an external debug component.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDCIDR0 is in the Debug power domain.

EDCIDR0 is optional to implement in the external register interface.

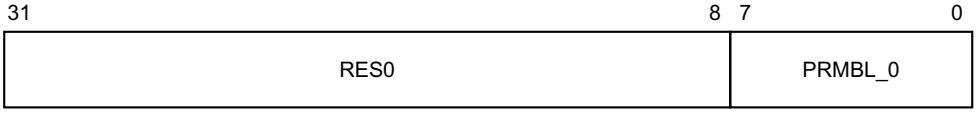
This register is required for CoreSight compliance.

**Attributes**

EDCIDR0 is a 32-bit register.

#### Field descriptions

The EDCIDR0 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PRMBL_0, bits [7:0]**

Preamble. Must read as `0x0D`.

#### Accessing the EDCIDR0:

EDCIDR0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xFF0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.13 EDCIDR1, External Debug Component Identification Register 1

The EDCIDR1 characteristics are:

**Purpose**

Provides information to identify an external debug component.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDCIDR1 is in the Debug power domain.

EDCIDR1 is optional to implement in the external register interface.

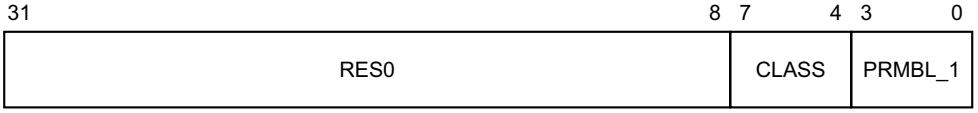
This register is required for CoreSight compliance.

**Attributes**

EDCIDR1 is a 32-bit register.

#### Field descriptions

The EDCIDR1 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**CLASS, bits [7:4]**

Component class. Reads as `0x9`, debug component.

**PRMBL_1, bits [3:0]**

Preamble. RAZ.

#### Accessing the EDCIDR1:

EDCIDR1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xFF4` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.14 EDCIDR2, External Debug Component Identification Register 2

The EDCIDR2 characteristics are:

**Purpose**

Provides information to identify an external debug component.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDCIDR2 is in the Debug power domain.

EDCIDR2 is optional to implement in the external register interface.

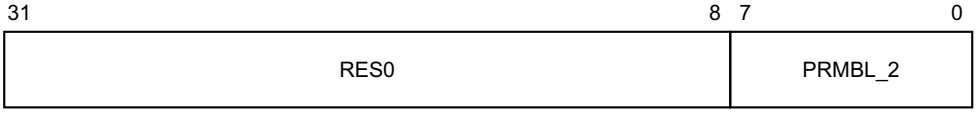
This register is required for CoreSight compliance.

**Attributes**

EDCIDR2 is a 32-bit register.

#### Field descriptions

The EDCIDR2 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PRMBL_2, bits [7:0]**

Preamble. Must read as `0x05`.

#### Accessing the EDCIDR2:

EDCIDR2 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xFF8` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.15 EDCIDR3, External Debug Component Identification Register 3

The EDCIDR3 characteristics are:

**Purpose**

Provides information to identify an external debug component.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDCIDR3 is in the Debug power domain.

EDCIDR3 is optional to implement in the external register interface.

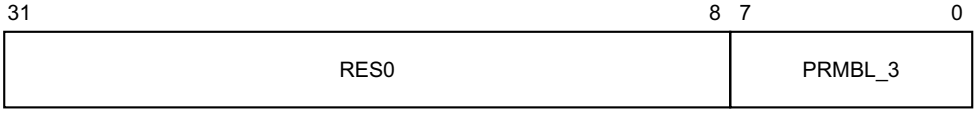
This register is required for CoreSight compliance.

**Attributes**

EDCIDR3 is a 32-bit register.

#### Field descriptions

The EDCIDR3 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PRMBL_3, bits [7:0]**

Preamble. Must read as `0xB1`.

#### Accessing the EDCIDR3:

EDCIDR3 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xFFC` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.16 EDCIDSR, External Debug Context ID Sample Register

The EDCIDSR characteristics are:

**Purpose**

Contains the sampled value of CONTEXTIDR_EL1, captured on reading the low half of EDPCSR.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | Default |
|---|---|---|---|
| Error | Error | Error | RO |

**Configurations**

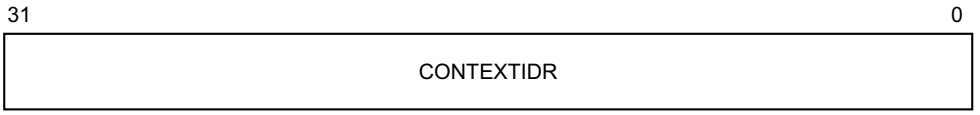
EDCIDSR is in the Core power domain.

**Attributes**

EDCIDSR is a 32-bit register.

#### Field descriptions

The EDCIDSR bit assignments are:




**CONTEXTIDR, bits [31:0]**

The sampled value of CONTEXTIDR_EL1, captured on reading the low half of EDPCSR.

If EL3 is implemented and using AArch32 then CONTEXTIDR is a Banked register, and EDCIDSR samples the current Banked copy of CONTEXTIDR.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

#### Accessing the EDCIDSR:

EDCIDSR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x0A4 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.17 EDDEVAFF0, External Debug Device Affinity register 0

The EDDEVAFF0 characteristics are:

**Purpose**

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDDEVAFF0 is in the Debug power domain.

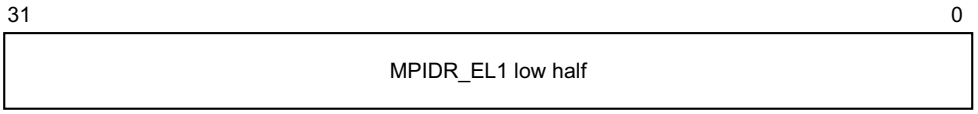
EDDEVAFF0 is optional to implement in the external register interface.

**Attributes**

EDDEVAFF0 is a 32-bit register.

#### Field descriptions

The EDDEVAFF0 bit assignments are:




**Bits [31:0]**

MPIDR_EL1 low half. Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.

#### Accessing the EDDEVAFF0:

EDDEVAFF0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xFA8 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.18 EDDEVAFF1, External Debug Device Affinity register 1

The EDDEVAFF1 characteristics are:

**Purpose**

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDDEVAFF1 is in the Debug power domain.

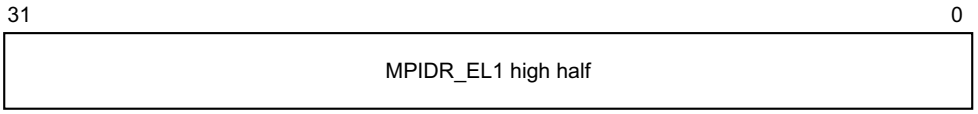
EDDEVAFF1 is optional to implement in the external register interface.

**Attributes**

EDDEVAFF1 is a 32-bit register.

#### Field descriptions

The EDDEVAFF1 bit assignments are:




**Bits [31:0]**

MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.

#### Accessing the EDDEVAFF1:

EDDEVAFF1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xFAC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.19 EDDEVARCH, External Debug Device Architecture register

The EDDEVARCH characteristics are:

**Purpose**

Identifies the programmers' model architecture of the external debug component.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDDEVARCH is in the Debug power domain.

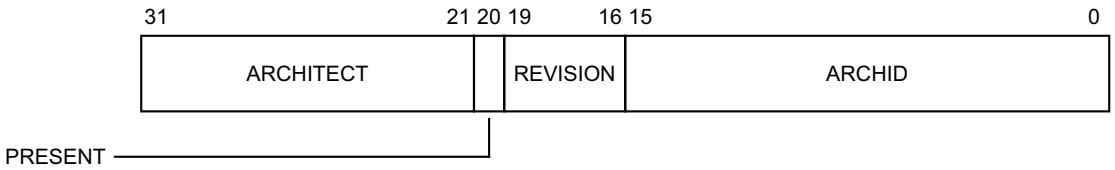
EDDEVARCH is optional to implement in the external register interface.

**Attributes**

EDDEVARCH is a 32-bit register.

## Field descriptions

The EDDEVARCH bit assignments are:




**ARCHITECT, bits [31:21]**

Defines the architecture of the component. For debug, this is ARM Limited.

Bits [31:28] are the JEP 106 continuation code, `0x4`.

Bits [27:21] are the JEP 106 ID code, `0x3B`.

**PRESENT, bit [20]**

When set to 1, indicates that the DEVARCH is present.

This field is 1 in ARMv8.

**REVISION, bits [19:16]**

Defines the architecture revision. For architectures defined by ARM this is the minor revision.

For debug, the revision defined by ARMv8 is `0x0`.

All other values are reserved.

**ARCHID, bits [15:0]**

Defines this part to be an ARMv8 debug component. For architectures defined by ARM this is further subdivided.

For debug:

- Bits [15:12] are the architecture version, `0x6`.
- Bits [11:0] are the architecture part number, `0xA15`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This corresponds to the ARMv8 debug architecture version.

### Accessing the EDDEVARCH:

EDDEVARCH can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xFBC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H9.2.20 EDDEVID, External Debug Device ID register 0

The EDDEVID characteristics are:

**Purpose**

Provides extra information for external debuggers about features of the debug implementation.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

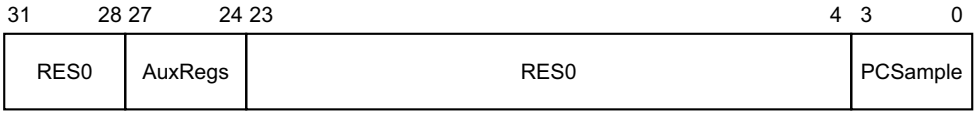
EDDEVID is in the Debug power domain.

**Attributes**

EDDEVID is a 32-bit register.

### Field descriptions

The EDDEVID bit assignments are:




**Bits [31:28]**

Reserved, RES0.

**AuxRegs, bits [27:24]**

Indicates support for Auxiliary registers. Permitted values for this field are:

`0000` None supported.

`0001` Support for External Debug Auxiliary Control Register, EDACR.

All other values are reserved.

**Bits [23:4]**

Reserved, RES0.

**PCSample, bits [3:0]**

Indicates the level of Sample-based profiling support using external debug registers 40 through 43. Permitted values of this field in ARMv8 are:

`0000` Architecture-defined form of Sample-based profiling not implemented.

`0010` EDPCSR and EDCIDSR are implemented (only permitted if EL3 and EL2 are not implemented).

`0011` EDPCSR, EDCIDSR, and EDVIDSR are implemented.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the EDDEVID:

EDDEVID can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xFC8 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.21 EDDEVID1, External Debug Device ID register 1

The EDDEVID1 characteristics are:

**Purpose**

Provides extra information for external debuggers about features of the debug implementation.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

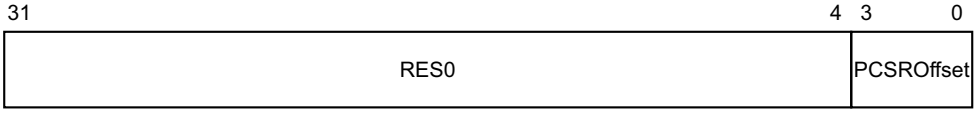
EDDEVID1 is in the Debug power domain.

**Attributes**

EDDEVID1 is a 32-bit register.

#### Field descriptions

The EDDEVID1 bit assignments are:




**Bits [31:4]**

Reserved, RES0.

**PCSROffset, bits [3:0]**

This field indicates the offset applied to PC samples returned by reads of EDPCSR. Permitted values of this field in ARMv8 are:

`0000` EDPCSR not implemented.

`0010` EDPCSR implemented, and samples have no offset applied and do not sample the instruction set state in AArch32 state.

#### Accessing the EDDEVID1:

EDDEVID1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xFC4` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.22 EDDEVID2, External Debug Device ID register 2

The EDDEVID2 characteristics are:

**Purpose**

Reserved for future descriptions of features of the debug implementation.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

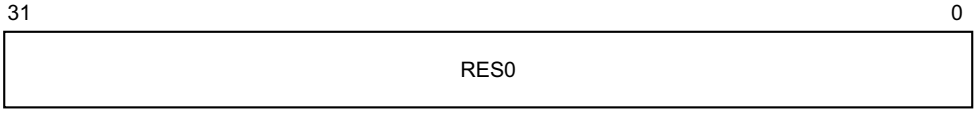
EDDEVID2 is in the Debug power domain.

**Attributes**

EDDEVID2 is a 32-bit register.

#### Field descriptions

The EDDEVID2 bit assignments are:




**Bits [31:0]**

Reserved, RES0.

#### Accessing the EDDEVID2:

EDDEVID2 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xFC0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.23 EDDEVTYPE, External Debug Device Type register

The EDDEVTYPE characteristics are:

**Purpose**

Indicates to a debugger that this component is part of a PEs debug logic.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDDEVTYPE is in the Debug power domain.

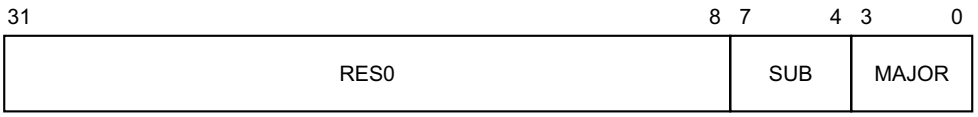
EDDEVTYPE is optional to implement in the external register interface.

**Attributes**

EDDEVTYPE is a 32-bit register.

#### Field descriptions

The EDDEVTYPE bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**SUB, bits [7:4]**

Subtype. Must read as `0x1` to indicate this is a component within a PE.

**MAJOR, bits [3:0]**

Major type. Must read as `0x5` to indicate this is a debug logic component.

#### Accessing the EDDEVTYPE:

EDDEVTYPE can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xFCC` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.24 EDDFR, External Debug Feature Register

The EDDFR characteristics are:

**Purpose**

Provides top level information about the debug system in AArch64.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

Debuggers must use EDDEVARCH to determine the Debug architecture version.

**Configurations**

External register EDDFR is architecturally mapped to AArch64 System register ID_AA64DFR0_EL1.

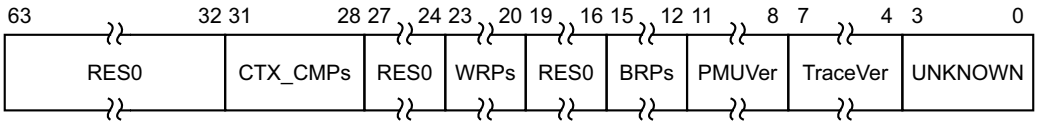
EDDFR is in the Debug power domain.

**Attributes**

EDDFR is a 64-bit register.

#### Field descriptions

The EDDFR bit assignments are:




**Bits [63:32]**

Reserved, RES0.

**CTX_CMPs, bits [31:28]**

Number of breakpoints that are context-aware, minus 1. These are the highest numbered breakpoints.

**Bits [27:24]**

Reserved, RES0.

**WRPs, bits [23:20]**

Number of watchpoints, minus 1. The value of `0b0000` is reserved.

**Bits [19:16]**

Reserved, RES0.

**BRPs, bits [15:12]**

Number of breakpoints, minus 1. The value of `0b0000` is reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**PMUVer, bits [11:8]**

Performance Monitors extension version. Indicates whether System register interface to Performance Monitors extension is implemented. Defined values are:

`0000` Performance Monitors extension System registers not implemented.

`0001` Performance Monitors extension System registers implemented, PMUv3.

`1111` IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported.

All other values are reserved.

**TraceVer, bits [7:4]**

Trace support. Indicates whether System register interface to a trace macrocell is implemented. Defined values are:

`0000` Trace macrocell System registers not implemented.

`0001` Trace macrocell System registers implemented.

All other values are reserved.

A value of `0000` only indicates that no System register interface to a trace macrocell is implemented. A trace macrocell might nevertheless be implemented without a System register interface.

**UNKNOWN, bits [3:0]**

Reserved, UNKNOWN.

### Accessing the EDDFR:

EDDFR[31:0] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xD28` |

EDDFR[63:32] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xD2C` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.25 EDECCR, External Debug Exception Catch Control Register

The EDECCR characteristics are:

**Purpose**

Controls exception catch debug events.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| Error | Error | Error | RO | RW |

**Configurations**

External register EDECCR is architecturally mapped to AArch64 System register OSECCR_EL1.

External register EDECCR is architecturally mapped to AArch32 System register DBGOSECCR.

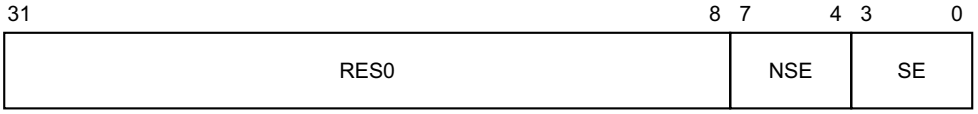
EDECCR is in the Core power domain.

**Attributes**

EDECCR is a 32-bit register.

## Field descriptions

The EDECCR bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**NSE, bits [7:4]**

Coarse-grained Non-secure exception catch. Possible values of this field are:

`0000` Exception catch debug event disabled for Non-secure Exception levels.

`0010` Exception catch debug event enabled for Non-secure EL1.

`0100` Exception catch debug event enabled for Non-secure EL2.

`0110` Exception catch debug event enabled for Non-secure EL1 and EL2.

All other values are reserved. Bits [7,4] are reserved, RES0.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Cold reset.

**SE, bits [3:0]**

Coarse-grained Secure exception catch. Possible values of this field are:

`0000` Exception catch debug event disabled for Secure Exception levels.

`0010` Exception catch debug event enabled for Secure EL1.

`1000` Exception catch debug event enabled for Secure EL3.

`1010` Exception catch debug event enabled for Secure EL1 and EL3.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

All other values are reserved. Bits [2,0] are reserved. RES0. Ignored if ExternalSecureInvasiveDebugEnabled() == FALSE.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Cold reset.

### Accessing the EDECCR:

EDECCR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0x098` |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.2.26 EDECR, External Debug Execution Control Register

The EDECR characteristics are:

**Purpose**

Controls Halting debug events.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RW |

**Configurations**

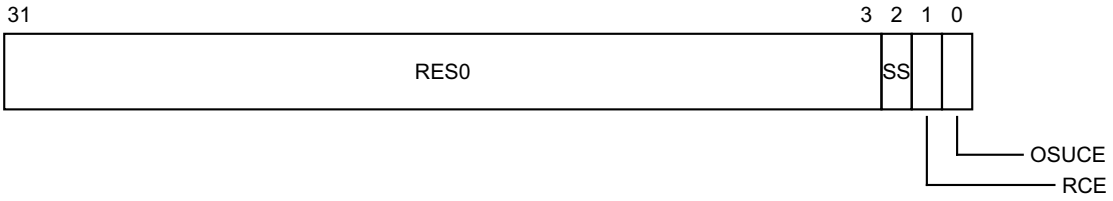
EDECR is in the Debug power domain.

**Attributes**

EDECR is a 32-bit register.

#### Field descriptions

The EDECR bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**SS, bit [2]**

Halting step enable. Possible values of this field are:

`0` Halting step debug event disabled.

`1` Halting step debug event enabled.

If the value of EDECR.SS is changed when the PE is in Non-debug state, the resulting value of EDECR.SS is UNKNOWN.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on External debug reset.

**RCE, bit [1]**

Reset catch enable. Possible values of this field are:

`0` Reset catch debug event disabled.

`1` Reset catch debug event enabled.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on External debug reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**OSUCE, bit [0]**

OS unlock catch enabled. Possible values of this field are:

`0` OS unlock catch debug event disabled.

`1` OS unlock catch debug event enabled.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on External debug reset.

### Accessing the EDECR:

EDECR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0x024` |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.2.27 EDESR, External Debug Event Status Register

The EDESR characteristics are:

**Purpose**

Indicates the status of internally pending Halting debug events.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | SLK | Default |
|---|---|---|---|
| Error | Error | RO | RW |

If a request to clear a pending Halting debug event is received at or about the time when halting becomes allowed, it is CONSTRAINED UNPREDICTABLE whether the event is taken.

If Core power is removed while a Halting debug event is pending, it is lost. However, it may become pending again when the Core is powered back on and Cold reset.

**Configurations**

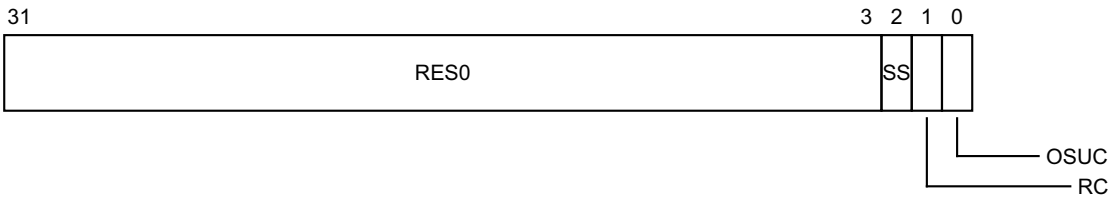
EDESR is in the Core power domain.

**Attributes**

EDESR is a 32-bit register.

## Field descriptions

The EDESR bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**SS, bit [2]**

Halting step debug event pending. Possible values of this field are:

`0` Reading this means that a Halting step debug event is not pending. Writing this means no action.

`1` Reading this means that a Halting step debug event is pending. Writing this clears the pending Halting step debug event.

When this register has an architecturally-defined reset value, this field resets to the value of EDECR.SS.

This field resets to its defined reset value on Warm reset.

**RC, bit [1]**

Reset catch debug event pending. Possible values of this field are:

`0` Reading this means that a Reset catch debug event is not pending. Writing this means no action.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`1` Reading this means that a Reset catch debug event is pending. Writing this clears the pending Reset catch debug event.

When this register has an architecturally-defined reset value, this field resets to the value of EDECR.RCE.

This field resets to its defined reset value on Warm reset.

**OSUC, bit [0]**

OS unlock debug event pending. Possible values of this field are:

`0` Reading this means that an OS unlock catch debug event is not pending. Writing this means no action.

`1` Reading this means that an OS unlock catch debug event is pending. Writing this clears the pending OS unlock catch debug event.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

### Accessing the EDESR:

EDESR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0x020` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.28 EDITCTRL, External Debug Integration mode Control register

The EDITCTRL characteristics are:

**Purpose**

Enables the external debug to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | Default |
|---|---|---|---|
| IMP DEF | IMP DEF | IMP DEF | RW |

**Configurations**

It is IMPLEMENTATION DEFINED whether EDITCTRL is in the Core power domain or in the Debug power domain.

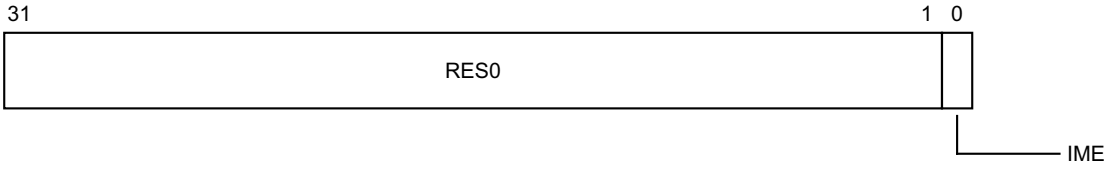
EDITCTRL is optional to implement in the external register interface.

**Attributes**

EDITCTRL is a 32-bit register.

## Field descriptions

The EDITCTRL bit assignments are:




**Bits [31:1]**

Reserved, RES0.

**IME, bit [0]**

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection. The integration mode behavior is IMPLEMENTATION DEFINED.

`0` Normal operation.

`1` Integration mode enabled.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on IMPLEMENTATION DEFINED reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the EDITCTRL:

EDITCTRL can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xF00` |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.2.29 EDITR, External Debug Instruction Transfer Register

The EDITR characteristics are:

**Purpose**

Used in Debug state for passing instructions to the PE for execution.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| Error | Error | Error | WI | WO |

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any instruction issued through the ITR in Normal access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

**Configurations**

EDITR is in the Core power domain.

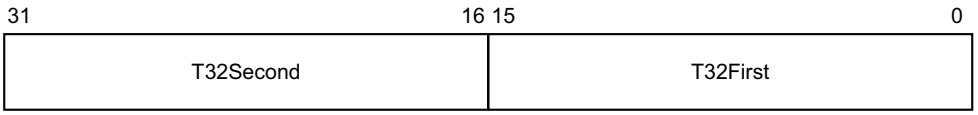
**Attributes**

EDITR is a 32-bit register.

#### Field descriptions

The EDITR bit assignments are:

***When in AArch32 state:***



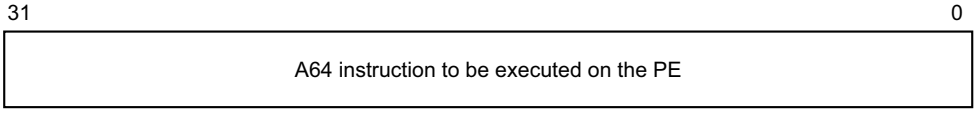

**T32Second, bits [31:16]**

Second halfword of the T32 instruction to be executed on the PE.

**T32First, bits [15:0]**

First halfword of the T32 instruction to be executed on the PE.

***When in AArch64 state:***




**Bits [31:0]**

A64 instruction to be executed on the PE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the EDITR:

EDITR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x084 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.2.30 EDLAR, External Debug Lock Access Register

The EDLAR characteristics are:

**Purpose**

Allows or disallows access to the external debug registers through a memory-mapped interface.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| WO |

**Configurations**

EDLAR is in the Debug power domain.

If OPTIONAL memory-mapped access to the external debug interface is supported then an OPTIONAL Software lock can be implemented as part of CoreSight compliance.

EDLAR ignores writes if the Software lock is not implemented and ignores writes for other accesses to the external debug interface.

The Software lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

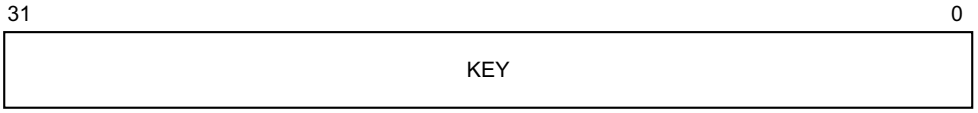
Software uses EDLAR to set or clear the lock, and EDLSR to check the current status of the lock.

**Attributes**

EDLAR is a 32-bit register.

#### Field descriptions

The EDLAR bit assignments are:




**KEY, bits [31:0]**

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

#### Accessing the EDLAR:

EDLAR can be accessed through the memory-mapped interface:

| Component | Offset |
|---|---|
| Debug | `0xFB0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H9.2.31 EDLSR, External Debug Lock Status Register

The EDLSR characteristics are:

**Purpose**

Indicates the current status of the software lock for external debug registers.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDLSR is in the Debug power domain.

If OPTIONAL memory-mapped access to the external debug interface is supported then an OPTIONAL Software lock can be implemented as part of CoreSight compliance.

EDLSR is RAZ if the Software lock is not implemented and is RAZ for other accesses to the external debug interface.

The Software lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

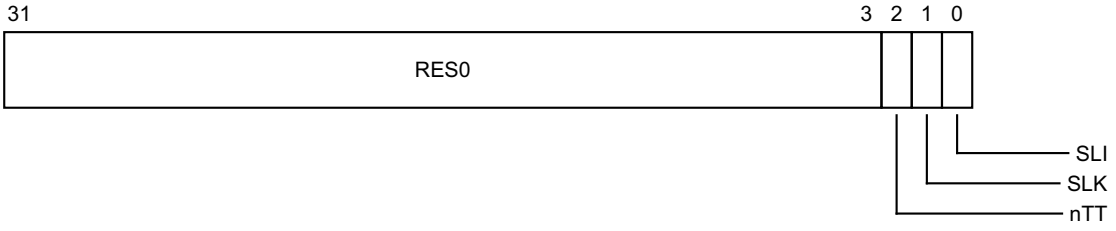
Software uses EDLAR to set or clear the lock, and EDLSR to check the current status of the lock.

**Attributes**

EDLSR is a 32-bit register.

### Field descriptions

The EDLSR bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**nTT, bit [2]**

Not thirty-two bit access required. RAZ.

**SLK, bit [1]**

Software lock status for this component. For an access to LSR that is not a memory-mapped access, or when the software lock is not implemented, this field is RES0.

For memory-mapped accesses when the software lock is implemented, possible values of this field are:

`0` Lock clear. Writes are permitted to this component's registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 Lock set. Writes to this component's registers are ignored, and reads have no side effects.

When this register has an architecturally-defined reset value, this field resets to 1.

This field resets to its defined reset value on External debug reset.

**SLI, bit [0]**

Software lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ. For memory-mapped accesses, the value of this field is IMPLEMENTATION DEFINED. Permitted values are:

0 Software lock not implemented or not memory-mapped access.

1 Software lock implemented and memory-mapped access.

### Accessing the EDLSR:

EDLSR can be accessed through the memory-mapped interface:

| Component | Offset |
|---|---|
| Debug | 0xFB4 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H9.2.32 EDPCSR, External Debug Program Counter Sample Register

The EDPCSR characteristics are:

**Purpose**

Holds a sampled instruction address value.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| Error | Error | Error | RO | RO |

**Configurations**

EDPCSR is in the Core power domain.

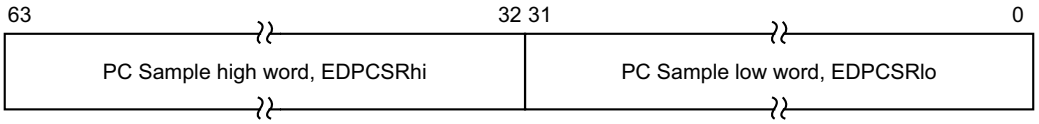
EDPCSR is optional to implement in the external register interface.

**Attributes**

EDPCSR is a 64-bit register.

### Field descriptions

The EDPCSR bit assignments are:




**Bits [63:32]**

PC Sample high word, EDPCSRhi. If EDVIDSR.HV == 0 then this field is RAZ, otherwise bits [63:32] of the sampled PC.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [31:0]**

PC Sample low word, EDPCSRlo. Bits [31:0] of the sampled instruction address value. Reading EDPCSRlo has the side-effect of updating EDCIDSR, EDVIDSR, and EDPCSRhi. However:

- If the PE is in Debug state, or Sample-based profiling is prohibited, EDPCSRlo reads as `0xFFFFFFFF` and EDCIDSR, EDVIDSR, and EDPCSRhi become UNKNOWN.
- If the PE is in Reset state, the sampled value is unknown and EDCIDSR, EDVIDSR and EDPCSRhi become UNKNOWN.
- If no instruction has been retired since the PE left Reset state, Debug state, or a state where Non-invasive debug is not permitted, the sampled value is UNKNOWN and EDCIDSR, EDVIDSR, and EDPCSRhi become UNKNOWN.
- For a read of EDPCSRlo from the memory-mapped interface, if EDLSR.SLK == 1, meaning the Software Lock is locked, then the access has no side-effects. That is, EDCIDSR, EDVIDSR, and EDPCSRhi are unchanged.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the EDPCSR:

EDPCSR[31:0] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x0A0 |

EDPCSR[63:32] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x0AC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.33 EDPFR, External Debug Processor Feature Register

The EDPFR characteristics are:

**Purpose**

Provides additional information about implemented PE features in AArch64.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

External register EDPFR is architecturally mapped to AArch64 System register ID_AA64PFR0_EL1.

EDPFR is in the Debug power domain.

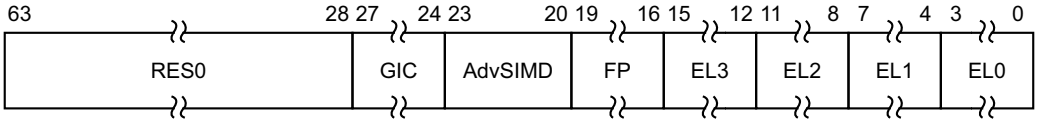
If AArch64 is supported at any Exception level, bits [27:0] take the value of the System register ID_AA64PFR0_EL1[27:0]. Otherwise, bits [27:16] have the same definition as ID_AA64PFR0_EL1, even though AArch64 is not implemented, and bits [15:0] are zero.

**Attributes**

EDPFR is a 64-bit register.

## Field descriptions

The EDPFR bit assignments are:




**Bits [63:28]**

Reserved, RES0.

**GIC, bits [27:24]**

System register GIC interface support. Defined values are:

0000 No System register interface to the GIC is supported.

0001 System register interface to the GIC CPU interface is supported.

All other values are reserved.

**AdvSIMD, bits [23:20]**

Advanced SIMD. Defined values are:

0000 Advanced SIMD is implemented.

1111 Advanced SIMD is not implemented.

All other values are reserved.

**FP, bits [19:16]**

Floating-point. Defined values are:

0000 Floating-point is implemented.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1111 Floating-point is not implemented.

All other values are reserved.

**EL3, bits [15:12]**

AArch64 EL3 Exception level handling. Defined values are:

0000 EL3 is not implemented or cannot be executed in AArch64 state.

0001 EL3 can be executed in AArch64 state only.

0010 EL3 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

**EL2, bits [11:8]**

AArch64 EL2 Exception level handling. Defined values are:

0000 EL2 is not implemented or cannot be executed in AArch64 state.

0001 EL2 can be executed in AArch64 state only.

0010 EL2 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

**EL1, bits [7:4]**

AArch64 EL1 Exception level handling. Defined values are:

0000 EL1 can be executed in AArch32 state only.

0001 EL1 can be executed in AArch64 state only.

0010 EL1 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

**EL0, bits [3:0]**

AArch64 EL0 Exception level handling. Defined values are:

0000 EL0 can be executed in AArch32 state only.

0001 EL0 can be executed in AArch64 state only.

0010 EL0 can be executed in either AArch64 or AArch32 state.

All other values are reserved.

### Accessing the EDPFR:

EDPFR[31:0] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xD20 |

EDPFR[63:32] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xD24 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.34 EDPIDR0, External Debug Peripheral Identification Register 0

The EDPIDR0 characteristics are:

**Purpose**

Provides information to identify an external debug component.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDPIDR0 is in the Debug power domain.

EDPIDR0 is optional to implement in the external register interface.

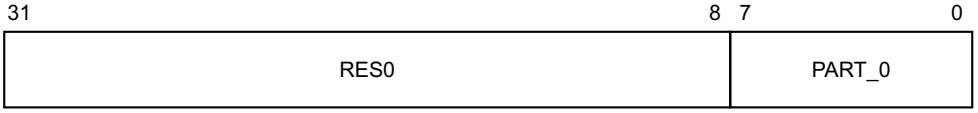
This register is required for CoreSight compliance.

**Attributes**

EDPIDR0 is a 32-bit register.

#### Field descriptions

The EDPIDR0 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PART_0, bits [7:0]**

Part number, least significant byte.

#### Accessing the EDPIDR0:

EDPIDR0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xFE0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.35 EDPIDR1, External Debug Peripheral Identification Register 1

The EDPIDR1 characteristics are:

**Purpose**

Provides information to identify an external debug component.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDPIDR1 is in the Debug power domain.

EDPIDR1 is optional to implement in the external register interface.

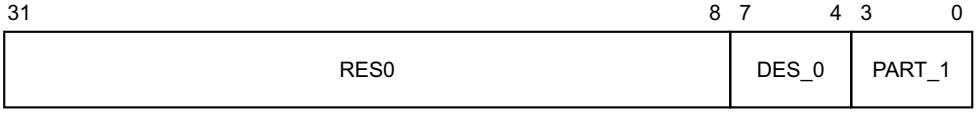
This register is required for CoreSight compliance.

**Attributes**

EDPIDR1 is a 32-bit register.

#### Field descriptions

The EDPIDR1 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**DES_0, bits [7:4]**

Designer, least significant nibble of JEP106 ID code. For ARM Limited, this field is `0b1011`.

**PART_1, bits [3:0]**

Part number, most significant nibble.

#### Accessing the EDPIDR1:

EDPIDR1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xFE4` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.36 EDPIDR2, External Debug Peripheral Identification Register 2

The EDPIDR2 characteristics are:

**Purpose**

Provides information to identify an external debug component.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDPIDR2 is in the Debug power domain.

EDPIDR2 is optional to implement in the external register interface.

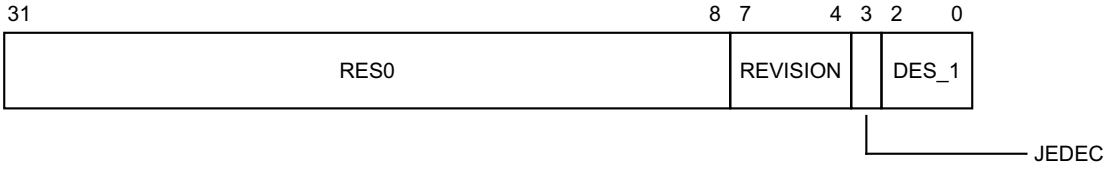
This register is required for CoreSight compliance.

**Attributes**

EDPIDR2 is a 32-bit register.

#### Field descriptions

The EDPIDR2 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**REVISION, bits [7:4]**

Part major revision. Parts can also use this field to extend Part number to 16-bits.

**JEDEC, bit [3]**

RAO. Indicates a JEP106 identity code is used.

**DES_1, bits [2:0]**

Designer, most significant bits of JEP106 ID code. For ARM Limited, this field is `0b011`.

#### Accessing the EDPIDR2:

EDPIDR2 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xFE8` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.37 EDPIDR3, External Debug Peripheral Identification Register 3

The EDPIDR3 characteristics are:

**Purpose**

Provides information to identify an external debug component.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDPIDR3 is in the Debug power domain.

EDPIDR3 is optional to implement in the external register interface.

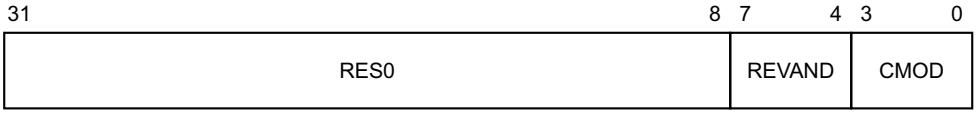
This register is required for CoreSight compliance.

**Attributes**

EDPIDR3 is a 32-bit register.

#### Field descriptions

The EDPIDR3 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**REVAND, bits [7:4]**

Part minor revision. Parts using EDPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

**CMOD, bits [3:0]**

Customer modified. Indicates someone other than the Designer has modified the component.

#### Accessing the EDPIDR3:

EDPIDR3 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xFEC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.38 EDPIDR4, External Debug Peripheral Identification Register 4

The EDPIDR4 characteristics are:

**Purpose**

Provides information to identify an external debug component.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

EDPIDR4 is in the Debug power domain.

EDPIDR4 is optional to implement in the external register interface.

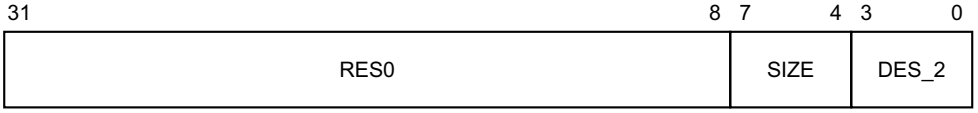
This register is required for CoreSight compliance.

**Attributes**

EDPIDR4 is a 32-bit register.

#### Field descriptions

The EDPIDR4 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**SIZE, bits [7:4]**

Size of the component. RAZ. $Log_2$ of the number of 4KB pages from the start of the component to the end of the component ID registers.

**DES_2, bits [3:0]**

Designer, JEP106 continuation code, least significant nibble. For ARM Limited, this field is `0b0100`.

#### Accessing the EDPIDR4:

EDPIDR4 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0xFD0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.39 EDPRCR, External Debug Power/Reset Control Register

The EDPRCR characteristics are:

**Purpose**

Controls the PE functionality related to powerup, reset, and powerdown.

**Usage constraints**

Accessing this register depends on which field is being accessed; see the register field descriptions for the states that they are accessible in.

**Configurations**

EDPRCR contains fields that are in the Core power domain and fields that are in the Debug power domain.

Bit [0] of this register is mapped to DBGPRCR.CORENPDRQ, bit [0] of the AArch32 view of this register.

Bit [0] of this register is mapped to DBGPRCR_EL1.CORENPDRQ, bit [0] of the AArch64 view of this register.

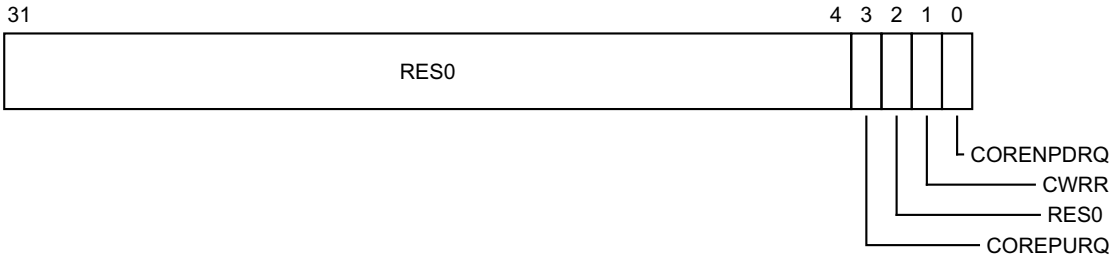
The other bits in these registers are not mapped to each other.

**Attributes**

EDPRCR is a 32-bit register.

#### Field descriptions

The EDPRCR bit assignments are:




**Bits [31:4]**

Reserved, RES0.

**COREPURQ, bit [3]**

Core powerup request. Allows a debugger to request that the power controller power up the core, enabling access to the debug register in the Core power domain. The actions on writing to this bit are:

`0` Do not request power up of the Core power domain.

`1` Request power up of the Core power domain, and emulation of powerdown.

In an implementation that includes the recommended external debug interface, this bit drives the DBGPWRUPREQ signal.

This bit can be read and written when the Core power domain is powered off.

The power controller must not allow the Core power domain to switch off while this bit is one.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on External debug reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is accessible as shown below:

| SLK | Default |
|---|---|
| RO | RW |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**Bit [2]**

Reserved, RES0.

**CWRR, bit [1]**

Warm reset request. Write only bit that reads as zero. The actions on writing to this bit are:

`0` No action.

`1` Request Warm reset.

The PE ignores writes to this bit if any of the following are the case:

- ExternalInvasiveDebugEnabled() == FALSE, EL3 is not implemented, and the implemented Security state is Non-Secure state.
- ExternalSecureInvasiveDebugEnabled() == FALSE and one of the following is true:
  - EL3 is implemented.
  - The implemented Security state is Secure state.
- The Core power domain is either completely off or in a low-power state where the Core power domain registers cannot be accessed.
- DoubleLockStatus() == TRUE (OS Double Lock is set).
- OSLSR.OSLK == 1 (OS lock is locked).

In an implementation that includes the recommended external debug interface, this bit drives the DBGRSTREQ signal.

When a write to this bit initiates a Warm Reset, the extent of the reset is IMPLEMENTATION DEFINED, but must be one of:

- The request is ignored.
- Only this PE is Warm reset.
- This PE and other components of the system, possibly including other PEs, are Warm reset.

**Note**

Although the ARM architecture permits the first option from the above list, ARM recommends that either of the other options is implemented.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Warm reset.

This field is accessible as shown below:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| WI | WI | WI | WI | WO |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**CORENPDRQ, bit [0]**

Core no powerdown request. Requests emulation of powerdown. Possible values of this bit are:

`0` On a powerdown request, the system powers down the Core power domain.

`1` On a powerdown request, the system emulates powerdown of the Core power domain. In this emulation mode the Core power domain is not actually powered down.

This bit is UNKNOWN, and the PE ignores writes to this bit if any of the following are the case:

- The Core power domain is either completely off or in a low-power state where the Core power domain registers cannot be accessed.
- DoubleLockStatus() == TRUE (OS Double Lock is set).
- OSLSR.OSLK == 1 (OS lock is locked).

When this register has an architecturally-defined reset value, this field resets to the value of EDPRCR.COREPURQ.

This field resets to its defined reset value on Cold reset.

This field is accessible as shown below:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| WI | WI | WI | RO | RW |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

## Accessing the EDPRCR:

EDPRCR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0x310` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H9.2.40 EDPRSR, External Debug Processor Status Register

The EDPRSR characteristics are:

**Purpose**

Holds information about the reset and powerdown state of the PE.

**Usage constraints**

Accessing this register depends on which field is being accessed; see the register field descriptions for the states that they are accessible in.

If the Core power domain is powered up (EDPRSR.PU == 1), then following a read of EDPRSR:

- If EDPRSR.DLK == 0, then:
  - EDPRSR.{SDR, SPMAD, SDAD, SPD} are cleared to 0.
  - EDPRSR.SR is cleared to 0 if the non-debug logic of the PE is not in reset state (EDPRSR.R == 0).
- Otherwise it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

If the Core power domain is powered down (EDPRSR.PU == 0), then:

- EDPRSR.{SDR, SPMAD, SDAD, SR} are all UNKNOWN, and are either reset or restored on being powered up.
- EDPRSR.SPD is not cleared following a read of EDPRSR. See the SPD bit description for more information.

**Configurations**

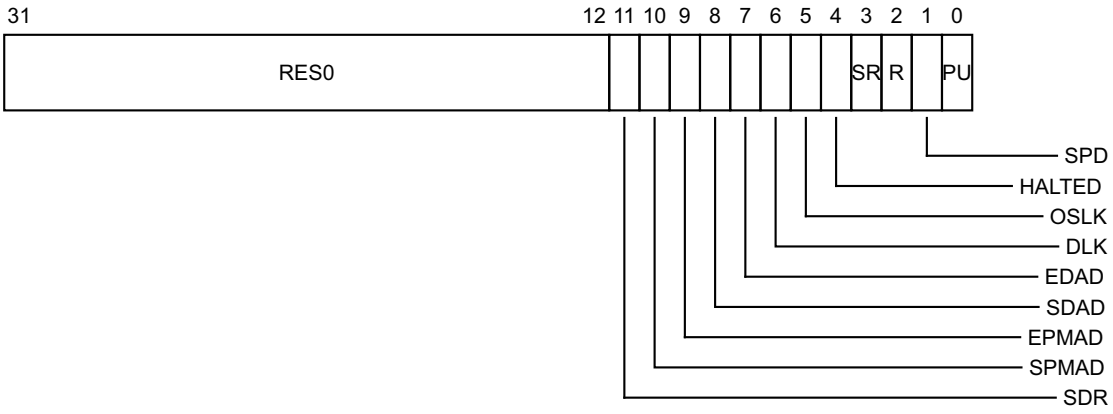
EDPRSR contains fields that are in the Core power domain and fields that are in the Debug power domain.

**Attributes**

EDPRSR is a 32-bit register.

### Field descriptions

The EDPRSR bit assignments are:




**Bits [31:12]**

Reserved, RES0.

**SDR, bit [11]**

Sticky debug restart. Set to 1 when the PE exits Debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit is UNKNOWN on reads if any of the following are true:

- DoubleLockStatus() == TRUE (EDPRSR.DLK == 1). The OS double-lock is locked.
- EDPRSR.R == 1. The PE is in Reset state.
- CorePoweredUp() == FALSE (EDPRSR.PU == 0). The Core power domain is powered down.

Otherwise permitted values are:

`0` The PE has not restarted since EDPRSR was last read.

`1` The PE has restarted since EDPRSR was last read.

**Note**

If a reset occurs when the PE is in Debug state, the PE exits Debug state. SDR is UNKNOWN on Warm reset, meaning a debugger must also use the SR bit to infer the PE has left Debug state in this case.

If DoubleLockStatus() == FALSE then following a read of EDPRSR, this bit clears to 0. Otherwise it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

This field is accessible as shown below:

| Off | DLK | SLK | Default |
|---|---|---|---|
| UNK | UNK | RO | RC |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**SPMAD, bit [10]**

Sticky EPMAD error. Set to 1 if an external debug interface access to a Performance Monitors register returns an error because AllowExternalPMUAccess() == FALSE.

This bit is UNKNOWN on reads if any of EDPRSR.{DLK, OSLK, R} is 1, or EDPRSR.PU is 0.

Otherwise permitted values are:

`0` No accesses to the external Performance Monitors registers have failed since EDPRSR was last read.

`1` At least one access to the external Performance Monitors registers has failed since EDPRSR was last read.

If DoubleLockStatus() == FALSE then following a read of EDPRSR, this bit clears to 0. Otherwise it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Cold reset.

This field is accessible as shown below:

| Off | DLK | SLK | Default |
|---|---|---|---|
| UNK | UNK | RO | RC |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EPMAD, bit [9]**

External Performance Monitors access disable status.

This bit is UNKNOWN on reads if any of EDPRSR.{DLK, OSLK, R} is 1, or EDPRSR.PU is 0.

Otherwise permitted values are:

`0` External Performance Monitors access enabled. AllowExternalPMUAccess() == TRUE.

`1` External Performance Monitors access disabled. AllowExternalPMUAccess() == FALSE.

If external performance monitors access is not implemented, EPMAD is RAO.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

This field is accessible as shown below:

| Off | DLK | Default |
|---|---|---|
| UNK | UNK | RO |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**SDAD, bit [8]**

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because AllowExternalDebugAccess() == FALSE.

This bit is UNKNOWN on reads if any of the following are true:

- Either of EDPRSR.{DLK, R} is 1.
- EDPRSR.PU is 0.
- EDPRSR.OSLK is 1 and external debug writes to OSLAR_EL1 do not return an error when AllowExternalDebugAccess() == FALSE.

Otherwise permitted values are:

`0` No accesses to the external debug registers have failed since EDPRSR was last read.

`1` At least one access to the external debug registers has failed since EDPRSR was last read.

If DoubleLockStatus() == FALSE then following a read of EDPRSR, this bit clears to 0. Otherwise it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Cold reset.

This field is accessible as shown below:

| Off | DLK | SLK | Default |
|---|---|---|---|
| UNK | UNK | RO | RC |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**EDAD, bit [7]**

External debug access disable status.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This bit is UNKNOWN on reads if any of the following are true:

- Either of EDPRSR.{DLK, R} is 1.
- EDPRSR.PU is 0.
- EDPRSR.OSLK is 1 and external debug writes to OSLAR_EL1 do not return an error when AllowExternalDebugAccess() == FALSE.

Otherwise permitted values are:

`0` External debug access enabled. AllowExternalDebugAccess() == TRUE.

`1` External debug access disabled. AllowExternalDebugAccess() == FALSE.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

This field is accessible as shown below:

| Off | DLK | EDAD | Default |
|---|---|---|---|
| UNK | UNK | RAO | RAZ |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**DLK, bit [6]**

OS Double Lock status bit. Returns the result of the pseudocode function DoubleLockStatus().

This bit is UNKNOWN on reads if EDPRSR.PU is 0.

Otherwise reads as zero if any of the following are true, that is when DoubleLockStatus() == FALSE:

- OSDLR_EL1.DLK == 0.
- DBGPRCR_EL1.CORENPDRQ == 1.
- The PE is in Debug state.

If the Core power domain is powered up and DoubleLockStatus() == TRUE, it is IMPLEMENTATION DEFINED whether:

- EDPRSR.PU reads as 0, EDPRSR.DLK reads as 1, and EDPRSR.SPD is UNKNOWN.
- EDPRSR.PU reads as 1, EDPRSR.DLK is UNKNOWN, and EDPRSR.SPD reads as 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

This field is accessible as shown below:

| Off | DLK | Default |
|---|---|---|
| UNK | RAO | RAZ |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**OSLK, bit [5]**

OS lock status bit.

This bit is UNKNOWN on reads if either of EDPRSR.{DLK, R} is 1 or EDPRSR.PU is 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

A read of this bit returns the value of OSLSR_EL1.OSLK.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

This field is accessible as shown below:

| Off | DLK | OSLK | Default |
|---|---|---|---|
| UNK | UNK | RAO | RAZ |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**HALTED, bit [4]**

Halted status bit.

This bit is UNKNOWN on reads if EDPRSR.PU is 0.

Otherwise permitted values are:

`0` EDSCR.STATUS is `0b000010` (PE in Non-debug state).

`1` EDSCR.STATUS is not `0b000010`.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

This field is accessible as shown below:

| Off | DLK | Default |
|---|---|---|
| UNK | RAZ | RO |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**SR, bit [3]**

Sticky core reset status bit.

This bit is UNKNOWN on reads if EDPRSR.DLK is 1 or EDPRSR.PU is 0.

Otherwise permitted values are:

`0` The non-debug logic of the PE is not in reset state and has not been reset since the last time EDPRSR was read.

`1` The non-debug logic of the PE is in reset state or has been reset since the last time EDPRSR was read.

If DoubleLockStatus() == FALSE then following a read of EDPRSR, this bit clears to 0 if the non-debug logic of the PE is not in reset state. Otherwise it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

When this register has an architecturally-defined reset value, this field resets to `1`.

This field resets to its defined reset value on Warm reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field is accessible as shown below:

| Off | DLK | SLK | Default |
|---|---|---|---|
| UNK | UNK | RO | RC |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**R, bit [2]**

Core reset status bit.

This bit is UNKNOWN on reads if either EDPRSR.DLK is 1 or EDPRSR.PU is 0.

Otherwise permitted values are:

`0` The non-debug logic of the PE is not in reset state.

`1` The non-debug logic of the PE is in reset state.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

This field is accessible as shown below:

| Off | DLK | Default |
|---|---|---|
| UNK | UNK | RO |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**SPD, bit [1]**

Sticky core powerdown status bit.

This bit is UNKNOWN on reads if both EDPRSR.DLK and EDPRSR.PU are 1.

Otherwise, permitted values are:

`0` If the Core power domain is in a low-power or powerdown state (EDPRSR.PU is 0), it is not known whether the state of the debug registers in the Core power domain is lost.
If the Core power domain is in a powerup state (EDPRSR.PU is 1), the state of the debug registers in the Core power domain has not been lost.

`1` The state of the debug registers in the Core power domain is lost.

If DoubleLockStatus() == FALSE and the PE is not in the powerdown state, then following a read of EDPRSR, this bit clears to 0. Otherwise it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

If DoubleLockStatus() == TRUE and the PE is not in the powerdown state, it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

There are two logical states for the Core power domain when EDPRSR.PU is 0:

`Retention` The state of the debug registers, including EDPRSR.SPD, in the Core power domain is preserved, and restored on leaving retention state.

`Powerdown` The state of the debug registers in the Core power domain is lost, and a Cold reset is asserted on leaving powerdown state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In these states, EDPRSR.SPD is not cleared following a read of EDPRSR, and it is IMPLEMENTATION DEFINED whether:

- EDPRSR.SPD shows whether the state of the debug registers in the Core power domain has been lost since the last time EDPRSR was read when the Core power domain was powered up. That is:
  - While in the powerdown state, EDPRSR.SPD is RAO, as the state of the debug registers has been lost.
  - While in the retention state, the value of EDPRSR.EPD is unchanging and shows whether the state of the debug registers was lost before entering retention state. This does not allow EDPRSR.SPD to be fixed RAO in the retention state, as the state of the debug registers in the Core power domain is not lost by entering this state. EDPRSR.SPD can read as either 0 or 1.
- EDPRSR.SPD reads-as-zero, and:
  - On leaving the powerdown state, EDPRSR.SPD is set to 1, as the state of the debug registers has been lost.
  - On leaving the retention state, EDPRSR.SPD reverts to the value it had on entering retention state.

ARM recommends that an implementation make EDPRSR.SPD fixed RAO when in the powerdown state, particularly if it does not support a low-power retention state.

When this register has an architecturally-defined reset value, this field resets to 1.

This field resets to its defined reset value on Cold reset.

This field is accessible as shown below:

| Off | DLK | SLK | Default |
|---|---|---|---|
| RO | UNK | RO | RC |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.

**PU, bit [0]**

Core powerup status bit. Indicates whether the Core power domain debug registers can be accessed.

The value of EDPRSR.PU is IMPLEMENTATION DEFINED when the OS double-lock is locked. See the description of DLK for more information.

Otherwise, permitted values are:

`0` Core is in a low-power or powerdown state where the debug registers cannot be accessed.

`1` Core is in a powerup state where the debug registers can be accessed.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Warm reset.

This field is accessible as shown below:

| Off | Default |
|---|---|
| RAZ | RAO |

The meanings of the conditions in the table above are summarized in *Access permissions for the External debug interface registers* on page H8-4958. The priority at which each condition applies is from highest priority on the left to lowest on the right.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the EDPRSR:

EDPRSR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x314 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.41 EDRCR, External Debug Reserve Control Register

The EDRCR characteristics are:

**Purpose**

This register is used to allow imprecise entry to Debug state and clear sticky bits in EDSCR.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| Error | Error | Error | WI | WO |

**Configurations**

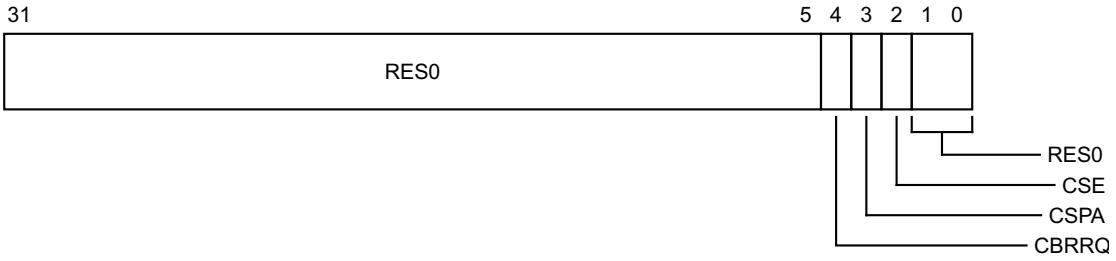
EDRCR is in the Core power domain.

**Attributes**

EDRCR is a 32-bit register.

## Field descriptions

The EDRCR bit assignments are:




**Bits [31:5]**

Reserved, RES0.

**CBRRQ, bit [4]**

Allow imprecise entry to Debug state. The actions on writing to this bit are:

0 No action.

1 Allow imprecise entry to Debug state, for example by canceling pending bus accesses.

Setting this bit to 1 allows a debugger to request imprecise entry to Debug state. An External Debug Request debug event must be pending before the debugger sets this bit to 1.

This feature is optional. If this feature is not implemented, writes to this bit are ignored.

**CSPA, bit [3]**

Clear Sticky Pipeline Advance. This bit is used to clear the EDSCR.PipeAdv bit to 0. The actions on writing to this bit are:

0 No action.

1 Clear the EDSCR.PipeAdv bit to 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**CSE, bit [2]**

Clear Sticky Error. Used to clear the EDSCR cumulative error bits to 0. The actions on writing to this bit are:

`0` No action.

`1` Clear the EDSCR.{TXU, RXO, ERR} bits, and, if the PE is in Debug state, the EDSCR.ITO bit, to 0.

**Bits [1:0]**

Reserved, RES0.

## Accessing the EDRCR:

EDRCR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | `0x090` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H9.2.42 EDSCR, External Debug Status and Control Register

The EDSCR characteristics are:

**Purpose**

Main control register for the debug implementation.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | SLK | Default |
|---|---|---|---|---|
| Error | Error | Error | RO | RW |

**Configurations**

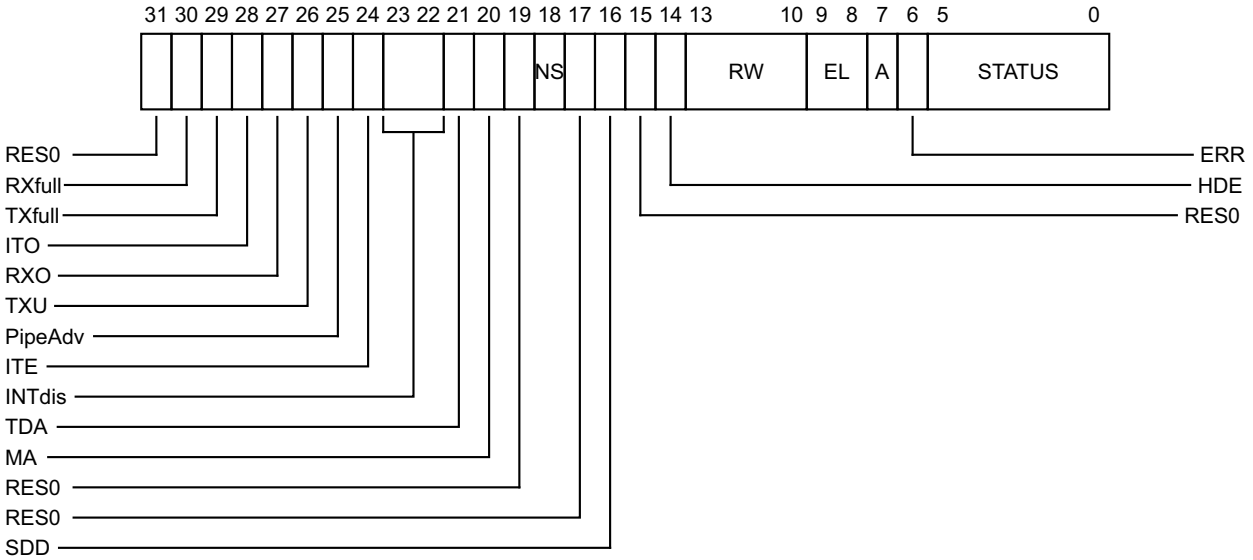
EDSCR is in the Core power domain.

**Attributes**

EDSCR is a 32-bit register.

### Field descriptions

The EDSCR bit assignments are:




**Bit [31]**

Reserved, RES0.

**RXfull, bit [30]**

DTRRX full. This bit is RO.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Cold reset.

**TXfull, bit [29]**

DTRTX full. This bit is RO.

When this register has an architecturally-defined reset value, this field resets to `0`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This field resets to its defined reset value on Cold reset.

**ITO, bit [28]**

EDITR overrun. This bit is RO.

If the PE is in Non-debug state, this bit is UNKNOWN. ITO is set to 0 on entry to Debug state.

**RXO, bit [27]**

DTRRX overrun. This bit is RO.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Cold reset.

**TXU, bit [26]**

DTRTX underrun. This bit is RO.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Cold reset.

**PipeAdv, bit [25]**

Pipeline advance. Read-only. Set to 1 every time the PE pipeline retires one or more instructions. Cleared to 0 by a write to EDRCR.CSPA.

The architecture does not define precisely when this bit is set to 1. It requires only that this happen periodically in Non-debug state to indicate that software execution is progressing.

**ITE, bit [24]**

ITR empty. This bit is RO.

If the PE is in Non-debug state, this bit is UNKNOWN. It is always valid in Debug state.

**INTdis, bits [23:22]**

Interrupt disable. Disables taking interrupts (including virtual interrupts and System Error interrupts) in Non-Debug state.

If external invasive debug is disabled, the value of this field is ignored.

If external invasive debug is enabled, the possible values of this field are:

`00` Do not disable interrupts

`01` Disable interrupts taken to Non-secure EL1.

`10` Disable interrupts taken only to Non-secure EL1 and Non-secure EL2. If external secure invasive debug is enabled, also disable interrupts taken to Secure EL1.

`11` Disable interrupts taken only to Non-secure EL1 and Non-secure EL2. If external secure invasive debug is enabled, also disable all other interrupts.

The value of INTdis does not affect whether an interrupt is a WFI wake-up event, but can mask an interrupt as a WFE wake-up event.

If EL3 and EL2 are not implemented, INTdis[0] is RO and reads the same value as INTdis[1], meaning only the values `00` and `11` can be selected.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on Cold reset.

**TDA, bit [21]**

Traps accesses to the following Debug System registers:

- AArch64: DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, DBGWVR<n>_EL1.
- AArch32: DBGBCR<n>, DBGBVR<n>, DBGBXVR<n>, DBGWCR<n>, DBGWVR<n>.

The possible values of this field are:

`0` Accesses to Debug System registers do not generate a Software Access debug event.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 Accesses to Debug System registers generate a Software Access debug event, if OSLSR.OSLK is 0 and if halting is allowed.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Cold reset.

**MA, bit [20]**

Memory access mode. Controls use of memory-access mode for accessing EDITR and the DCC. This bit is ignored if in Non-debug state and set to zero on entry to Debug state.

Possible values of this field are:

0 Normal access mode.

1 Memory access mode.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Cold reset.

**Bit [19]**

Reserved, RES0.

**NS, bit [18]**

Non-secure status. Read-only. When in Debug state, gives the current Security state:

0 Secure state, IsSecure() == TRUE.

1 Non-secure state, IsSecure() == FALSE.

In Non-debug state, this bit is UNKNOWN.

**Bit [17]**

Reserved, RES0.

**SDD, bit [16]**

Secure debug disabled. This bit is RO.

On entry to Debug state:

- If entering in Secure state, SDD is set to 0.
- If entering in Non-secure state, SDD is set to the inverse of ExternalSecureInvasiveDebugEnabled().

In Debug state, the value of the SDD bit does not change, even if ExternalSecureInvasiveDebugEnabled() changes.

In Non-debug state:

- SDD returns the inverse of ExternalSecureInvasiveDebugEnabled(). If the authentication signals that control ExternalSecureInvasiveDebugEnabled() change, a context synchronization operation is required to guarantee their effect.
- This bit is unaffected by the Security state of the PE.

If EL3 is not implemented and the implementation is Non-secure, this bit is RES1.

**Bit [15]**

Reserved, RES0.

**HDE, bit [14]**

Halting debug enable.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Cold reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**RW, bits [13:10]**

Exception level Execution state status. Read-only. In Debug state, each bit gives the current Execution state of each EL:

| RW | Meaning |
|---|---|
| 1111 | All Exception levels are using AArch64. |
| 1110 | EL0 is using AArch32. All other Exception levels are using AArch64. |
| 110x | EL0 and EL1 are using AArch32. All other Exception levels are using AArch64. Never seen if EL2 is not implemented in the current Security state. |
| 10xx | EL0, EL1, and, if implemented in the current Security state, EL2 are using AArch32. All other Exception levels are using AArch64. |
| 0xxx | All Exception levels are using AArch32. |

However:

- The value of 1110 is only permitted at EL0.
- The values 110x are not permitted if either:
  - EL2 is not implemented.
  - EL3 is implemented and SCR_EL3.NS/SCR.NS == 0.
- The values 10xx are not permitted if EL3 is not implemented.

In Non-debug state, this field is RAO.

**EL, bits [9:8]**

Exception level. Read-only. In Debug state, this gives the current EL of the PE.

In Non-debug state, this field is RAZ.

**A, bit [7]**

System Error interrupt pending. Read-only. In Debug state, indicates whether a SError interrupt is pending:

- If HCR_EL2.{AMO, TGE} = {1, 0} and in Non-secure EL0 or EL1, a virtual SError interrupt.
- Otherwise, a physical SError interrupt.

0 No SError interrupt pending.

1 SError interrupt pending.

A debugger can read EDSCR to check whether a SError interrupt is pending without having to execute further instructions. A pending SError might indicate data from target memory is corrupted.

UNKNOWN in Non-debug state.

**ERR, bit [6]**

Cumulative error flag. This field is RO. It is set to 1 following exceptions in Debug state and on any signaled overrun or underrun on the DTR or EDITR.

When this register has an architecturally-defined reset value, this field resets to 0.

This field resets to its defined reset value on Cold reset.

**STATUS, bits [5:0]**

Debug status flags. This field is RO.

The possible values of this field are:

000010 PE is in Non-debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| | |
|---|---|
| 000001 | PE is restarting (exiting Debug state). |
| 000111 | Breakpoint. |
| 010011 | External debug request. |
| 011011 | Halting step, normal. |
| 011111 | Halting step, exclusive. |
| 100011 | OS unlock catch. |
| 100111 | Reset catch. |
| 101011 | Watchpoint. |
| 101111 | HLT instruction. |
| 110011 | Software access to debug register. |
| 110111 | Exception catch. |
| 111011 | Halting step, no syndrome. |

All other values of STATUS are reserved.

### Accessing the EDSCR:

EDSCR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x088 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.43 EDVIDSR, External Debug Virtual Context Sample Register

The EDVIDSR characteristics are:

**Purpose**

Contains sampled values captured on reading EDPCSR.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | Default |
|---|---|---|---|
| Error | Error | Error | RO |

**Configurations**

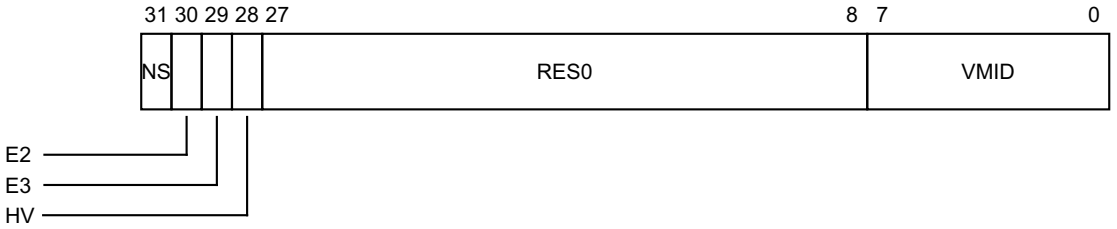
EDVIDSR is in the Core power domain.

**Attributes**

EDVIDSR is a 32-bit register.

## Field descriptions

The EDVIDSR bit assignments are:




**NS, bit [31]**

Non-secure state sample. Indicates the Security state associated with the most recent EDPCSR sample.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**E2, bit [30]**

Exception level 2 status sample. Indicates whether the most recent EDPCSR sample was associated with EL2. If EDVIDSR.NS == 0, this bit is 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**E3, bit [29]**

Exception level 3 status sample. Indicates whether the most recent EDPCSR sample was associated with AArch64 EL3. If EDVIDSR.NS == 1 or the PE was in AArch32 state when EDPCSR was read, this bit is 0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**HV, bit [28]**

EDPCSR high half valid. Indicates whether bits [63:32] of the most recent EDPCSR sample are valid. If EDVIDSR.HV == 0, the value of EDPCSR[63:32] is RAZ.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

**Bits [27:8]**

Reserved, RES0.

**VMID, bits [7:0]**

VMID sample. The value of VTTBR_EL2.VMID associated with the most recent EDPCSR sample. If EDVIDSR.NS == 0 or EDVIDSR.E2 == 1, this field is RAZ.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on Cold reset.

## Accessing the EDVIDSR:

EDVIDSR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x0A8 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.44 EDWAR, External Debug Watchpoint Address Register

The EDWAR characteristics are:

**Purpose**

Contains the virtual data address being accessed when a watchpoint debug event was triggered.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | Default |
|---|---|---|---|
| Error | Error | Error | RO |

**Configurations**

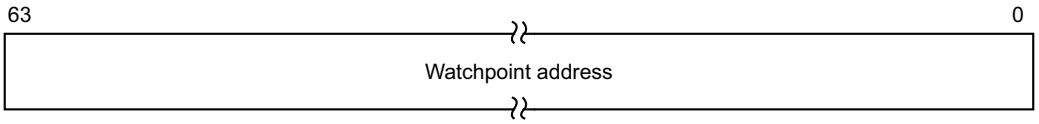
EDWAR is in the Core power domain.

**Attributes**

EDWAR is a 64-bit register.

#### Field descriptions

The EDWAR bit assignments are:




**Bits [63:0]**

Watchpoint address. The virtual data address being accessed when a watchpoint debug event was triggered and caused entry to Debug state.

UNKNOWN if the PE is in Non-debug state, or if Debug state was entered other than for a watchpoint debug event.

The address must be within a naturally-aligned block of memory of power-of-two size no larger than the DC ZVA block size.

#### Accessing the EDWAR:

EDWAR[31:0] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x030 |

EDWAR[63:32] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x034 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.2.45 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

**Purpose**

Provides identification information for the PE, including an implementer code for the device and a device ID number.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

External register MIDR_EL1 is architecturally mapped to AArch64 System register MIDR_EL1.

External register MIDR_EL1 is architecturally mapped to AArch32 System register MIDR.

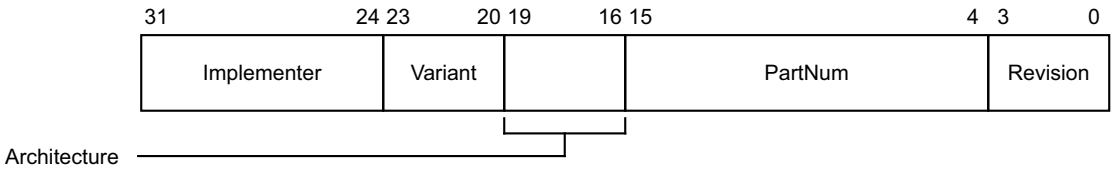
MIDR_EL1 is in the Debug power domain.

**Attributes**

MIDR_EL1 is a 32-bit register.

## Field descriptions

The MIDR_EL1 bit assignments are:




**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by ARM. Assigned codes include the following:

| Hex representation | ASCII representation | Implementer |
|---|---|---|
| 0x41 | A | ARM Limited |
| 0x42 | B | Broadcom Corporation |
| 0x43 | C | Cavium Inc. |
| 0x44 | D | Digital Equipment Corporation |
| 0x49 | I | Infineon Technologies AG |
| 0x4D | M | Motorola or Freescale Semiconductor Inc. |
| 0x4E | N | NVIDIA Corporation |
| 0x50 | P | Applied Micro Circuits Corporation |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

| Hex representation | ASCII representation | Implementer |
|---|---|---|
| 0x51 | Q | Qualcomm Inc. |
| 0x56 | V | Marvell International Ltd. |
| 0x69 | i | Intel Corporation |

ARM can assign codes that are not published in this manual. All values not assigned by ARM are reserved and must not be used.

**Variant, bits [23:20]**

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

**Architecture, bits [19:16]**

The permitted values of this field are:

0001 ARMv4

0010 ARMv4T

0011 ARMv5 (obsolete)

0100 ARMv5T

0101 ARMv5TE

0110 ARMv5TEJ

0111 ARMv6

1111 Architectural features are individually identified in the ID_* registers, see *Identification registers, functional group* on page G4-4162.

All other values are reserved.

**PartNum, bits [15:4]**

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by ARM, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

**Revision, bits [3:0]**

An IMPLEMENTATION DEFINED revision number for the device.

## Accessing the MIDR_EL1:

MIDR_EL1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0xD00 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H9.2.46 OSLAR_EL1, OS Lock Access Register

The OSLAR_EL1 characteristics are:

**Purpose**

Used to lock or unlock the OS lock.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | SLK | Default |
|---|---|---|---|
| Error | Error | WI | WO |

**Configurations**

External register OSLAR_EL1 is architecturally mapped to AArch64 System register OSLAR_EL1.

External register OSLAR_EL1 is architecturally mapped to AArch32 System register DBGOSLAR.

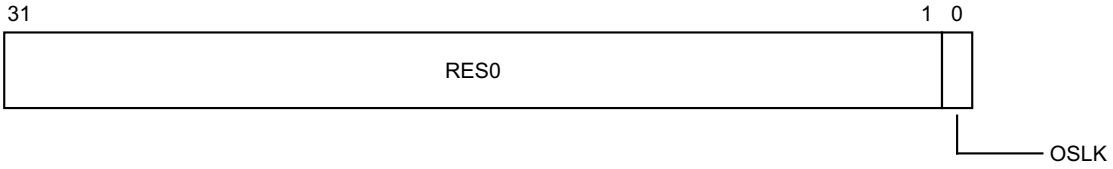
OSLAR_EL1 is in the Core power domain.

**Attributes**

OSLAR_EL1 is a 32-bit register.

### Field descriptions

The OSLAR_EL1 bit assignments are:




**Bits [31:1]**

Reserved, RES0.

**OSLK, bit [0]**

On writes to OSLAR_EL1, bit[0] is copied to the OS lock.

Use EDPRSR.OSLK to check the current status of the lock.

### Accessing the OSLAR_EL1:

OSLAR_EL1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| Debug | 0x300 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## H9.3 Cross-Trigger Interface registers

This section lists the Cross-Trigger Interface registers.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.3.1 ASICCTL, CTI External Multiplexer Control register

The ASICCTL characteristics are:

**Purpose**

Provides a control for external multiplexing of additional triggers into the CTI.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RW |

**Configurations**

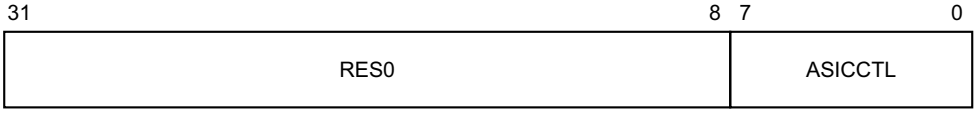
ASICCTL is in the Debug power domain.

**Attributes**

ASICCTL is a 32-bit register.

#### Field descriptions

The ASICCTL bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**ASICCTL, bits [7:0]**

IMPLEMENTATION DEFINED ASIC control. Provides a control for external multiplexing of additional triggers into the CTI.

If external multiplexing of trigger signals is implemented then the number of multiplexed signals on each trigger must be reflected in CTIDEVID.EXTMUXNUM.

If CTIDEVID.EXTMUXNUM is zero, this field is RAZ.

When this register has an architecturally-defined reset value, if this field is implemented as an RW field, it resets to an architecturally UNKNOWN value.

This field resets to its defined reset value on IMPLEMENTATION DEFINED reset.

#### Accessing the ASICCTL:

ASICCTL can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | 0x144 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.2 CTIAPPCLEAR, CTI Application Trigger Clear register

The CTIAPPCLEAR characteristics are:

**Purpose**

Clears bits of the Application Trigger register.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | WO |

**Configurations**

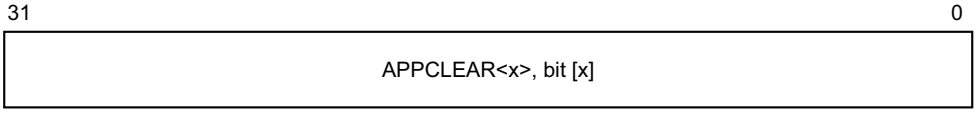
CTIAPPCLEAR is in the Debug power domain.

**Attributes**

CTIAPPCLEAR is a 32-bit register.

#### Field descriptions

The CTIAPPCLEAR bit assignments are:




**APPCLEAR<x>, bit [x], for x = 0 to 31**

Application trigger <x> disable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Writing to this bit has the following effect:

`0` No effect.

`1` Clear corresponding bit in CTIAPPTRIG to 0 and clear the corresponding channel event.

If the ECT does not support multicycle channel events, use of CTIAPPCLEAR is deprecated and the debugger must only use CTIAPPPULSE.

#### Accessing the CTIAPPCLEAR:

CTIAPPCLEAR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x018` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.3 CTIAPPPULSE, CTI Application Pulse register

The CTIAPPPULSE characteristics are:

**Purpose**

Causes event pulses to be generated on ECT channels.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | WO |

**Configurations**

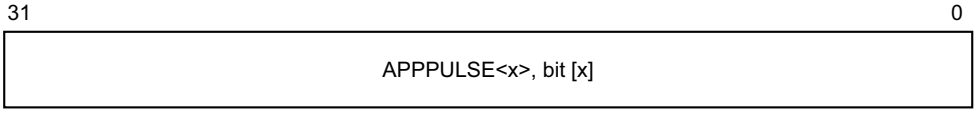
CTIAPPPULSE is in the Debug power domain.

**Attributes**

CTIAPPPULSE is a 32-bit register.

#### Field descriptions

The CTIAPPPULSE bit assignments are:




**APPPULSE<x>, bit [x], for x = 0 to 31**

Generate event pulse on ECT channel <x>.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Writing to this bit has the following effect:

`0` No effect.

`1` Channel <x> event pulse generated for one clock period.

#### Accessing the CTIAPPPULSE:

CTIAPPPULSE can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x01C` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.4 CTIAPPSET, CTI Application Trigger Set register

The CTIAPPSET characteristics are:

**Purpose**

Sets bits of the Application Trigger register.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RW |

**Configurations**

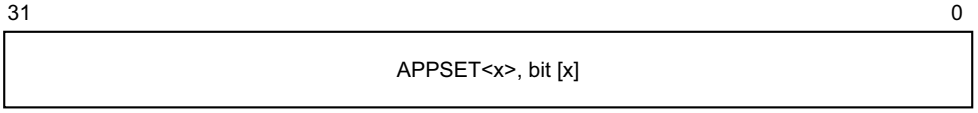
CTIAPPSET is in the Debug power domain.

**Attributes**

CTIAPPSET is a 32-bit register.

#### Field descriptions

The CTIAPPSET bit assignments are:




**APPSET<x>, bit [x], for x = 0 to 31**

Application trigger <x> enable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0` Reading this means the application trigger is inactive. Writing this has no effect.

`1` Reading this means the application trigger is active. Writing this sets the corresponding bit in CTIAPPTRIG to 1 and generates a channel event.

If the ECT does not support multicycle channel events, use of CTIAPPSET is deprecated and the debugger must only use CTIAPPPULSE.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on External debug reset.

#### Accessing the CTIAPPSET:

CTIAPPSET can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x014` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.5 CTIAUTHSTATUS, CTI Authentication Status register

The CTIAUTHSTATUS characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for CTI.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTIAUTHSTATUS is in the Debug power domain.

This register is OPTIONAL, and is required for CoreSight compliance. ARM recommends that this register is implemented.

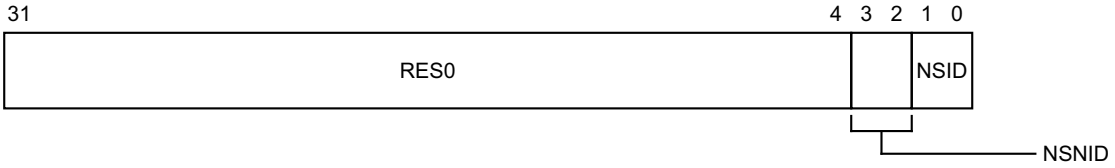
CTIAUTHSTATUS.{SNID,SID} are RAZ, because the CTI does not itself provide Secure authentication.

**Attributes**

CTIAUTHSTATUS is a 32-bit register.

## Field descriptions

The CTIAUTHSTATUS bit assignments are:




**Bits [31:4]**

Reserved, RES0.

**NSNID, bits [3:2]**

If EL3 is not implemented and the implemented Security state is Secure state, holds the same value as DBGAUTHSTATUS_EL1.SNID.

Otherwise, holds the same value as DBGAUTHSTATUS_EL1.NSNID.

**NSID, bits [1:0]**

If EL3 is not implemented and the implemented Security state is Secure state, holds the same value as DBGAUTHSTATUS_EL1.SID.

Otherwise, holds the same value as DBGAUTHSTATUS_EL1.NSID.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CTIAUTHSTATUS:

CTIAUTHSTATUS can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | 0xFB8 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.3.6 CTICHINSTATUS, CTI Channel In Status register

The CTICHINSTATUS characteristics are:

**Purpose**

Provides the raw status of the ECT channel inputs to the CTI.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

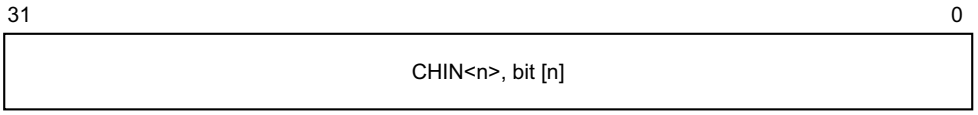
CTICHINSTATUS is in the Debug power domain.

**Attributes**

CTICHINSTATUS is a 32-bit register.

#### Field descriptions

The CTICHINSTATUS bit assignments are:




**CHIN<n>, bit [n], for n = 0 to 31**

Input channel <n> status.

Bits [31:N] are RAZ. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0` Input channel <n> is inactive.

`1` Input channel <n> is active.

#### Accessing the CTICHINSTATUS:

CTICHINSTATUS can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x138` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.7 CTICHOUTSTATUS, CTI Channel Out Status register

The CTICHOUTSTATUS characteristics are:

**Purpose**

Provides the status of the ECT channel outputs from the CTI.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

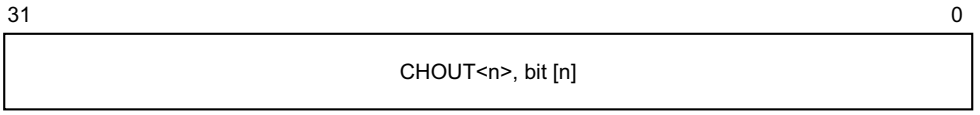
CTICHOUTSTATUS is in the Debug power domain.

**Attributes**

CTICHOUTSTATUS is a 32-bit register.

#### Field descriptions

The CTICHOUTSTATUS bit assignments are:




**CHOUT<n>, bit [n], for n = 0 to 31**

Output channel <n> status.

Bits [31:N] are RAZ. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0` Output channel <n> is inactive.

`1` Output channel <n> is active.

#### Accessing the CTICHOUTSTATUS:

CTICHOUTSTATUS can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x13C` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.8 CTICIDR0, CTI Component Identification Register 0

The CTICIDR0 characteristics are:

**Purpose**

Provides information to identify a CTI component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTICIDR0 is in the Debug power domain.

CTICIDR0 is optional to implement in the external register interface.

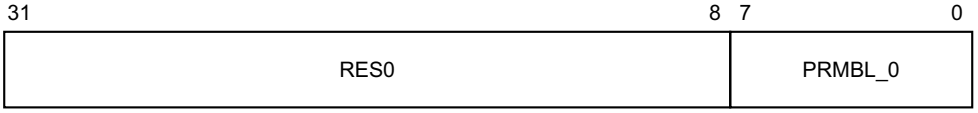
This register is required for CoreSight compliance.

**Attributes**

CTICIDR0 is a 32-bit register.

#### Field descriptions

The CTICIDR0 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PRMBL_0, bits [7:0]**

Preamble. Must read as `0x0D`.

#### Accessing the CTICIDR0:

CTICIDR0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFF0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.9 CTICIDR1, CTI Component Identification Register 1

The CTICIDR1 characteristics are:

**Purpose**

Provides information to identify a CTI component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTICIDR1 is in the Debug power domain.

CTICIDR1 is optional to implement in the external register interface.

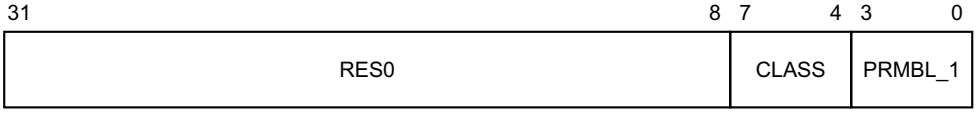
This register is required for CoreSight compliance.

**Attributes**

CTICIDR1 is a 32-bit register.

#### Field descriptions

The CTICIDR1 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**CLASS, bits [7:4]**

Component class. Reads as `0x9`, debug component.

**PRMBL_1, bits [3:0]**

Preamble. RAZ.

#### Accessing the CTICIDR1:

CTICIDR1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFF4` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.10 CTICIDR2, CTI Component Identification Register 2

The CTICIDR2 characteristics are:

**Purpose**

Provides information to identify a CTI component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTICIDR2 is in the Debug power domain.

CTICIDR2 is optional to implement in the external register interface.

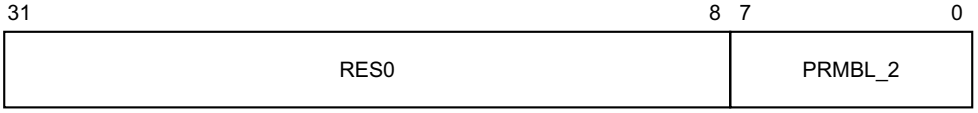
This register is required for CoreSight compliance.

**Attributes**

CTICIDR2 is a 32-bit register.

#### Field descriptions

The CTICIDR2 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PRMBL_2, bits [7:0]**

Preamble. Must read as `0x05`.

#### Accessing the CTICIDR2:

CTICIDR2 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFF8` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.11 CTICIDR3, CTI Component Identification Register 3

The CTICIDR3 characteristics are:

**Purpose**

Provides information to identify a CTI component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTICIDR3 is in the Debug power domain.

CTICIDR3 is optional to implement in the external register interface.

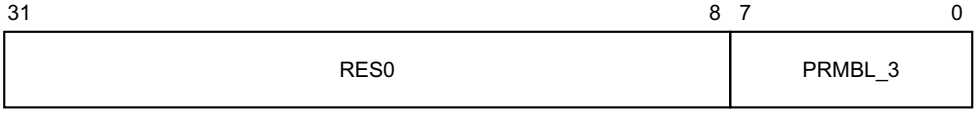
This register is required for CoreSight compliance.

**Attributes**

CTICIDR3 is a 32-bit register.

#### Field descriptions

The CTICIDR3 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PRMBL_3, bits [7:0]**

Preamble. Must read as `0xB1`.

#### Accessing the CTICIDR3:

CTICIDR3 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFFC` |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.3.12 CTICLAIMCLR, CTI CLAIM Tag Clear register

The CTICLAIMCLR characteristics are:

**Purpose**

Used by software to read the values of the CLAIM bits, and to clear these bits to 0.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RW |

**Configurations**

CTICLAIMCLR is in the Debug power domain.

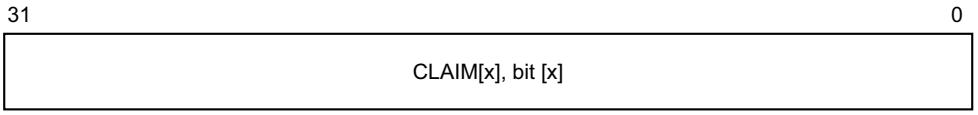
CTICLAIMCLR is optional to implement in the external register interface.

**Attributes**

CTICLAIMCLR is a 32-bit register.

#### Field descriptions

The CTICLAIMCLR bit assignments are:




**CLAIM[x], bit [x], for x = 0 to 31**

Clear CLAIM tag. If x is greater than or equal to the IMPLEMENTATION DEFINED number of CLAIM tags, this bit is RAZ/SBZ. Software can rely on these bits reading as zero, and must use a Should-Be-Zero policy on writes. Implementations must ignore writes.

Otherwise, reads return the value of CLAIM[x] and the behavior on writes is:

`0` No action.

`1` Indirectly clear CLAIM[x] to 0.

A single write to CTICLAIMCLR can clear multiple tags to 0.

#### Accessing the CTICLAIMCLR:

CTICLAIMCLR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFA4` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.13 CTICLAIMSET, CTI CLAIM Tag Set register

The CTICLAIMSET characteristics are:

**Purpose**

Used by software to set CLAIM bits to 1.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RW |

**Configurations**

CTICLAIMSET is in the Debug power domain.

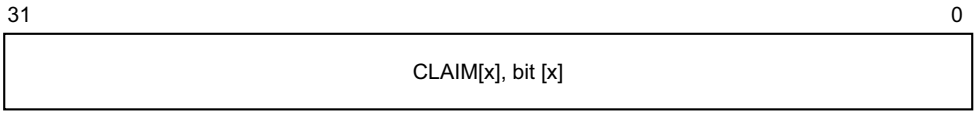
CTICLAIMSET is optional to implement in the external register interface.

**Attributes**

CTICLAIMSET is a 32-bit register.

#### Field descriptions

The CTICLAIMSET bit assignments are:




**CLAIM[x], bit [x], for x = 0 to 31**

CLAIM tag set bit. If x is greater than or equal to the IMPLEMENTATION DEFINED number of CLAIM tags, this bit is RAZ/SBZ. Software can rely on these bits reading as zero, and must use a Should-Be-Zero policy on writes. Implementations must ignore writes.

Otherwise, the bit is RAO and the behavior on writes is:

`0` No action.

`1` Indirectly set CLAIM[x] tag to 1.

A single write to CTICLAIMSET can set multiple tags to 1.

#### Accessing the CTICLAIMSET:

CTICLAIMSET can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFA0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.14 CTICONTROL, CTI Control register

The CTICONTROL characteristics are:

**Purpose**

Controls whether the CTI is enabled.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RW |

**Configurations**

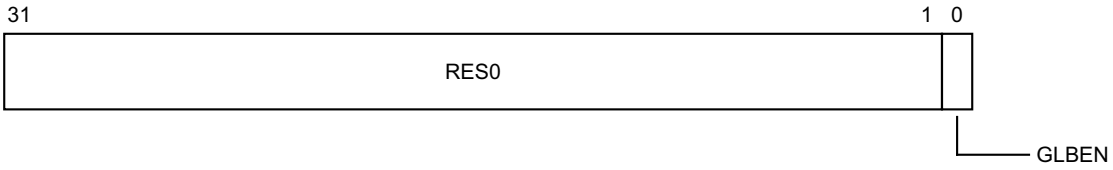
CTICONTROL is in the Debug power domain.

**Attributes**

CTICONTROL is a 32-bit register.

#### Field descriptions

The CTICONTROL bit assignments are:




**Bits [31:1]**

Reserved, RES0.

**GLBEN, bit [0]**

Enables or disables the CTI mapping functions. Possible values of this field are:

`0` CTI mapping functions disabled.

`1` CTI mapping functions enabled.

When the mapping functions are disabled, no new events are signaled on either output triggers or output channels. If a previously asserted output trigger has not been acknowledged, it remains asserted after the mapping functions are disabled. All output triggers are disabled by CTI reset.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on External debug reset.

#### Accessing the CTICONTROL:

CTICONTROL can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x000` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.15 CTIDEVAFF0, CTI Device Affinity register 0

The CTIDEVAFF0 characteristics are:

**Purpose**

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTIDEVAFF0 is in the Debug power domain.

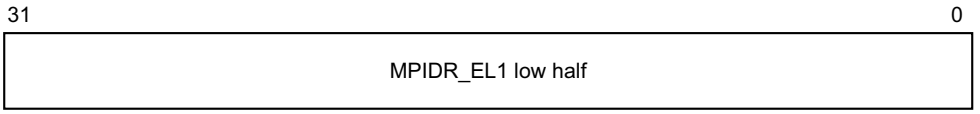
CTIDEVAFF0 is optional to implement in the external register interface.

**Attributes**

CTIDEVAFF0 is a 32-bit register.

#### Field descriptions

The CTIDEVAFF0 bit assignments are:




**Bits [31:0]**

MPIDR_EL1 low half. Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.

#### Accessing the CTIDEVAFF0:

CTIDEVAFF0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | 0xFA8 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.16 CTIDEVAFF1, CTI Device Affinity register 1

The CTIDEVAFF1 characteristics are:

**Purpose**

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTIDEVAFF1 is in the Debug power domain.

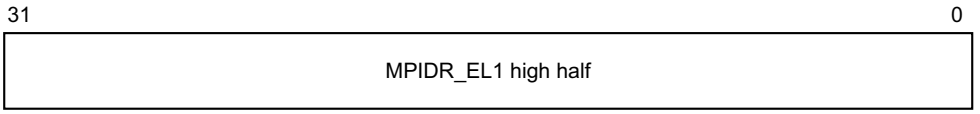
CTIDEVAFF1 is optional to implement in the external register interface.

**Attributes**

CTIDEVAFF1 is a 32-bit register.

#### Field descriptions

The CTIDEVAFF1 bit assignments are:




**Bits [31:0]**

MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.

#### Accessing the CTIDEVAFF1:

CTIDEVAFF1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | 0xFAC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.17 CTIDEVARCH, CTI Device Architecture register

The CTIDEVARCH characteristics are:

**Purpose**

Identifies the programmers' model architecture of the CTI component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTIDEVARCH is in the Debug power domain.

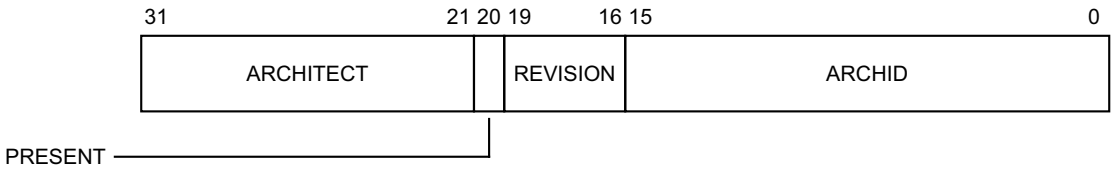
CTIDEVARCH is optional to implement in the external register interface.

**Attributes**

CTIDEVARCH is a 32-bit register.

#### Field descriptions

The CTIDEVARCH bit assignments are:




**ARCHITECT, bits [31:21]**

Defines the architecture of the component. For CTI, this is ARM Limited.

Bits [31:28] are the JEP 106 continuation code, `0x4`.

Bits [27:21] are the JEP 106 ID code, `0x3B`.

**PRESENT, bit [20]**

When set to 1, indicates that the DEVARCH is present.

This field is 1 in ARMv8.

**REVISION, bits [19:16]**

Defines the architecture revision. For architectures defined by ARM this is the minor revision.

For CTI, the revision defined by ARMv8 is `0x0`.

All other values are reserved.

**ARCHID, bits [15:0]**

Defines this part to be an ARMv8 debug component. For architectures defined by ARM this is further subdivided.

For CTI:

- Bits [15:12] are the architecture version, `0x1`.
- Bits [11:0] are the architecture part number, `0xA14`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This corresponds to CTI architecture version CTIv2.

### Accessing the CTIDEVARCH:

CTIDEVARCH can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | 0xFBC |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.3.18 CTIDEVID, CTI Device ID register 0

The CTIDEVID characteristics are:

**Purpose**

Describes the CTI component to the debugger.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

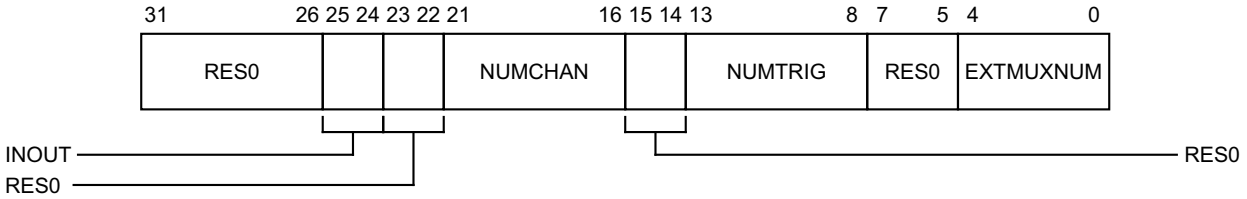
CTIDEVID is in the Debug power domain.

**Attributes**

CTIDEVID is a 32-bit register.

## Field descriptions

The CTIDEVID bit assignments are:




**Bits [31:26]**

Reserved, RES0.

**INOUT, bits [25:24]**

Input/output options. Indicates presence of the input gate. If the CTM is not implemented, this field is RAZ.

`00` CTIGATE does not mask propagation of input events from external channels.

`01` CTIGATE masks propagation of input events from external channels.

All other values are reserved.

**Bits [23:22]**

Reserved, RES0.

**NUMCHAN, bits [21:16]**

Number of ECT channels implemented. IMPLEMENTATION DEFINED. For ARMv8, valid values are:

`000011` 3 channels (0..2) implemented.

`000100` 4 channels (0..3) implemented.

`000101` 5 channels (0..4) implemented.

`000110` 6 channels (0..5) implemented.

and so on up to `100000`, 32 channels (0..31) implemented.

All other values are reserved.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Bits [15:14]**

Reserved, RES0.

**NUMTRIG, bits [13:8]**

Number of triggers implemented. IMPLEMENTATION DEFINED. This is one more than the index of the largest trigger, rather than the actual number of triggers.

For ARMv8, valid values are:

`000011` Up to 3 triggers (0..2) implemented.

`001000` Up to 8 triggers (0..7) implemented.

`001001` Up to 9 triggers (0..8) implemented.

`001010` Up to 10 triggers (0..9) implemented.

and so on up to `100000`, 32 triggers (0..31) implemented.

All other values are reserved. If the implementation includes an ARM trace macrocell, this field must be at least `001000`. There is no guarantee that any of the implemented triggers, including the highest numbered, are connected to any components.

**Bits [7:5]**

Reserved, RES0.

**EXTMUXNUM, bits [4:0]**

Maximum number of external triggers available for multiplexing into the CTI. This relates only to additional external triggers outside those defined for ARMv8.

### Accessing the CTIDEVID:

CTIDEVID can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFC8` |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.3.19 CTIDEVID1, CTI Device ID register 1

The CTIDEVID1 characteristics are:

**Purpose**

Reserved for future information about the CTI component to the debugger.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

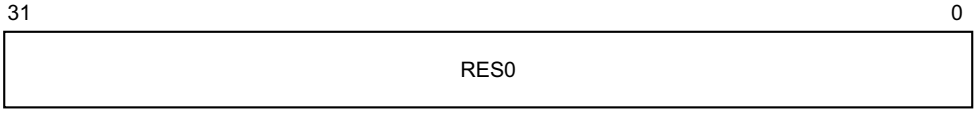
CTIDEVID1 is in the Debug power domain.

**Attributes**

CTIDEVID1 is a 32-bit register.

#### Field descriptions

The CTIDEVID1 bit assignments are:




**Bits [31:0]**

Reserved, RES0.

#### Accessing the CTIDEVID1:

CTIDEVID1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | 0xFC4 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.3.20 CTIDEVID2, CTI Device ID register 2

The CTIDEVID2 characteristics are:

**Purpose**

Reserved for future information about the CTI component to the debugger.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

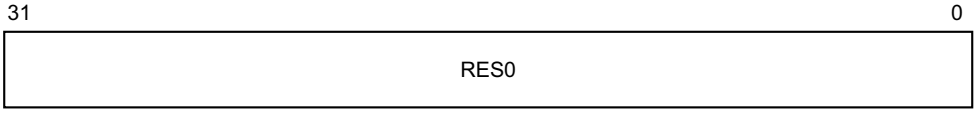
CTIDEVID2 is in the Debug power domain.

**Attributes**

CTIDEVID2 is a 32-bit register.

#### Field descriptions

The CTIDEVID2 bit assignments are:




**Bits [31:0]**

Reserved, RES0.

#### Accessing the CTIDEVID2:

CTIDEVID2 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | 0xFC0 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.3.21 CTIDEVTYPE, CTI Device Type register

The CTIDEVTYPE characteristics are:

**Purpose**

Indicates to a debugger that this component is part of a PEs cross-trigger interface.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTIDEVTYPE is in the Debug power domain.

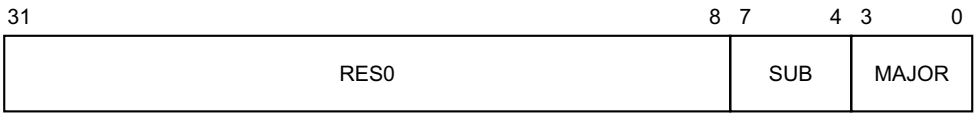
CTIDEVTYPE is optional to implement in the external register interface.

**Attributes**

CTIDEVTYPE is a 32-bit register.

#### Field descriptions

The CTIDEVTYPE bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**SUB, bits [7:4]**

Subtype. Must read as `0x1` to indicate this is a component within a PE.

**MAJOR, bits [3:0]**

Major type. Must read as `0x4` to indicate this is a cross-trigger component.

#### Accessing the CTIDEVTYPE:

CTIDEVTYPE can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFCC` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.22 CTIGATE, CTI Channel Gate Enable register

The CTIGATE characteristics are:

**Purpose**

Determines whether events on channels propagate through the CTM to other ECT components, or from the CTM into the CTI.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RW |

**Configurations**

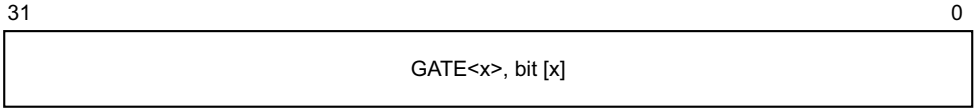
CTIGATE is in the Debug power domain.

**Attributes**

CTIGATE is a 32-bit register.

#### Field descriptions

The CTIGATE bit assignments are:




**GATE<x>, bit [x], for x = 0 to 31**

Channel <x> gate enable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0` Disable output and, if CTIDEVID.INOUT == `0b01`, input channel <x> propagation.

`1` Enable output and, if CTIDEVID.INOUT == `0b01`, input channel <x> propagation.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on External debug reset.

#### Accessing the CTIGATE:

CTIGATE can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x140` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.23 CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31

The CTIINEN<n> characteristics are:

**Purpose**

Enables the signaling of an event on output channels when input trigger event n is received by the CTI.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RW |

**Configurations**

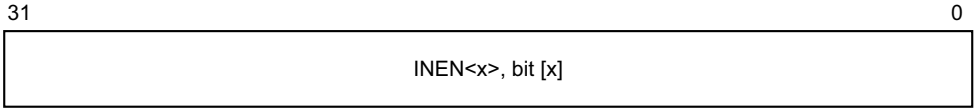
CTIINEN<n> is in the Debug power domain.

**Attributes**

CTIINEN<n> is a 32-bit register.

#### Field descriptions

The CTIINEN<n> bit assignments are:




**INEN<x>, bit [x], for x = 0 to 31**

Input trigger <n> to output channel <x> enable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0` Input trigger <n> will not generate an event on output channel <x>.

`1` Input trigger <n> will generate an event on output channel <x>.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on External debug reset.

#### Accessing the CTIINEN<n>:

CTIINEN<n> can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x020` + 4n |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.24 CTIINTACK, CTI Output Trigger Acknowledge register

The CTIINTACK characteristics are:

**Purpose**

Can be used to create soft acknowledges for output triggers.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | WO |

A debugger must read CTITRIGOUTSTATUS to confirm that the output trigger has been acknowledged before generating any event that must be ordered after the write to CTIINTACK.

**Configurations**

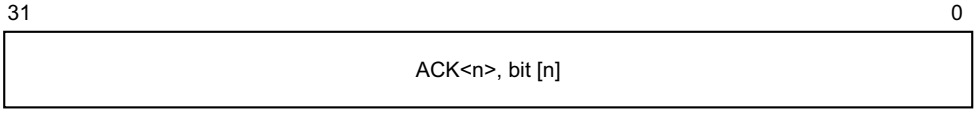
CTIINTACK is in the Debug power domain.

**Attributes**

CTIINTACK is a 32-bit register.

## Field descriptions

The CTIINTACK bit assignments are:




**ACK<n>, bit [n], for n = 0 to 31**

Acknowledge for output trigger <n>.

Bits [31:N] are RAZ/WI. N is the number of CTI triggers implemented as defined by the CTIDEVID.NUMTRIG field.

If any of the following is true, writes to ACK<n> are ignored:

- n >= CTIDEVID.NUMTRIG, the number of implemented triggers.
- Output trigger n is not active.
- The channel mapping function output, as controlled by CTIOUTEN<n>, is still active.

Otherwise, if any of the following are true, it is IMPLEMENTATION DEFINED whether writes to ACK<n> are ignored:

- Output trigger n is not implemented.
- Output trigger n is not connected.
- Output trigger n is self-acknowledging and does not require software acknowledge.

Otherwise, the behavior on writes to ACK<n> is as follows:

`0` No effect

`1` Deactivate the trigger.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CTIINTACK:

CTIINTACK can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x010` |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.3.25 CTIITCTRL, CTI Integration mode Control register

The CTIITCTRL characteristics are:

**Purpose**

Enables the CTI to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RW |

**Configurations**

CTIITCTRL is in the Debug power domain.

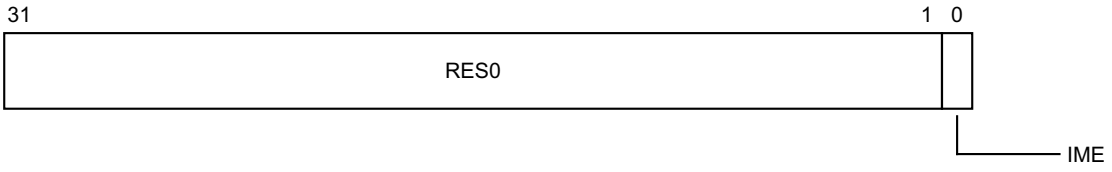
CTIITCTRL is optional to implement in the external register interface.

**Attributes**

CTIITCTRL is a 32-bit register.

#### Field descriptions

The CTIITCTRL bit assignments are:




**Bits [31:1]**

Reserved, RES0.

**IME, bit [0]**

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection. The integration mode behavior is IMPLEMENTATION DEFINED.

`0` Normal operation.

`1` Integration mode enabled.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on IMPLEMENTATION DEFINED reset.

#### Accessing the CTIITCTRL:

CTIITCTRL can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xF00` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.26 CTILAR, CTI Lock Access Register

The CTILAR characteristics are:

**Purpose**

Allows or disallows access to the CTI registers through a memory-mapped interface.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| WO |

**Configurations**

CTILAR is in the Debug power domain.

If OPTIONAL memory-mapped access to the external debug interface is supported then an OPTIONAL Software lock can be implemented as part of CoreSight compliance.

CTILAR ignores writes if the Software lock is not implemented and ignores writes for other accesses to the external debug interface.

The Software lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

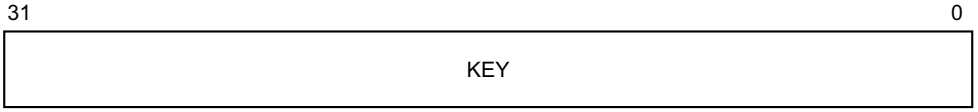
Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

**Attributes**

CTILAR is a 32-bit register.

#### Field descriptions

The CTILAR bit assignments are:




**KEY, bits [31:0]**

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

#### Accessing the CTILAR:

CTILAR can be accessed through the memory-mapped interface:

| Component | Offset |
|---|---|
| CTI | `0xFB0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.27 CTILSR, CTI Lock Status Register

The CTILSR characteristics are:

**Purpose**

Indicates the current status of the software lock for CTI registers.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

CTILSR is in the Debug power domain.

If OPTIONAL memory-mapped access to the external debug interface is supported then an OPTIONAL Software lock can be implemented as part of CoreSight compliance.

CTILSR is RAZ if the Software lock is not implemented and is RAZ for other accesses to the external debug interface.

The Software lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

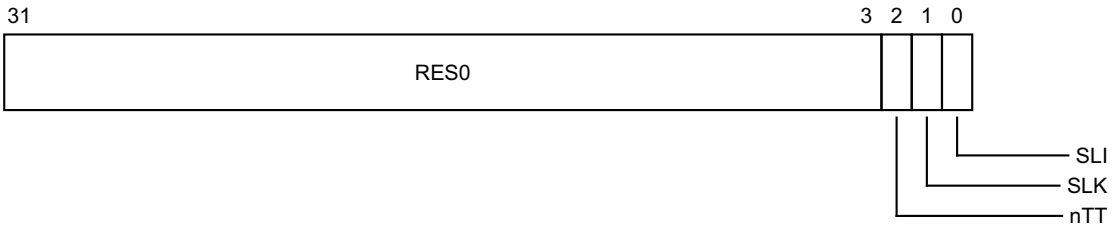
Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

**Attributes**

CTILSR is a 32-bit register.

#### Field descriptions

The CTILSR bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**nTT, bit [2]**

Not thirty-two bit access required. RAZ.

**SLK, bit [1]**

Software lock status for this component. For an access to LSR that is not a memory-mapped access, or when the software lock is not implemented, this field is RES0.

For memory-mapped accesses when the software lock is implemented, possible values of this field are:

`0` Lock clear. Writes are permitted to this component's registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 Lock set. Writes to this component's registers are ignored, and reads have no side effects.

When this register has an architecturally-defined reset value, this field resets to 1.

This field resets to its defined reset value on External debug reset.

**SLI, bit [0]**

Software lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ. For memory-mapped accesses, the value of this field is IMPLEMENTATION DEFINED. Permitted values are:

0 Software lock not implemented or not memory-mapped access.

1 Software lock implemented and memory-mapped access.

### Accessing the CTILSR:

CTILSR can be accessed through the memory-mapped interface:

| Component | Offset |
|---|---|
| CTI | 0xFB4 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.28 CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31

The CTIOUTEN<n> characteristics are:

**Purpose**

Defines which input channels generate output trigger n.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RW |

**Configurations**

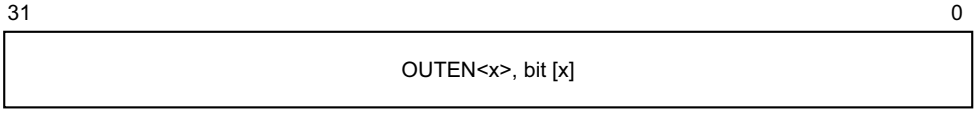
CTIOUTEN<n> is in the Debug power domain.

**Attributes**

CTIOUTEN<n> is a 32-bit register.

#### Field descriptions

The CTIOUTEN<n> bit assignments are:




**OUTEN<x>, bit [x], for x = 0 to 31**

Input channel <x> to output trigger <n> enable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

`0` An event on input channel <x> will not cause output trigger <n> to be asserted.

`1` An event on input channel <x> will cause output trigger <n> to be asserted.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

This field resets to its defined reset value on External debug reset.

#### Accessing the CTIOUTEN<n>:

CTIOUTEN<n> can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x0A0` + 4n |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.3.29 CTIPIDR0, CTI Peripheral Identification Register 0

The CTIPIDR0 characteristics are:

**Purpose**

Provides information to identify a CTI component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTIPIDR0 is in the Debug power domain.

CTIPIDR0 is optional to implement in the external register interface.

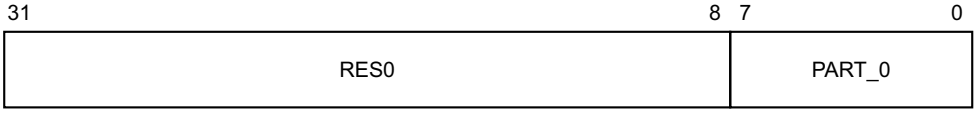
This register is required for CoreSight compliance.

**Attributes**

CTIPIDR0 is a 32-bit register.

#### Field descriptions

The CTIPIDR0 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PART_0, bits [7:0]**

Part number, least significant byte.

#### Accessing the CTIPIDR0:

CTIPIDR0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | 0xFE0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.30 CTIPIDR1, CTI Peripheral Identification Register 1

The CTIPIDR1 characteristics are:

**Purpose**

Provides information to identify a CTI component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTIPIDR1 is in the Debug power domain.

CTIPIDR1 is optional to implement in the external register interface.

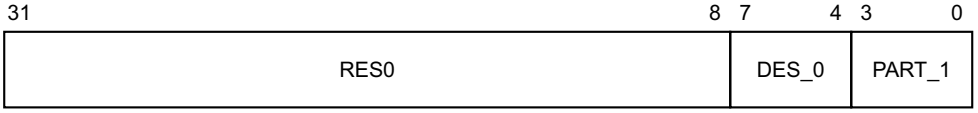
This register is required for CoreSight compliance.

**Attributes**

CTIPIDR1 is a 32-bit register.

#### Field descriptions

The CTIPIDR1 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**DES_0, bits [7:4]**

Designer, least significant nibble of JEP106 ID code. For ARM Limited, this field is `0b1011`.

**PART_1, bits [3:0]**

Part number, most significant nibble.

#### Accessing the CTIPIDR1:

CTIPIDR1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFE4` |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### H9.3.31 CTIPIDR2, CTI Peripheral Identification Register 2

The CTIPIDR2 characteristics are:

**Purpose**

Provides information to identify a CTI component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTIPIDR2 is in the Debug power domain.

CTIPIDR2 is optional to implement in the external register interface.

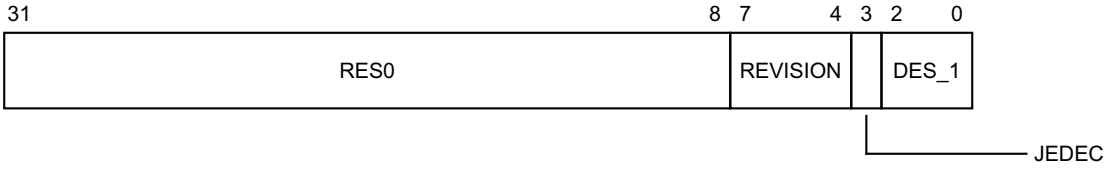
This register is required for CoreSight compliance.

**Attributes**

CTIPIDR2 is a 32-bit register.

#### Field descriptions

The CTIPIDR2 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**REVISION, bits [7:4]**

Part major revision. Parts can also use this field to extend Part number to 16-bits.

**JEDEC, bit [3]**

RAO. Indicates a JEP106 identity code is used.

**DES_1, bits [2:0]**

Designer, most significant bits of JEP106 ID code. For ARM Limited, this field is `0b011`.

#### Accessing the CTIPIDR2:

CTIPIDR2 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFE8` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.32 CTIPIDR3, CTI Peripheral Identification Register 3

The CTIPIDR3 characteristics are:

**Purpose**

Provides information to identify a CTI component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTIPIDR3 is in the Debug power domain.

CTIPIDR3 is optional to implement in the external register interface.

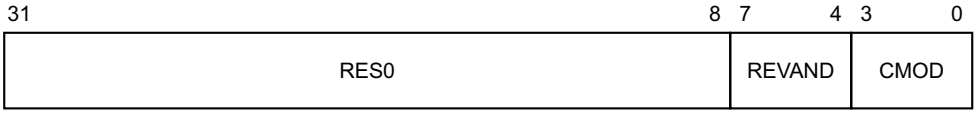
This register is required for CoreSight compliance.

**Attributes**

CTIPIDR3 is a 32-bit register.

#### Field descriptions

The CTIPIDR3 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**REVAND, bits [7:4]**

Part minor revision. Parts using CTIPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

**CMOD, bits [3:0]**

Customer modified. Indicates someone other than the Designer has modified the component.

#### Accessing the CTIPIDR3:

CTIPIDR3 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | 0xFEC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.33 CTIPIDR4, CTI Peripheral Identification Register 4

The CTIPIDR4 characteristics are:

**Purpose**

Provides information to identify a CTI component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

CTIPIDR4 is in the Debug power domain.

CTIPIDR4 is optional to implement in the external register interface.

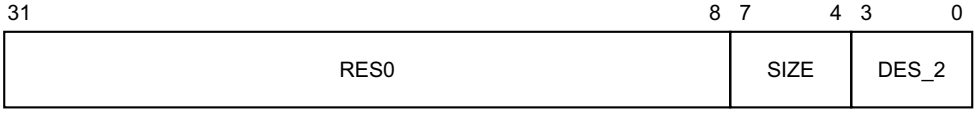
This register is required for CoreSight compliance.

**Attributes**

CTIPIDR4 is a 32-bit register.

#### Field descriptions

The CTIPIDR4 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**SIZE, bits [7:4]**

Size of the component. RAZ. $Log_2$ of the number of 4KB pages from the start of the component to the end of the component ID registers.

**DES_2, bits [3:0]**

Designer, JEP106 continuation code, least significant nibble. For ARM Limited, this field is `0b0100`.

#### Accessing the CTIPIDR4:

CTIPIDR4 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0xFD0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.34 CTITRIGINSTATUS, CTI Trigger In Status register

The CTITRIGINSTATUS characteristics are:

**Purpose**

Provides the status of the trigger inputs.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

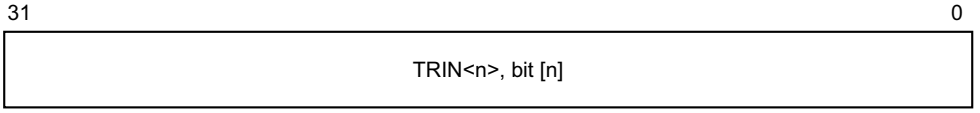
CTITRIGINSTATUS is in the Debug power domain.

**Attributes**

CTITRIGINSTATUS is a 32-bit register.

#### Field descriptions

The CTITRIGINSTATUS bit assignments are:




**TRIN<n>, bit [n], for n = 0 to 31**

Trigger input <n> status.

Bits [31:N] are RAZ. N is the number of CTI triggers implemented as defined by the CTIDEVID.NUMTRIG field.

Possible values of this bit are:

`0` Input trigger n is inactive.

`1` Input trigger n is active.

Not implemented and not-connected input triggers are always inactive.

#### Accessing the CTITRIGINSTATUS:

CTITRIGINSTATUS can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x130` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### H9.3.35 CTITRIGOUTSTATUS, CTI Trigger Out Status register

The CTITRIGOUTSTATUS characteristics are:

**Purpose**

Provides the raw status of the trigger outputs, after processing by any IMPLEMENTATION DEFINED trigger interface logic. For output triggers that are self-acknowledging, this is only meaningful if the CTI implements multicycle channel events.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

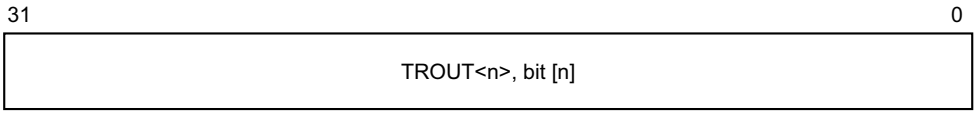
CTITRIGOUTSTATUS is in the Debug power domain.

**Attributes**

CTITRIGOUTSTATUS is a 32-bit register.

#### Field descriptions

The CTITRIGOUTSTATUS bit assignments are:




**TROUT<n>, bit [n], for n = 0 to 31**

Trigger output <n> status.

Bits [31:N] are RAZ, where N is the value of the CTIDEVID.NUMTRIG field.

If n < N, and output trigger <n> is implemented and connected, and either the trigger is not self-acknowledging or the CTI implements multicycle channel events, then permitted values for TROUT<n> are:

`0` Output trigger n is inactive.

`1` Output trigger n is active.

Otherwise when n < N it is IMPLEMENTATION DEFINED whether TROUT<n> behaves as described here or is RAZ.

#### Accessing the CTITRIGOUTSTATUS:

CTITRIGOUTSTATUS can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| CTI | `0x134` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Part I

## Memory-mapped Components of the ARMv8 Architecture

# Chapter I1
# System Level Implementation of the Generic Timer

This chapter defines the system level implementation of the Generic Timer. It contains the following sections:


- *About the Generic Timer specification* on page I1-5098.
- *Memory-mapped counter module* on page I1-5099.
- *Counter module control and status register summary* on page I1-5102.
- *Memory-mapped timer components* on page I1-5104.
- *The CNTBaseN and CNTEL0BaseN frames* on page I1-5105.
- *The CNTCTLBase frame* on page I1-5107.
- *Providing a complete set of counter and timer features* on page I1-5108.
- *Gray-count scheme for timer distribution scheme* on page I1-5110.


**Note**

- *Generic Timer memory-mapped register descriptions* on page I3-5173 describes the System level Generic Timer registers. These registers are memory-mapped.
- Chapter D6 *The Generic Timer in AArch64 state* gives a general description of the AArch64 state view of the Generic Timer, and describes the AArch64 System register interface to the Generic Timer.
- Chapter G5 *The Generic Timer in AArch32 state* gives a general description of the AArch32 state view of the Generic Timer, and describes the AArch32 System register interface to the Generic Timer.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## I1.1 About the Generic Timer specification

Chapter D6 *The Generic Timer in AArch64 state* describes the ARM Generic Timer and its implementation as seen from AArch64 state. Chapter G5 *The Generic Timer in AArch32 state* describes the ARM Generic Timer and its implementation as seen from AArch32 state. These chapters include the definition of the low-latency System register interface to the Generic Timer. However, the ARM Generic Timer architecture requires the inclusion of a memory-mapped component, the memory-mapped counter module, to control the generation of the Count value used by the Generic Timers. This is described in *Memory-mapped counter module* on page I1-5099.

In addition, the Generic Timer architecture defines the architecture of an optional memory-mapped timer component. This gives a standardized way of providing timers for programmable system components other than PEs that implement the ARM architecture. *Memory-mapped timer components* on page I1-5104 describes this memory-mapped timer.

### I1.1.1 Registers in the system level implementation of the Generic Timer

Registers that control components of the system level implementation of the Generic Timer are grouped into *frames*. This specification defines the registers in each frame, and their offsets within the frame. The system defines the position of each frame in the memory map. This means the base addresses for each frame is IMPLEMENTATION DEFINED.

**Note**

The final twelve words of the first or only 4KB block of a register memory frame is an ID block.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I1.2 Memory-mapped counter module

The memory-mapped counter module provides top-level control of the system counter. The CNTControlBase frame holds the registers for the memory-mapped counter, and provides:

- A RW control register CNTCR, that provides:
  - An enable bit for the system counter.
  - An enable bit for Halt-on-Debug. When this is enabled, if the debug halt signal into the system counter is asserted, it halts the system counter. Otherwise, the system counter ignores the state of this halt signal. For more information about Halt-on-Debug, contact ARM.
  - A field that can be written to request a change to the update frequency of the system counter, with a corresponding change to the increment made at each update. For more information see *Control of counter operating frequency and increment* on page I1-5100.

  Writes to this register are rare. In a system that supports two Security states, this register is writable only by Secure writes.
- A RO status register, CNTSR, that provides:
  - A bit that indicates whether the system counter is halted because of an asserted Halt-on-Debug signal.
  - A field that indicates the current update frequency of the system counter. This field can be polled to determine when a requested change to the update frequency has been made.
- Two contiguous 32-bit RW registers that hold the current system counter value, CNTCV. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

  The system counter must be disabled before writing to these registers, otherwise the effect of the write is UNPREDICTABLE.

  Writes to these registers are rare. In a system that supports two Security states, these registers are writable only by Secure writes.
- A table of one or more 32-bit entries, where:
  - The first entry defines the *base frequency* of the system counter. This is the maximum frequency at which the counter updates.
  - Each subsequent entry defines an alternative frequency of the system counter, that must be an exact divisor of the base frequency.

  A 32-bit zero entry immediately follows the last table entry.

  This table can be RO or RW. For more information, see *The frequency modes table* on page I1-5100.

In addition, the CNTReadBase frame includes a read-only copy of CNTCV, as two contiguous 32-bit RO registers. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

*Counter module control and status register summary* on page I1-5102 describes CNTReadBase and CNTControlBase memory maps, and the registers in each frame.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I1.2.1 Control of counter operating frequency and increment

The system counter has a fixed *base frequency*, and must maintain the required counter accuracy, meaning ARM recommends that it does not gain or lose more than ten seconds in a 24-hour period, see *System counter* on page D6-1899. However, the counter can increment at a lower frequency than the base frequency, using a correspondingly larger increment. For example, it can increment by four at a quarter of the base frequency. Any lower-frequency operation, and any switching between operating frequencies, must not reduce the accuracy of the counter.

Control of the system counter frequency and increment is provided only through the memory-mapped counter module. The following sections describe this control:

- *The frequency modes table*.
- *Changing the system counter and increment*.

#### The frequency modes table

The frequency modes table starts at offset `0x20` from CNTControlBase.

Table entries are 32-bits, and each entry specifies a system counter update frequency, in Hz.

The first entry in the table specifies the base frequency of the system counter.

When the system timer is operating at a lower frequency than the base frequency, the increment applied at each counter update is given by:

increment = (base_frequency) / (selected_frequency)

A 32-bit word of zero value marks the end of the table. That is, the word of memory immediately after the last entry in the table must be zero.

The only required entry in the table is the entry for the base frequency.

Typically, the frequency modes table is in RO memory. However, a system implementation might use RW memory for the table, and initialize the table entries as part of its startup sequence. Therefore, the CNTControlBase memory map shows the table region as RO or RW.

ARM strongly recommends that the frequency modes table is not updated once the system is running.

The architecture can support up to 1004 entries in the frequency modes table, and the maximum number of entries is IMPLEMENTATION DEFINED, up to this limit.

**Note**

ARM considers it likely that implementations will require significantly fewer entries than the architectural limit.

#### Changing the system counter and increment

The value of the CNTCR.FREQ field specifies which frequency modes table entry specifies the system counter update frequency.

Changing the value of CNTCR.FREQ requests a change to the system counter update frequency. To ensure the frequency change does not affect the overall accuracy of the counter, a change is made as follows:

- When changing from a higher frequency to a lower frequency, the counter:
  1. Continues running at the higher frequency until the count reaches an integer multiple of the required lower frequency.
  2. Switches to operating at the lower frequency.
- When changing from a lower frequency to a higher frequency, the counter:
  1. Waits until the end of the current lower-frequency cycle.
  2. Makes the counter increment required for operation at that lower frequency.
  3. Switches to operating at the higher frequency.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the frequency has changed, CNTSR is updated to indicate the new frequency. Therefore, a system component that is waiting for a frequency change can poll CNTSR to detect the change.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I1.3 Counter module control and status register summary

The Counter module control and status registers are memory-mapped registers in the following register memory *frames*:

- A control frame, with base address CNTControlBase.
- A status frame, with base address CNTReadBase.

Each of these register memory frames is at least 4KB in size, or is at least the size of the memory protection granule if this granule size is larger than 4KB. Similarly, each base address must be aligned to 4KB, or to the memory protection granule if that is larger than 4KB.

**Note**

The memory protection granule is either 4KB, 16KB or 64KB.

In each register memory frame, the memory at offset `0xFD0-0xFFF` is reserved for twelve 32-bit IMPLEMENTATION DEFINED ID registers, see the CounterID<n> register descriptions for more information.

The counter is assumed to be little-endian.

In an implementation that supports Secure and Non-secure memory spaces, CNTControlBase is implemented only in the Secure memory space.

Table I1-1 shows the CNTControlBase control registers, in order of their offsets from CNTControlBase. *Generic Timer memory-mapped register descriptions* on page I3-5173 describes each of these registers.

**Table I1-1 CNTControlBase memory map**

| Offset | Name | Type | Description |
|---|---|---|---|
| `0x000` | CNTCR | RW | Counter Control Register. |
| `0x004` | CNTSR | RO | Counter Status Register. |
| `0x008` | CNTCV[31:0] | RW | Counter Count Value register. |
| `0x00C` | CNTCV[63:32] | RW | |
| `0x010-0x01C` | - | RES0 | Reserved. |
| `0x020` | CNTFID0 | RO or RW | Frequency modes table, and end marker.<br>CNTFID0 is the base frequency, and each CNTFID*n* is an alternative frequency. For more information see *The frequency modes table* on page I1-5100. |
| `0x020`+4*n* | CNTFID*n* | RO or RW | |
| (`0x024`+4*n*)-`0x0BC` | - | RES0 | Reserved. |
| `0x0C0-0x0FC` | - | IMPLEMENTATION DEFINED | Reserved for IMPLEMENTATION DEFINED registers. |
| `0x100-0xFCC` | - | RES0 | Reserved. |
| `0xFD0-0xFFC` | CounterID<n> | RO | Counter ID registers 0-11. |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table I1-2 shows the CNTReadBase control registers, in order of their offsets from CNTReadBase. *Generic Timer memory-mapped register descriptions* on page I3-5173 describes each of these registers.

**Table I1-2 CNTReadBase memory map**

| Offset | Name | Type | Description |
|---|---|---|---|
| 0x000 | CNTCV[31:0] | RO | Counter Count Value register |
| 0x004 | CNTCV[63:32] | RO | |
| 0x008-0xFCC | - | RES0 | Reserved |
| 0xFD0-0xFFC | CounterID<n> | RO | Counter ID registers 0-11 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I1.4 Memory-mapped timer components

This part of the ARM Generic Timer specification defines an optional memory-mapped timer component. This can be placed close to any programmable system component that does not incorporate a System register mapped ARM Generic Timer, to provide equivalent time functionality to that system component.

The memory map consists of up to 8 timer *frames*. Each timer frame:

- Provides its own set of timers and associated interrupts.
- Is in its own memory protection region that is:
  - In its own memory protection region, with a system-defined size of 4KB, 16KB or 64KB.
  - At a start address that is aligned to 4KB.

  **Note**

  The 4KB alignment requirement applies regardless of the memory protection region size.

The base address of a frame is CNTBase*N*, where *N* numbers from 0 up to a maximum permitted value of 7.

**Note**

In the naming of the registers associated with a CNTBase*N* frame, the value of N is represented as <n>, for example CNTACR<n>.

For each implemented CNTBase*N* frame the system can optionally provide an EL0 view of the frame. The base address of this second view of the CNTBase*N* frame is CNTEL0Base*N*.

If CNTEL0Base*N* is implemented:

- All registers visible in CNTBase*N* are visible, except for CNTVOFF and CNTEL0ACR.
- The offsets of all visible registers are the same as their offsets in the CNTBase*N* frame.

In addition, the system provides a control frame at base address CNTCTLBase.

The memory protection region and alignment requirements for the CNTEL0Base*N* and CNTCTLBase frames are the same as the requirements for the CNTBase*N* frames.

The system defines the position of each frame in the memory map. This means the values of each of the CNTBase*N*, CNTEL0Base*N*, and CNTCTLBase base addresses is IMPLEMENTATION DEFINED.

The memory-mapped timers are assumed to be little-endian.

The following sections describe the implementation of a memory-mapped view of the counter and timer:

- *The CNTBaseN and CNTEL0BaseN frames* on page I1-5105.
- *The CNTCTLBase frame* on page I1-5107.
- *Providing a complete set of counter and timer features* on page I1-5108.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I1.5 The CNTBase*N* and CNTEL0Base*N* frames

Table I1-3 shows the CNTBase*N* registers, in order of their offsets from CNTBase*N*. Whether a frame includes a virtual timer is IMPLEMENTATION DEFINED. If it does not then memory at offsets `0x030-0x03C` is RAZ/WI. Except for CNTEL0ACR and the CounterID<n> registers, equivalent registers are also implemented in the System register interface to the Generic Timer.

*Generic Timer memory-mapped register descriptions* on page I3-5173 describes each of these registers.

**Table I1-3 CNTBase*N* memory map**

| Offset | Register | Type | Description |
|---|---|---|---|
| `0x000` | CNTPCT[31:0][a] | RO | Physical Count register. |
| `0x004` | CNTPCT[63:32][a] | RO | |
| `0x008` | CNTVCT[31:0][a] | RO | Virtual Count register. |
| `0x00C` | CNTVCT[63:32][a] | RO | |
| `0x010` | CNTFRQ[a] | RO[c] | Counter Frequency register. |
| `0x014` | CNTEL0ACR | RW[b] | Counter EL0 Access Control Register, optional in the CNTBase*N* memory map. |
| `0x018` | CNTVOFF[31:0][a] | RO[c] | Virtual Offset register. If CNTVOFF<n> in the CNTCTLBase frame is a RW register, retains the value of that register. Otherwise is RAZ. |
| `0x01C` | CNTVOFF[63:32][a] | RO[c] | |
| `0x020` | CNTP_CVAL[31:0][a] | RW | Physical Timer CompareValue register. |
| `0x024` | CNTP_CVAL[63:32][a] | RW | |
| `0x028` | CNTP_TVAL[a] | RW | Physical TimerValue register. |
| `0x02C` | CNTP_CTL[a] | RW | Physical Timer Control register. |
| `0x030` | CNTV_CVAL[31:0][a] | RW[b] | Virtual Timer CompareValue register, optional in the CNTBase*N* memory map. |
| `0x034` | CNTV_CVAL[63:32][a] | RW[b] | |
| `0x038` | CNTV_TVAL[a] | RW[b] | Virtual TimerValue register, optional in the CNTBase*N* memory map. |
| `0x03C` | CNTV_CTL[a] | RW[b] | Virtual Timer Control register, optional in the CNTBase*N* memory map. |
| `0x040-0xFCF` | - | RES0 | Reserved. |
| `0xFD0-0xFFC` | CounterID<n> | RO | Counter ID registers 0-11. |

a. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in *Generic Timer registers* on page D7-2276 and *Generic Timer registers* on page G6-4780. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

b. Address is reserved, RAZ/WI if register not implemented

c. The CNTCTLBase frame includes a RW view of this register.

CNTTIDR controls whether the CNTBase*N* and CNTEL0Base*N* frames are implemented.

For any value of *N*, the layout of the registers in the frame at CNTEL0Base*N* is identical to that at CNTBase*N*, except that, in the CNTEL0Base*N* frame:

- CNTVOFF is never visible, and the memory at `0x018-0x01C` is RAZ/WI.
- CNTEL0ACR is never visible, and the memory at `0x014` is RAZ/WI.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If implemented in the frame at CNTBase*N*, CNTEL0ACR controls whether CNTPCT, CNTVCT, CNTFRQ, the EL1 Physical Timer, and the Virtual Timer registers are visible in the frame at CNTEL0Base*N*.

  If CNTEL0ACR is not implemented then these registers are not visible in the frame at CNTEL0Base*N*, and their addresses are RAZ/WI.

If an implementation supports 64-bit atomic accesses, then CNTPCT, CNTVCT, CNTVOFF, CNTP_CVAL, and CNTV_CVAL must be accessible as atomic 64-bit values.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I1.6 The CNTCTLBase frame

The CNTCTLBase frame contains:

- An identification register for the features of the memory-mapped counter and timer implementation.
- Access controls for each CNTBase*N* frame.
- A virtual offset register for frames that implement a virtual timer.

Table I1-4 shows the CNTCTLBase registers, in order of their offsets from CNTCTLBase.

**Note**

CNTFRQ and CNTVOFF registers are also implemented in a System register interface to the Generic Timer.

*Generic Timer memory-mapped register descriptions* on page I3-5173 describes each of these registers.

**Table I1-4 CNTCTLBase memory map**

| Offset | Register | Type | Security[a] | Description |
|---|---|---|---|---|
| 0x000 | CNTFRQ[b] | RW | Secure | Counter Frequency register. |
| 0x004 | CNTNSAR | RW | Secure | Counter Non-Secure Access register. |
| 0x008 | CNTTIDR | RO | Both | Counter Timer ID register. |
| 0x00C- 0x03F | - | RES0 | - | Reserved. |
| 0x040+4*N*[c] | CNTACR<n> | RW | Configurable[d] | Counter Access Control register *N*. |
| 0x060- 0x07F | - | RES0 | - | Reserved. |
| 0x080+8*N*[c] | CNTVOFF<n>[31:0][b] | RW[e] | Configurable[d] | Virtual Offset register *N*. IMPLEMENTATION DEFINED whether this register is RW or RAZ. Optional in the CNTCTLBase memory map. |
| 0x084+8*N*[c] | CNTVOFF<n>[63:32][b] | RW[e] | | |
| 0x0C0-0x0FC | - | UNK/SBZP | - | Reserved. |
| 0x100-0x7FC | - | - | - | IMPLEMENTATION DEFINED. |
| 0x800-0xFBC | - | UNK/SBZP | - | Reserved. |
| 0xFC0-0xFCF | - | - | - | IMPLEMENTATION DEFINED. |
| 0xFD0- 0xFFC | CounterID<n> | RO | Both | Counter ID registers 0-11. |

a. Access security requirement in an implementation that supports two Security states. In an implementation that does not support multiple Security states all registers are accessible as shown in the *Type* column.

b. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in *Generic Timer registers* on page D7-2276 and *Generic Timer registers* on page G6-4780. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

c. Implemented for each value of *N* from 0 to 7 for which a CNTBase*N* frame is implemented.

d. The CNTNSAR determines the Non-secure accessibility of the CNTACR<n>s and the CNTVOFF<n>s in the CNTCTLBase frame. For more information, see the register descriptions.

e. Address is reserved, RAZ/WI if register not implemented.

All implementations of the Generic Timer include the virtual counter. Therefore, conceptually, all implementations include the CNTVOFF register that defines the virtual offset between the physical count and the virtual count. If a memory-mapped Generic Timer component does not distinguish between real time and virtual time then it can implement CNTVOFF as RAZ/WI. Otherwise CNTVOFF is a RW register, and ARM strongly recommends that the system only permits access to CNTVOFF from EL2 or higher.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I1.7 Providing a complete set of counter and timer features

Using the general model for implementing a memory-mapped interface to the Generic Timer described in this section, the feature set of a System registers counter and timer, in an implementation that includes EL2 and EL3, can be implemented using the following set of timer frames:

- A CNTCTLBase control frame.
- The following CNTBase*N* timer frames:

  **Frame 0** Accessible from Non-secure state, with second view and virtual capability. This provides the Non-secure EL1&0 timers.

  **Frame 1** Accessible from Non-secure state, with no second view and no virtual capability. This provides the Non-secure EL2 timers.

  **Frame 2** Accessible only Secure state, with a second view but no virtual capability. This provides the Secure PL1&0 timers, meaning:
  - Compared to a PE where EL3 is using AArch32, it provides the only Secure state timer.
  - Compared to a PE where EL3 is using AArch64, it provides the Secure EL1&0 timer.

  **Frame 3** Accessible only Secure state, with no second view and no virtual capability. This provides the Secure EL3 timers.

  **Note**

  This frame is not required for a memory-mapped timer that provides only the feature set of a PE for which EL3 is using AArch32.

In this implementation, the full set of implemented frames, and their configuration in the memory map, is as follows:

**CNTCTLBase**

The control frame. This frame is located in both Secure and Non-secure physical memory, and:
- In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is not accessible.

**CNTBase0** The first view of the Non-secure EL1&0 timers. This frame is located only in Non-secure physical memory, and:
- In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is accessible only at EL1.

**CNTEL0Base0**

The second view of CNTBase0, meaning it is the EL0 view of the Non-secure EL1&0 timers. This frame is located only in Non-secure physical memory, and:
- In the Secure EL1&0 translation regime, the architecture permits this frame to be accessible at El1, or at EL1 and EL0, but does not require either of these options.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is accessible at EL1 and EL0.

**CNTBase1** The first and only view of the Non-secure EL2 timers. This frame is located only in Non-secure physical memory, and:
- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
  - In the Secure EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible only at EL3.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is not accessible.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**CNTBase2** The first view of the Secure PL1&0 timers. This frame is located only in Secure physical memory, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
  - In the Secure EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible only at EL3.
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

**CNTEL0Base2**

The second view of CNTBase2, meaning it is the EL0 view of the Secure PL1&0 timers. This frame is located only in Secure physical memory, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible at EL1 and EL0.
  - In the Secure EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible at EL3 and EL0.
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

**CNTBase3** The first and only view of the Non-secure EL3 timers. This frame is located only in Secure physical memory, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is not accessible.
  - In the Secure EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, this frame is not accessible.
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

**Note**

*About the Virtual Memory System Architecture (VMSA)* on page D4-1742 describes the VMSAv8-64 translation regimes, and *About VMSAv8-32* on page G4-3992 describes the VMSAv8-32 translation regimes.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I1.8 Gray-count scheme for timer distribution scheme

The distribution of the Counter value using a Gray-code provides a relatively simple mechanism to avoid any danger of the count being sampled with an intermediate value even if the clocking is asynchronous. It has a further advantage that the distribution is relatively low power, since only one bit changes on the main distribution wires for each clock tick.

A suitable Gray-coding scheme can be achieved with the following logic:

Gray[N] = Count[N]

Gray[i] = (XOR(Gray[N:i+1])) XOR Count[i] for N–1 >= i >= 0

Count[i] = XOR(Gray[N:i]) for N >= i >= 0

This is for an N+1 bit counter, where Count is a conventional binary count value, and Gray is the corresponding Gray count value.

**Note**

This scheme has the advantage of being relatively simple to switch, in either direction, between operating with low-frequency and low-precision, and operating with high-frequency and high-precision. To achieve this, the ratio of the frequencies must be $2^n$, where n is an integer. A switch-over can occur only on the 2n+1 boundary to avoid losing the Gray-coding property on a switch-over.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter I2
# Recommended Memory-mapped Interfaces to the Performance Monitors

This chapter describes the recommended memory-mapped and external debug interfaces to the Performance Monitors. It contains the following section:

- *About the memory-mapped views of the Performance Monitors registers* on page I2-5112.

**Note**

*Performance Monitors memory-mapped register descriptions* on page I3-5121 describes the memory-mapped registers for the Performance Monitors.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I2.1 About the memory-mapped views of the Performance Monitors registers

An implementation can provide:

- Memory-mapped access to the Performance Monitors registers. Software running on any processor in a system can use this interface to access counters in the Performance Monitors.
- Access to the Performance Monitors registers through an external debug interface. A debugger can use this interface to access counters in the Performance Monitors.

  ARM recommends that any external debug interface is implemented as defined in the *ARM Debug Interface v5 Architecture Specification*.

  An external debug interface provides a memory-mapped view of the Performance Monitors registers.

*Performance Monitors memory-mapped registers summary* on page I3-5119 gives the memory map of these registers.

The following sections describe the memory-mapped views of the Performance Monitors registers:

- *Differences in the memory-mapped views of the Performance Monitors registers*.
- *Synchronization of changes to the memory-mapped views*.
- *Access permissions for memory-mapped views of the Performance Monitors* on page I2-5113.

In this section, unless the context explicitly indicates otherwise, any reference to *a memory-mapped view* applies equally to a register view using:

- A access through an external debug interface.
- A memory-mapped access.

### I2.1.1 Differences in the memory-mapped views of the Performance Monitors registers

A memory-mapped view of the Performance Monitors registers accesses the same registers as the System registers interface described in *Performance Monitors Extension registers* on page D5-1893, except that:

1. The PMSELR is accessible only in the System registers interface.
2. The PMCFGR, PMLAR, PMLSR, PMAUTHSTATUS, PMDEVARCH, PMDEVTYPE, PMPIDR0, PMPIDR1, PMPIDR2, PMPIDR3, PMPIDR4, PMCIDR0, PMCIDR1, PMCIDR2, and PMCIDR3 registers are accessible only in memory-mapped views. *Performance Monitors memory-mapped register descriptions* on page I3-5121 describes these registers.
3. The following controls do not affect the memory-mapped view:
   - PMSELR.
   - PMUSERENR.
   - HDCR.{TPM, TPMCR, HPMN}.

   Instead, see the register descriptions in Chapter I3 *Memory-Mapped System Control Register Descriptions*.

### I2.1.2 Synchronization of changes to the memory-mapped views

If a Performance Monitor is visible in both System register and a memory-mapped views, and is accessed simultaneously through these two mechanisms, the behavior must be as if the access occurred atomically in any order. For more information, see *Synchronization of changes to the external debug registers* on page H8-4946.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I2.1.3 Access permissions for memory-mapped views of the Performance Monitors

For more information, see *External debug interface register access permissions* on page H8-4952.

Table I2-1 shows the access permissions for the Performance Monitors registers in a v8 Debug implementation. This table uses the following terms:

**DLK** When the OS Double Lock is locked, EDPRSR.DLK == 1, accesses to some registers produce an error. Applies to both interfaces.

**EPMAD** When `AllowExternalPMUAccess()` == FALSE, external debug access is disabled. See also *Behavior of a not permitted memory-mapped access* on page H8-4951.

**Error** Indicates that the access gives an error response.

**Default** This shows the default access permissions, if none of the conditions in this list prevent access to the register.

**Off** When EDPRSR.PU == 0, the Core power domain is completely off, or in a low-power state where the Core power domain registers cannot be accessed.

**Note**

If debug power is off, then all external debug interface accesses return an error.

**OSLK** When the OS Lock is locked, OSLAR_EL1.OSLK == 1, accesses to some registers produces an error. This column shows the effect of this control on accesses using the external debug interface.

**SLK** This indicates the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software lock is locked. See *Register access permissions for memory-mapped accesses* on page H8-4950.

For all other accesses, this column is ignored.

**-** Indicates that the control has no effect on the behavior of the access:

- If no other control affects the behavior, the Default access behavior applies.
- However, another control might determine the behavior.

**Table I2-1 Access permissions for the Performance Monitors registers**

| Offset | Register | Domain | Off | DLK | OSLK | EPMAD | Default | SLK |
|---|---|---|---|---|---|---|---|---|
| 0x000+8xn | PMEVCNTR<n>_EL0[a] | Core | Error | Error | Error | Error | RW | RO |
| 0x0F8 | PMCCNTR_EL0[31:0] | Core | Error | Error | Error | Error | RW | RO |
| 0x0FC | PMCCNTR_EL0[63:32] | Core | Error | Error | Error | Error | RW | RO |
| 0x400+4xn | PMEVTYPER<n>_EL0[a] | Core | Error | Error | Error | Error | RW | RO |
| 0x47C | PMCCFILTR_EL0 | Core | Error | Error | Error | Error | RW | RO |
| 0x600-0x6FC | - | - | Access is IMPLEMENTATION DEFINED | | | | | |
| 0xA00-0xBFC | - | - | Access is IMPLEMENTATION DEFINED | | | | | |
| 0xC00 | PMCNTENSET_EL0 | Core | Error | Error | Error | Error | RW | RO |
| 0xC20 | PMCNTENCLR_EL0 | Core | Error | Error | Error | Error | RW | RO |
| 0xC40 | PMINTENSET_EL1 | Core | Error | Error | Error | Error | RW | RO |
| 0xC60 | PMINTENCLR_EL1 | Core | Error | Error | Error | Error | RW | RO |
| 0xC80 | PMOVSCLR_EL0 | Core | Error | Error | Error | Error | RW | RO |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table I2-1 Access permissions for the Performance Monitors registers (continued)**

| Offset | Register | Domain | Off | DLK | OSLK | EPMAD | Default | SLK |
|---|---|---|---|---|---|---|---|---|
| 0xCA0 | PMSWINC_EL0[b] | Core | Error | Error | Error | Error | WO | WI |
| 0xCC0 | PMOVSSET_EL0 | Core | Error | Error | Error | Error | RW | RO |
| 0xD80-0xDFC | - | - | Access is IMPLEMENTATION DEFINED | | | | | |
| 0xE00 | PMCFGR | Core | Error | Error | Error | Error | RO | RO |
| 0xE04 | PMCR_EL0 | Core | Error | Error | Error | Error | RW | RO |
| 0xE20 | PMCEID0_EL0 | Core | Error | Error | Error | Error | RO | RO |
| 0xE24 | PMCEID1_EL0 | Core | Error | Error | Error | Error | RO | RO |
| 0xE80-0xEFC | Integration registers | - | Access is IMPLEMENTATION DEFINED | | | | | |
| 0xF00-0xFFC | *Management registers and CoreSight compliance* on page K2-5523 | | | | | | | |

a. Implemented counters only. *n* is the counter number.
b. Only if the OPTIONAL PMSWINC_EL0 register is implemented in the external debug interface.

### I2.1.4 Power domains and Performance Monitors registers reset

For ARMv8-A implementations, ARM recommends that Performance Monitors are implemented as part of the core power domain, not as part of a separate debug power domain. There is no interface to access the Performance Monitors registers when the core power domain is powered down.

A Warm or Cold reset sets the Performance Monitors registers to their reset values. An External Debug reset does not change the values of the Performance Monitors registers.

For more information about the reset scheme recommended for a v8 Debug implementation see Chapter H6 *Debug Reset and Powerdown Support*.

Table I2-2 on page I2-5115 shows the Performance Monitors register resets for writable register fields. The column headings use the following terms:

**64** This is the architectural reset value when resetting into AArch64 state.

**32** This is the architectural reset value when resetting into AArch32 state.

**-** This indicates an IMPLEMENTATION DEFINED reset value on the specified reset. This might be UNKNOWN.

**Note**

This table does not include:

- Read-only identification registers and fields that have a fixed value. In this case, the reset value is that fixed value. An example of this is PMCR_EL0.N.
- Write-only registers and fields that only have an effect on writes. These do not have a reset value. An example of this is PMSWINC_EL0.
- IMPLEMENTATION DEFINED registers. In this case, the reset domains are IMPLEMENTATION DEFINED. The reset values are IMPLEMENTATION DEFINED and might be UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table I2-2 Performance Monitors System register resets**

| Register | Domain | Field | 64 | 32 | Description |
|---|---|---|---|---|---|
| PMCR_EL0 | Warm | DP | - | 0 | Disable PMCCNTR_EL0 when prohibited |
| | | X | - | 0 | Export enable |
| | | D | - | 0 | Clock divider |
| | | E | 0 | 0 | Performance Monitors enable |
| PMCNTENSET_EL0<br>PMCNTENCLR_EL0 | Warm | - | - | - | All fields in register |
| PMOVSSET_EL0<br>PMOVSCLR_EL0 | Warm | - | - | - | All fields in register |
| PMSELR_EL0 | Warm | SEL | - | - | Selected event counter |
| PMCCNTR_EL0 | Warm | - | - | - | All fields in register |
| PMEVTYPER<n>_EL0 | Warm | - | - | - | All fields in register |
| PMCCFILTR_EL0 | Warm | [31:26] | - | 0x00 | PMCCNTR_EL0 filtering controls |
| PMEVCNTR<n>_EL0 | Warm | - | - | - | All fields in register |
| PMUSERENR_EL0 | Warm | ER | - | 0 | Enable counter read access in EL0 |
| | | CR | - | 0 | Enable PMCCNTR_EL0 read access in EL0 |
| | | SW | - | 0 | Enable PMSWINC_EL0 write access in EL0 |
| | | EN | - | 0 | Enable Performance Monitors access in EL0 |
| PMINTENSET_EL1<br>PMINTENCLR_EL1 | Warm | - | - | - | All fields in register |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Chapter I3
# Memory-Mapped System Control Register Descriptions

This chapter describes the memory-mapped system control registers. It contains the following sections:


- *About the memory-mapped system control register descriptions* on page I3-5118.
- *Performance Monitors memory-mapped registers summary* on page I3-5119.
- *Performance Monitors memory-mapped register descriptions* on page I3-5121.
- *Generic Timer memory-mapped registers overview* on page I3-5172.
- *Generic Timer memory-mapped register descriptions* on page I3-5173.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## I3.1 About the memory-mapped system control register descriptions

This chapter describes the memory-mapped system control registers other than the memory-mapped debug registers. That is, it describes:

**The memory-mapped view of the Performance Monitors registers**

- *Performance Monitors memory-mapped registers summary* on page I3-5119 lists these registers and describes their memory map.
- *Performance Monitors memory-mapped register descriptions* on page I3-5121 describes each of the memory-mapped registers.

Chapter I2 *Recommended Memory-mapped Interfaces to the Performance Monitors* describes the recommended interface to these registers.

**Note**

Chapter D5 *The Performance Monitors Extension* describes the Performance Monitors. The following sections describe the System register interfaces to the Performance Monitors:

- *Performance Monitors registers* on page D7-2236, for accesses from an Exception level that is using AArch64.
- *Performance Monitors registers* on page G6-4737, for accesses from an Exception level that is using AArch32.

**The registers for the system-level Generic Timer component**

Any implementation that includes the Generic Timer must include the memory-mapped system-level component described in Chapter I1 *System Level Implementation of the Generic Timer*. In this chapter:

- *Generic Timer memory-mapped registers overview* on page I3-5172 gives an overview of the registers, referring to Chapter I1 for more information.
- *Generic Timer memory-mapped register descriptions* on page I3-5173 describes each of the memory-mapped registers.

**Note**

Chapter D6 *The Generic Timer in AArch64 state* describes the Generic Timer component that is accessible using the System registers. The following sections describe the System register interfaces to that component:

- *Generic Timer registers* on page D7-2276, for accesses from an Exception level that is using AArch64.
- *Generic Timer registers* on page G6-4780, for accesses from an Exception level that is using AArch32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I3.2 Performance Monitors memory-mapped registers summary

The locations of the registers in the memory-mapped view of the Performance Monitors are defined as offsets from a system-defined base address. *Performance Monitors memory-mapped register views* defines this memory map.

### I3.2.1 Performance Monitors memory-mapped register views

Table I3-1 shows the memory-mapped view of the Performance Monitors registers. All other entries are reserved.

**Note**

- Counters that are reserved because HDCR.HPMN has been changed from its reset value remain visible in any memory-mapped view.
- The registers that relate to an implemented event counter, PMN*x*, are PMEVCNTR<n> and PMEVTYPER<n>.
- The mapping of the *Performance Monitors Event Counter Registers*, at offsets `0x000-0x0F4`, has changed compared to the mappings of the equivalent registers in ARMv7.

Each entry in the Name column links to the register description in *Performance Monitors memory-mapped register descriptions* on page I3-5121, and:

- If the *System register?* column of the table shows that the register is a System register, the memory-mapped interface provides a view of the System register described in:
  - *Performance Monitors registers* on page D7-2236, for the AArch64 System register
  - *Performance Monitors registers* on page G6-4737, for the AArch32 System register
- Otherwise, the register is accessible only using the memory-mapped interface.

**Table I3-1 Performance Monitors memory-mapped register views**

| Offset | Type | Name | Description | System register? |
|---|---|---|---|---|
| `0x000`+8n | RW | PMEVCNTR<n>_EL0 | Performance Monitors Event Counter Register. | Yes |
| `0x0F8`<br>`0x0FC` | RW | PMCCNTR_EL0[31:0]<br>PMCCNTR_EL0[63:32] | Performance Monitors Cycle Counter Register[a] | Yes |
| `0x400`+4n | RW | PMEVTYPER<n>_EL0 | Performance Monitors Event Type and Filter Register. | Yes |
| `0x47C` | RW | PMCCFILTR_EL0 | Performance Monitors Cycle Counter Filter Register | Yes |
| `0x600-0x6FC` | - | - | IMPLEMENTATION DEFINED | - |
| `0xA00-0xBFC` | - | - | IMPLEMENTATION DEFINED | - |
| `0xC00` | RW | PMCNTENSET_EL0 | Performance Monitors Count Enable Set register | Yes |
| `0xC20` | RW | PMCNTENCLR_EL0 | Performance Monitors Count Enable Clear register | Yes |
| `0xC40` | RW | PMINTENSET_EL1 | Performance Monitors Interrupt Enable Set register | Yes |
| `0xC60` | RW | PMINTENCLR_EL1 | Performance Monitors Interrupt Enable Clear register | Yes |
| `0xC80` | RW | PMOVSCLR_EL0 | Performance Monitors Overflow Flag Status Clear register | Yes |
| `0xCA0` | WO | PMSWINC_EL0 | Performance Monitors Software Increment register | Yes |
| `0xCC0` | RW | PMOVSSET_EL0 | Performance Monitors Overflow Flag Status Set register | Yes |
| `0xD80-0xDFC` | - | - | IMPLEMENTATION DEFINED | - |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table I3-1 Performance Monitors memory-mapped register views (continued)**

| Offset | Type | Name | Description | System register? |
|---|---|---|---|---|
| 0xE00 | RO | PMCFGR | Performance Monitors Configuration Register | No |
| 0xE04 | RW | PMCR_EL0 | Performance Monitors Control Register | Yes |
| 0xE20 | RO | PMCEID0_EL0 | Performance Monitors Common Event Identification register 0 | Yes |
| 0xE24 | RO | PMCEID1_EL0 | Performance Monitors Common Event Identification register 1 | Yes |
| 0xE80-0xEFC | - | - | IMPLEMENTATION DEFINED | - |
| 0xF00 | RW | PMITCTRL[b] | Integration Model Control registers | No |
| 0xFA8 | RO | PMDEVAFF0[b] | Device Affinity registers | No |
| 0xFAC | RO | PMDEVAFF1[b] | | |
| 0xFB0 | WO | PMLAR[b, c] | Lock Access register | No |
| 0xFB4 | RO | PMLSR[b, c] | Lock Status register | No |
| 0XFB8 | RO | PMAUTHSTATUS[b] | Authentication Status register | No |
| 0xFBC | RO | PMDEVARCH[b] | Device Architecture register | No |
| 0xFCC | RO | PMDEVTYPE[b] | Device Type register | No |
| 0xFD0 | RO | PMPIDR4[b] | Peripheral ID registers | No |
| 0xFE0 | RO | PMPIDR0[b] | | |
| 0xFE4 | RO | PMPIDR1[b] | | |
| 0xFE8 | RO | PMPIDR2[b] | | |
| 0xFEC | RO | PMPIDR3[b] | | |
| 0xFF0 | RO | PMCIDR0[b] | Component ID registers | No |
| 0xFF4 | RO | PMCIDR1[b] | | |
| 0xFF8 | RO | PMCIDR2[b] | | |
| 0xFFC | RO | PMCIDR3[b] | | |

a. The interface must support at least single-copy atomic 32-bit accesses. If single-copy atomic 64-bit access to the registers is not possible, software must use a high-low-high read access to read the counter value if the counter is enabled.

b. CoreSight interface registers, see *Management registers and CoreSight compliance* on page K2-5523.

c. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## I3.3 Performance Monitors memory-mapped register descriptions

This section describes the Performance Monitoring registers. *Performance Monitors memory-mapped registers summary* on page I3-5119 lists these registers in their memory map.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### I3.3.1 PMAUTHSTATUS, Performance Monitors Authentication Status register

The PMAUTHSTATUS characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for Performance Monitors.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMAUTHSTATUS is in the Debug power domain.

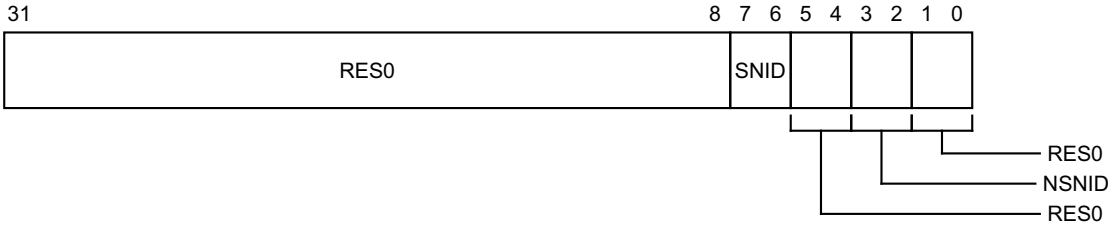
This register is OPTIONAL, and is required for CoreSight compliance. ARM recommends that this register is implemented.

**Attributes**

PMAUTHSTATUS is a 32-bit register.

## Field descriptions

The PMAUTHSTATUS bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**SNID, bits [7:6]**

Holds the same value as DBGAUTHSTATUS_EL1.SNID.

**Bits [5:4]**

Reserved, RES0.

**NSNID, bits [3:2]**

Holds the same value as DBGAUTHSTATUS_EL1.NSNID.

**Bits [1:0]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMAUTHSTATUS:

PMAUTHSTATUS can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xFB8 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.2 PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register

The PMCCFILTR_EL0 characteristics are:

**Purpose**

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register PMCCFILTR_EL0 is architecturally mapped to AArch64 System register PMCCFILTR_EL0.

External register PMCCFILTR_EL0 is architecturally mapped to AArch32 System register PMCCFILTR.

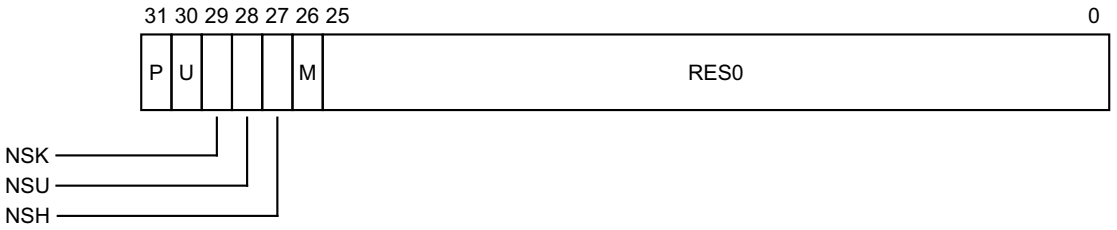
PMCCFILTR_EL0 is in the Core power domain.

**Attributes**

PMCCFILTR_EL0 is a 32-bit register.

## Field descriptions

The PMCCFILTR_EL0 bit assignments are:




**P, bit [31]**

EL1 modes filtering bit. Controls counting in EL1. If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the NSK bit. The possible values of this bit are:

`0` Count cycles in EL1.

`1` Do not count cycles in EL1.

**U, bit [30]**

EL0 filtering bit. Controls counting in EL0. If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the NSU bit. The possible values of this bit are:

`0` Count cycles in EL0.

`1` Do not count cycles in EL0.

**NSK, bit [29]**

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, cycles in Non-secure EL1 are counted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Otherwise, cycles in Non-secure EL1 are not counted.

**NSU, bit [28]**

Non-secure User mode filtering bit. Controls counting in Non-secure EL0. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of U, cycles in Non-secure EL0 are counted.

Otherwise, cycles in Non-secure EL0 are not counted.

**NSH, bit [27]**

Non-secure Hyp modes filtering bit. Controls counting in Non-secure EL2. If EL2 is not implemented, this bit is RES0.

`0` Do not count cycles in EL2.

`1` Count cycles in EL2.

**M, bit [26]**

Secure EL3 filtering bit. Most applications can ignore this bit and set the value to zero. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, cycles in Secure EL3 are counted.

Otherwise, cycles in Secure EL3 are not counted.

**Bits [25:0]**

Reserved, RES0.

### Accessing the PMCCFILTR_EL0:

PMCCFILTR_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0x47C |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.3 PMCCNTR_EL0, Performance Monitors Cycle Counter

The PMCCNTR_EL0 characteristics are:

**Purpose**

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See *Time as measured by the Performance Monitors cycle counter* on page D5-1858 for more information.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register PMCCNTR_EL0 is architecturally mapped to AArch64 System register PMCCNTR_EL0.

External register PMCCNTR_EL0 is architecturally mapped to AArch32 System register PMCCNTR.

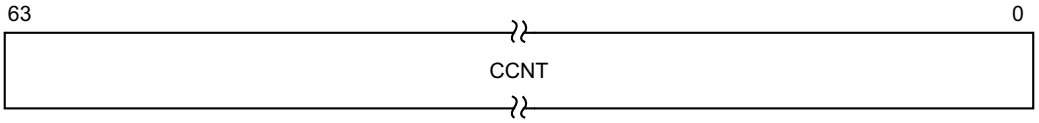
PMCCNTR_EL0 is in the Core power domain.

**Attributes**

PMCCNTR_EL0 is a 64-bit register.

#### Field descriptions

The PMCCNTR_EL0 bit assignments are:




**CCNT, bits [63:0]**

Cycle count. Depending on the values of PMCR_EL0.{LC,D}, the cycle count increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR_EL0.C sets this field to 0.

#### Accessing the PMCCNTR_EL0:

PMCCNTR_EL0[31:0] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0x0F8 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

PMCCNTR_EL0[63:32] can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0x0FC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.4 PMCEID0_EL0, Performance Monitors Common Event Identification register 0

The PMCEID0_EL0 characteristics are:

**Purpose**

Defines which common architectural and common microarchitectural feature events are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RO |

**Configurations**

External register PMCEID0_EL0 is architecturally mapped to AArch64 System register PMCEID0_EL0.

External register PMCEID0_EL0 is architecturally mapped to AArch32 System register PMCEID0.

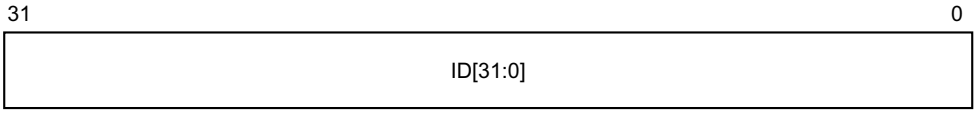
PMCEID0_EL0 is in the Core power domain.

**Attributes**

PMCEID0_EL0 is a 32-bit register.

#### Field descriptions

The PMCEID0_EL0 bit assignments are:




**ID[31:0], bits [31:0]**

PMCEID0_EL0[n] maps to event n. For a list of event numbers and descriptions, see *Events, Event numbers and mnemonics* on page D5-1872.

For each bit:

`0` The common event is not implemented.

`1` The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

#### Accessing the PMCEID0_EL0:

PMCEID0_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xE20` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.5 PMCEID1_EL0, Performance Monitors Common Event Identification register 1

The PMCEID1_EL0 characteristics are:

**Purpose**

Defines which common architectural and common microarchitectural feature events are implemented. If a particular bit is set to 1, then the event for that bit is implemented.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RO |

**Configurations**

External register PMCEID1_EL0 is architecturally mapped to AArch64 System register PMCEID1_EL0.

External register PMCEID1_EL0 is architecturally mapped to AArch32 System register PMCEID1.

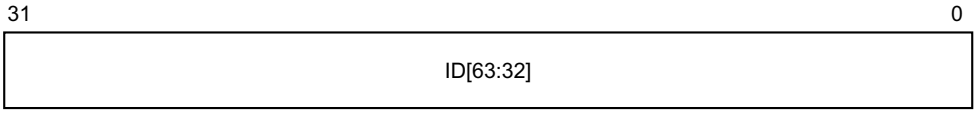
PMCEID1_EL0 is in the Core power domain.

**Attributes**

PMCEID1_EL0 is a 32-bit register.

#### Field descriptions

The PMCEID1_EL0 bit assignments are:




**ID[63:32], bits [31:0]**

PMCEID1_EL0[n] maps to event (n + 32). For a list of event numbers and descriptions, see *Events, Event numbers and mnemonics* on page D5-1872.

For each bit:

`0` The common event is not implemented.

`1` The common event is implemented.

Bits that map to reserved event numbers are reserved to identify events that might be defined in future revisions to the architecture.

Events that do not require additional features in the PMU can be defined retrospectively, meaning that they can be implemented as part of a PMUv3 implementation.

#### Accessing the PMCEID1_EL0:

PMCEID1_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xE24` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.6 PMCFGR, Performance Monitors Configuration Register

The PMCFGR characteristics are:

**Purpose**

Contains PMU-specific configuration data.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RO |

**Configurations**

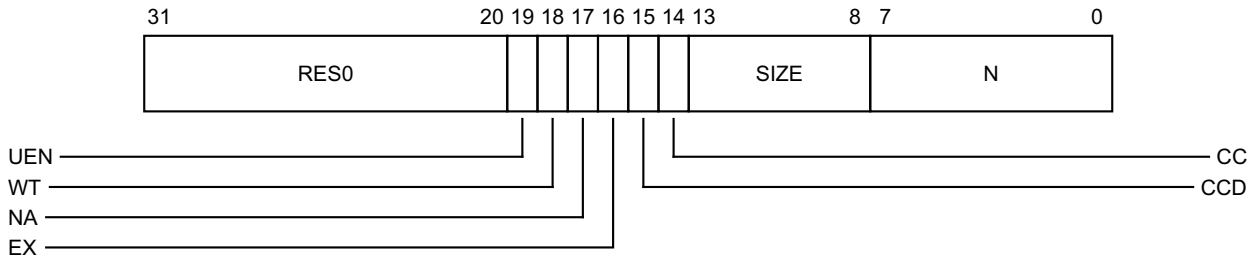
PMCFGR is in the Core power domain.

**Attributes**

PMCFGR is a 32-bit register.

## Field descriptions

The PMCFGR bit assignments are:




**Bits [31:20]**

Reserved, RES0.

**UEN, bit [19]**

User-mode Enable Register supported. PMUSERENR_EL0 is not visible in the external debug interface, so this bit is RES0.

**WT, bit [18]**

This feature is not supported, so this bit is RES0.

**NA, bit [17]**

This feature is not supported, so this bit is RES0.

**EX, bit [16]**

Export supported. Value is IMPLEMENTATION DEFINED.

`0` PMCR_EL0.X is RES0.

`1` PMCR_EL0.X is read/write.

**CCD, bit [15]**

Cycle counter has prescale. This is RES1 if AArch32 is supported at any EL, and RES0 otherwise.

`0` PMCR_EL0.D is RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 PMCR_EL0.D is read/write.

**CC, bit [14]**

Dedicated cycle counter (counter 31) supported. This bit is RES1.

**SIZE, bits [13:8]**

Size of counters. This field determines the spacing of counters in the memory-map.

In ARMv8 the counters are at doubleword-aligned addresses, and the largest counter is 64-bits, so this field is `0b111111`.

**N, bits [7:0]**

Number of counters implemented in addition to the cycle counter, PMCCNTR_EL0. The maximum number of event counters is 31, so bits[7:5] are always RES0.

`00000000` Only PMCCNTR_EL0 implemented.

`00000001` PMCCNTR_EL0 plus one event counter implemented.

and so on up to `00011111`, which indicates PMCCNTR_EL0 and 31 event counters implemented.

## Accessing the PMCFGR:

PMCFGR can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xE00` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.7 PMCIDR0, Performance Monitors Component Identification Register 0

The PMCIDR0 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMCIDR0 is in the Debug power domain.

PMCIDR0 is optional to implement in the external register interface.

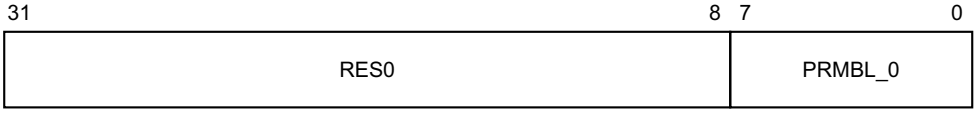
This register is required for CoreSight compliance.

**Attributes**

PMCIDR0 is a 32-bit register.

#### Field descriptions

The PMCIDR0 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PRMBL_0, bits [7:0]**

Preamble. Must read as `0x0D`.

#### Accessing the PMCIDR0:

PMCIDR0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xFF0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.8 PMCIDR1, Performance Monitors Component Identification Register 1

The PMCIDR1 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMCIDR1 is in the Debug power domain.

PMCIDR1 is optional to implement in the external register interface.

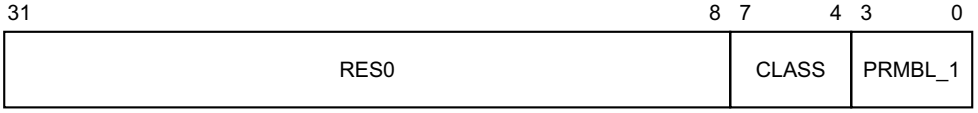
This register is required for CoreSight compliance.

**Attributes**

PMCIDR1 is a 32-bit register.

#### Field descriptions

The PMCIDR1 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**CLASS, bits [7:4]**

Component class. Reads as `0x9`, debug component.

**PRMBL_1, bits [3:0]**

Preamble. RAZ.

#### Accessing the PMCIDR1:

PMCIDR1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xFF4` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.9 PMCIDR2, Performance Monitors Component Identification Register 2

The PMCIDR2 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMCIDR2 is in the Debug power domain.

PMCIDR2 is optional to implement in the external register interface.

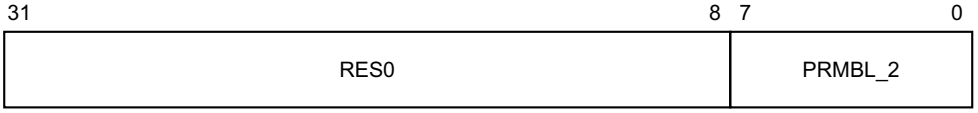
This register is required for CoreSight compliance.

**Attributes**

PMCIDR2 is a 32-bit register.

#### Field descriptions

The PMCIDR2 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PRMBL_2, bits [7:0]**

Preamble. Must read as `0x05`.

#### Accessing the PMCIDR2:

PMCIDR2 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xFF8` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.10 PMCIDR3, Performance Monitors Component Identification Register 3

The PMCIDR3 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMCIDR3 is in the Debug power domain.

PMCIDR3 is optional to implement in the external register interface.

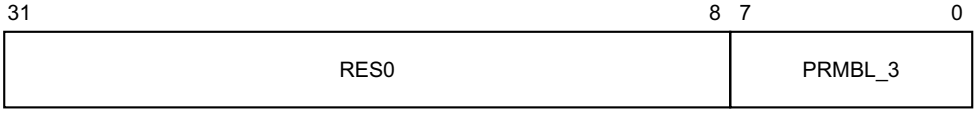
This register is required for CoreSight compliance.

**Attributes**

PMCIDR3 is a 32-bit register.

#### Field descriptions

The PMCIDR3 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PRMBL_3, bits [7:0]**

Preamble. Must read as `0xB1`.

#### Accessing the PMCIDR3:

PMCIDR3 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xFFC` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.11 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register

The PMCNTENCLR_EL0 characteristics are:

**Purpose**

Disables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<x>. Reading this register shows which counters are enabled.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register PMCNTENCLR_EL0 is architecturally mapped to AArch64 System register PMCNTENCLR_EL0.

External register PMCNTENCLR_EL0 is architecturally mapped to AArch32 System register PMCNTENCLR.

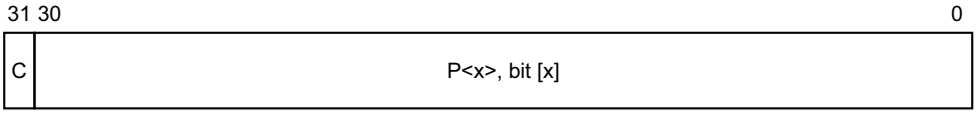
PMCNTENCLR_EL0 is in the Core power domain.

**Attributes**

PMCNTENCLR_EL0 is a 32-bit register.

## Field descriptions

The PMCNTENCLR_EL0 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 disable bit. Disables the cycle counter register. Possible values are:

0 When read, means the cycle counter is disabled. When written, has no effect.

1 When read, means the cycle counter is enabled. When written, disables the cycle counter.

**P<x>, bit [x], for x = 0 to 30**

Event counter disable bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI. N is the value in PMCR_EL0.N.

Possible values of each bit are:

0 When read, means that PMEVCNTR<x> is disabled. When written, has no effect.

1 When read, means that PMEVCNTR<x> is enabled. When written, disables PMEVCNTR<x>.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMCNTENCLR_EL0:

PMCNTENCLR_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xC20 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.12 PMCNTENSET_EL0, Performance Monitors Count Enable Set register

The PMCNTENSET_EL0 characteristics are:

**Purpose**

Enables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<x>. Reading this register shows which counters are enabled.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register PMCNTENSET_EL0 is architecturally mapped to AArch64 System register PMCNTENSET_EL0.

External register PMCNTENSET_EL0 is architecturally mapped to AArch32 System register PMCNTENSET.

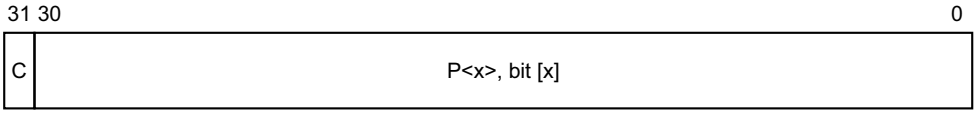
PMCNTENSET_EL0 is in the Core power domain.

**Attributes**

PMCNTENSET_EL0 is a 32-bit register.

#### Field descriptions

The PMCNTENSET_EL0 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 enable bit. Enables the cycle counter register. Possible values are:

0 When read, means the cycle counter is disabled. When written, has no effect.

1 When read, means the cycle counter is enabled. When written, enables the cycle counter.

**P<x>, bit [x], for x = 0 to 30**

Event counter enable bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI. N is the value in PMCR_EL0.N.

Possible values of each bit are:

0 When read, means that PMEVCNTR<x> is disabled. When written, has no effect.

1 When read, means that PMEVCNTR<x> event counter is enabled. When written, enables PMEVCNTR<x>.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMCNTENSET_EL0:

PMCNTENSET_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xC00 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.13 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

**Purpose**

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register PMCR_EL0[6:0] is architecturally mapped to AArch32 System register PMCR[6:0].

External register PMCR_EL0[6:0] is architecturally mapped to AArch64 System register PMCR_EL0[6:0].

PMCR_EL0 is in the Core power domain.

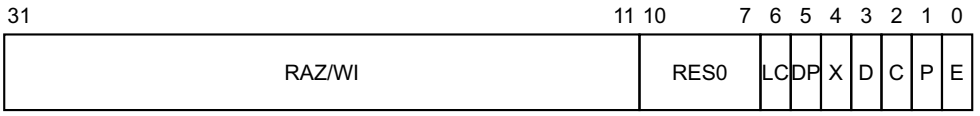
This register is only partially mapped to the internal PMCR System register. An external agent must use other means to discover the information held in PMCR[31:11], such as accessing PMCFGR and the ID registers.

**Attributes**

PMCR_EL0 is a 32-bit register.

## Field descriptions

The PMCR_EL0 bit assignments are:




**Bits [31:11]**

Reserved, RAZ/WI.

**Bits [10:7]**

Reserved, RES0.

**LC, bit [6]**

Long cycle counter enable. Determines which PMCCNTR_EL0 bit generates an overflow recorded by PMOVSR[31].

`0` Cycle counter overflow on increment that changes PMCCNTR_EL0[31] from 1 to 0.

`1` Cycle counter overflow on increment that changes PMCCNTR_EL0[63] from 1 to 0.

ARM deprecates use of PMCR_EL0.LC = 0.

In an AArch64-only implementation, this field is RES1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**DP, bit [5]**

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

0 PMCCNTR_EL0, if enabled, counts when event counting is prohibited.

1 PMCCNTR_EL0 does not count when event counting is prohibited.

Event counting is prohibited when ProfilingProhibited(IsSecure(),PSTATE.EL) == TRUE.

This bit is RW.

If EL3 is not implemented, this field is RES0.

**X, bit [4]**

Enable export of events in an IMPLEMENTATION DEFINED event stream. The possible values of this bit are:

0 Do not export events.

1 Export events where not prohibited.

This bit is used to permit events to be exported to another debug device, such as an OPTIONAL trace macrocell, over an event bus. If the implementation does not include such an event bus, this bit is RAZ/WI.

This bit does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If the implementation does not include an exported event stream, this bit is RAZ/WI. Otherwise this bit is RW.

**D, bit [3]**

Clock divider. The possible values of this bit are:

0 When enabled, PMCCNTR_EL0 counts every clock cycle.

1 When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

This bit is RW.

If PMCR_EL0.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

ARM deprecates use of PMCR.D = 1.

In an AArch64-only implementation, this field is RES0.

**C, bit [2]**

Cycle counter reset. This bit is WO. The effects of writing to this bit are:

0 No action.

1 Reset PMCCNTR_EL0 to zero.

This bit is always RAZ.

Resetting PMCCNTR_EL0 does not clear the PMCCNTR_EL0 overflow bit to 0.

**P, bit [1]**

Event counter reset. This bit is WO. The effects of writing to this bit are:

0 No action.

1 Reset all event counters, not including PMCCNTR_EL0, to zero.

This bit is always RAZ.

Resetting the event counters does not clear any overflow bits to 0.

**E, bit [0]**

Enable. The possible values of this bit are:

0 All counters, including PMCCNTR_EL0, are disabled.

1 All counters are enabled by PMCNTENSET_EL0.

This bit is RW.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMCR_EL0:

PMCR_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xE04 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.14 PMDEVAFF0, Performance Monitors Device Affinity register 0

The PMDEVAFF0 characteristics are:

**Purpose**

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMDEVAFF0 is in the Debug power domain.

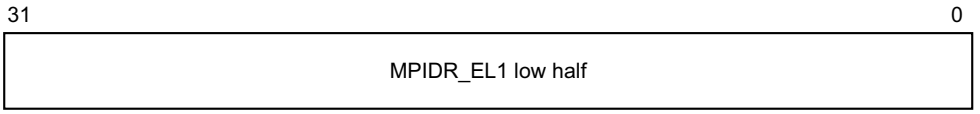
PMDEVAFF0 is optional to implement in the external register interface.

**Attributes**

PMDEVAFF0 is a 32-bit register.

#### Field descriptions

The PMDEVAFF0 bit assignments are:




**Bits [31:0]**

MPIDR_EL1 low half. Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.

#### Accessing the PMDEVAFF0:

PMDEVAFF0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xFA8 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.15 PMDEVAFF1, Performance Monitors Device Affinity register 1

The PMDEVAFF1 characteristics are:

**Purpose**

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMDEVAFF1 is in the Debug power domain.

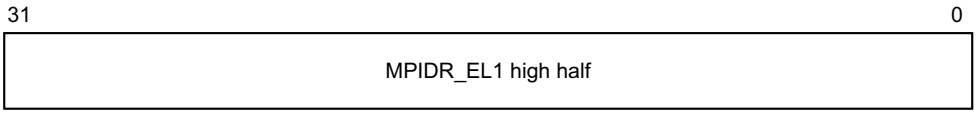
PMDEVAFF1 is optional to implement in the external register interface.

**Attributes**

PMDEVAFF1 is a 32-bit register.

#### Field descriptions

The PMDEVAFF1 bit assignments are:




**Bits [31:0]**

MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.

#### Accessing the PMDEVAFF1:

PMDEVAFF1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xFAC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.16 PMDEVARCH, Performance Monitors Device Architecture register

The PMDEVARCH characteristics are:

**Purpose**

Identifies the programmers' model architecture of the Performance Monitor component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMDEVARCH is in the Debug power domain.

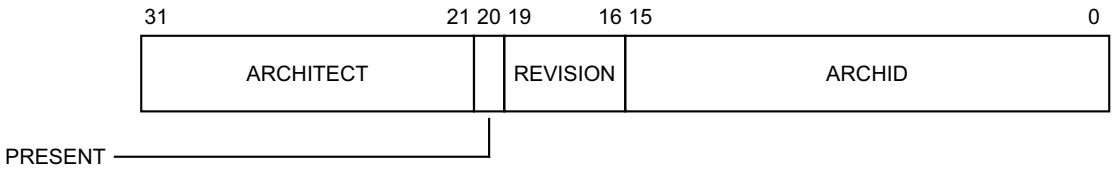
PMDEVARCH is optional to implement in the external register interface.

**Attributes**

PMDEVARCH is a 32-bit register.

#### Field descriptions

The PMDEVARCH bit assignments are:




**ARCHITECT, bits [31:21]**

Defines the architecture of the component. For Performance Monitors, this is ARM Limited.

Bits [31:28] are the JEP 106 continuation code, `0x4`.

Bits [27:21] are the JEP 106 ID code, `0x3B`.

**PRESENT, bit [20]**

When set to 1, indicates that the DEVARCH is present.

This field is 1 in ARMv8.

**REVISION, bits [19:16]**

Defines the architecture revision. For architectures defined by ARM this is the minor revision.

For Performance Monitors, the revision defined by ARMv8 is `0x0`.

All other values are reserved.

**ARCHID, bits [15:0]**

Defines this part to be an ARMv8 debug component. For architectures defined by ARM this is further subdivided.

For Performance Monitors:

- Bits [15:12] are the architecture version, `0x2`.
- Bits [11:0] are the architecture part number, `0xA16`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

This corresponds to Performance Monitors architecture version PMUv3.

**Note**

The PMUv3 memory-mapped programmers' model can be used by devices other than ARMv8 processors. Software must determine whether the PMU is attached to an ARMv8 processor by using the PMDEVAFF0 and PMDEVAFF1 registers to discover the affinity of the PMU to any ARMv8 processors.

### Accessing the PMDEVARCH:

PMDEVARCH can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xFBC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.17 PMDEVTYPE, Performance Monitors Device Type register

The PMDEVTYPE characteristics are:

**Purpose**

Indicates to a debugger that this component is part of a PEs performance monitor interface.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMDEVTYPE is in the Debug power domain.

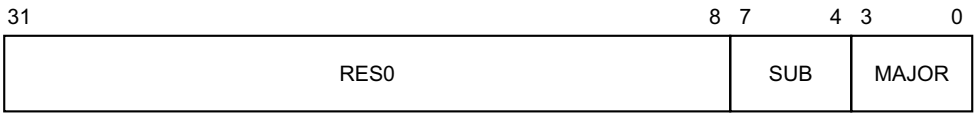
PMDEVTYPE is optional to implement in the external register interface.

**Attributes**

PMDEVTYPE is a 32-bit register.

#### Field descriptions

The PMDEVTYPE bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**SUB, bits [7:4]**

Subtype. Must read as `0x1` to indicate this is a component within a PE.

**MAJOR, bits [3:0]**

Major type. Must read as `0x6` to indicate this is a performance monitor component.

#### Accessing the PMDEVTYPE:

PMDEVTYPE can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xFCC` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.18 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

**Purpose**

Holds event counter n, which counts events, where n is 0 to 30.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

External accesses to the performance monitors ignore PMUSERENR_EL0 and, if implemented, MDCR_EL2.{TPM, TPMCR, HPMN} and MDCR_EL3.TPM. This means that all counters are accessible regardless of the current EL or privilege of the access.

**Configurations**

External register PMEVCNTR<n>_EL0 is architecturally mapped to AArch64 System register PMEVCNTR<n>_EL0.

External register PMEVCNTR<n>_EL0 is architecturally mapped to AArch32 System register PMEVCNTR<n>.

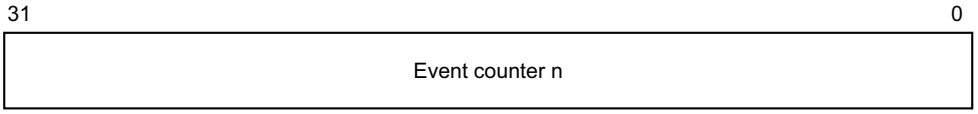
PMEVCNTR<n>_EL0 is in the Core power domain.

**Attributes**

PMEVCNTR<n>_EL0 is a 32-bit register.

#### Field descriptions

The PMEVCNTR<n>_EL0 bit assignments are:




**Bits [31:0]**

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

#### Accessing the PMEVCNTR<n>_EL0:

PMEVCNTR<n>_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0x000` + 8n |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.19 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

**Purpose**

Configures event counter n, where n is 0 to 30.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register PMEVTYPER<n>_EL0 is architecturally mapped to AArch64 System register PMEVTYPER<n>_EL0.

External register PMEVTYPER<n>_EL0 is architecturally mapped to AArch32 System register PMEVTYPER<n>.

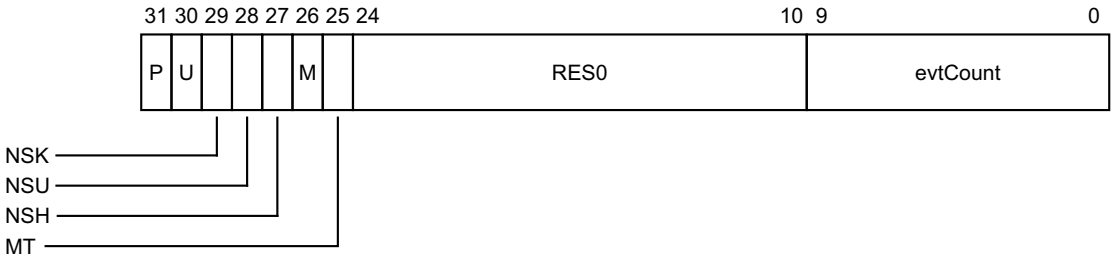
PMEVTYPER<n>_EL0 is in the Core power domain.

**Attributes**

PMEVTYPER<n>_EL0 is a 32-bit register.

## Field descriptions

The PMEVTYPER<n>_EL0 bit assignments are:




**P, bit [31]**

EL1 modes filtering bit. Controls counting in EL1. If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the NSK bit. The possible values of this bit are:

`0` Count events in EL1.

`1` Do not count events in EL1.

**U, bit [30]**

EL0 filtering bit. Controls counting in EL0. If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the NSU bit. The possible values of this bit are:

`0` Count events in EL0.

`1` Do not count events in EL0.

**NSK, bit [29]**

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1. If EL3 is not implemented, this bit is RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the value of this bit is equal to the value of P, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.

**NSU, bit [28]**

Non-secure User mode filtering bit. Controls counting in Non-secure EL0. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of U, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.

**NSH, bit [27]**

Non-secure Hyp modes filtering bit. Controls counting in Non-secure EL2. If EL2 is not implemented, this bit is RES0.

`0` Do not count events in EL2.

`1` Count events in EL2.

**M, bit [26]**

Secure EL3 filtering bit. Most applications can ignore this bit and set the value to zero. If EL3 is not implemented, this bit is RES0.

If the value of this bit is equal to the value of P, events in Secure EL3 are counted.

Otherwise, events in Secure EL3 are not counted.

**MT, bit [25]**

Multi-threading. If MPIDR_EL1.MT is set to 0, this bit is RES0. Otherwise valid values for this bit are:

`0` Count events only on controlling PE.

`1` Count events from any PE with the same affinity at level 1 and above as this PE.

**Note**

Events from a different thread of a multi-threaded implementation are not Attributable to the thread counting the event.

**Bits [24:10]**

Reserved, RES0.

**evtCount, bits [9:0]**

Event to count. The event number of the event that is counted by event counter PMEVCNTR<n>_EL0.

Software must program this field with an event defined by the processor or a common event defined by the architecture.

If evtCount is programmed to an event that is reserved or not implemented, the behavior depends on the event type.

For common architectural and microarchitectural events:

- No events are counted.
- The value read back on evtCount is the value written.

For IMPLEMENTATION DEFINED events:

- It is UNPREDICTABLE what event, if any, is counted. UNPREDICTABLE in this case means the event must not expose privileged information.
- The value read back on evtCount is an UNKNOWN value with the same effect.

ARM recommends that the behavior across a family of implementations is defined such that if a given implementation does not include an event from a set of common IMPLEMENTATION DEFINED events, then no event is counted and the value read back on evtCount is the value written.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMEVTYPER<n>_EL0:

PMEVTYPER<n>_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0x400 + 4n |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.20 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR_EL1 characteristics are:

**Purpose**

Disables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<n>_EL0. Reading the register shows which overflow interrupt requests are enabled.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register PMINTENCLR_EL1 is architecturally mapped to AArch64 System register PMINTENCLR_EL1.

External register PMINTENCLR_EL1 is architecturally mapped to AArch32 System register PMINTENCLR.

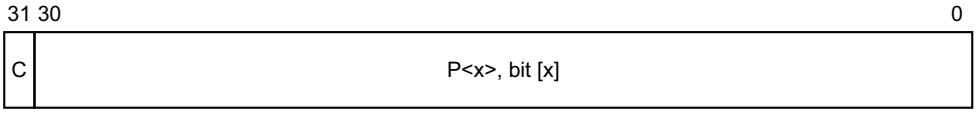
PMINTENCLR_EL1 is in the Core power domain.

**Attributes**

PMINTENCLR_EL1 is a 32-bit register.

## Field descriptions

The PMINTENCLR_EL1 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 overflow interrupt request disable bit. Possible values are:

`0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`1` When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow interrupt request disable bit for PMEVCNTR<x>_EL0.

N is the value in PMCR_EL0.N. Bits [30:N] are RAZ/WI.

Possible values are:

`0` When read, means that the PMEVCNTR<x>_EL0 event counter interrupt request is disabled. When written, has no effect.

`1` When read, means that the PMEVCNTR<x>_EL0 event counter interrupt request is enabled. When written, disables the PMEVCNTR<x>_EL0 interrupt request.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMINTENCLR_EL1:

PMINTENCLR_EL1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xC60 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.21 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register

The PMINTENSET_EL1 characteristics are:

**Purpose**

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<n>_EL0. Reading the register shows which overflow interrupt requests are enabled.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register PMINTENSET_EL1 is architecturally mapped to AArch64 System register PMINTENSET_EL1.

External register PMINTENSET_EL1 is architecturally mapped to AArch32 System register PMINTENSET.

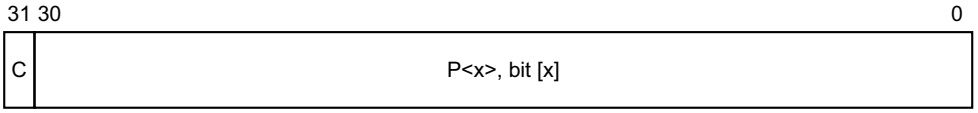
PMINTENSET_EL1 is in the Core power domain.

**Attributes**

PMINTENSET_EL1 is a 32-bit register.

## Field descriptions

The PMINTENSET_EL1 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 overflow interrupt request enable bit. Possible values are:

`0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

`1` When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow interrupt request enable bit for PMEVCNTR<x>_EL0.

N is the value in PMCR_EL0.N. Bits [30:N] are RAZ/WI.

Possible values are:

`0` When read, means that the PMEVCNTR<x>_EL0 event counter interrupt request is disabled. When written, has no effect.

`1` When read, means that the PMEVCNTR<x>_EL0 event counter interrupt request is enabled. When written, enables the PMEVCNTR<x>_EL0 interrupt request.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMINTENSET_EL1:

PMINTENSET_EL1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xC40 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.22 PMITCTRL, Performance Monitors Integration mode Control register

The PMITCTRL characteristics are:

**Purpose**

Enables the Performance Monitors to switch from default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | Default |
|---|---|---|---|
| IMP DEF | IMP DEF | IMP DEF | RW |

**Configurations**

It is IMPLEMENTATION DEFINED whether PMITCTRL is in the Core power domain or in the Debug power domain.

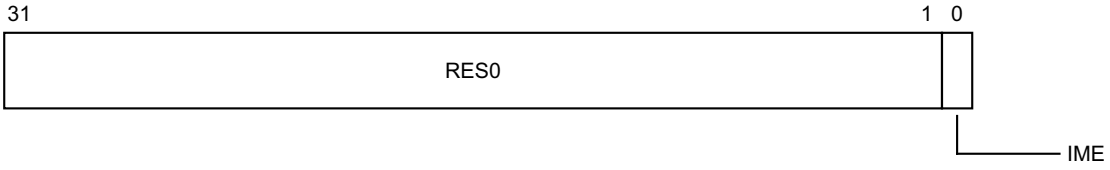
PMITCTRL is optional to implement in the external register interface.

**Attributes**

PMITCTRL is a 32-bit register.

## Field descriptions

The PMITCTRL bit assignments are:




**Bits [31:1]**

Reserved, RES0.

**IME, bit [0]**

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection. The integration mode behavior is IMPLEMENTATION DEFINED.

`0` Normal operation.

`1` Integration mode enabled.

When this register has an architecturally-defined reset value, this field resets to `0`.

This field resets to its defined reset value on IMPLEMENTATION DEFINED reset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMITCTRL:

PMITCTRL can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xF00 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.23 PMLAR, Performance Monitors Lock Access Register

The PMLAR characteristics are:

**Purpose**

Allows or disallows access to the Performance Monitors registers through a memory-mapped interface.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| WO |

**Configurations**

PMLAR is in the Debug power domain.

If OPTIONAL memory-mapped access to the external debug interface is supported then an OPTIONAL Software lock can be implemented as part of CoreSight compliance.

PMLAR ignores writes if the Software lock is not implemented and ignores writes for other accesses to the external debug interface.

The Software lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

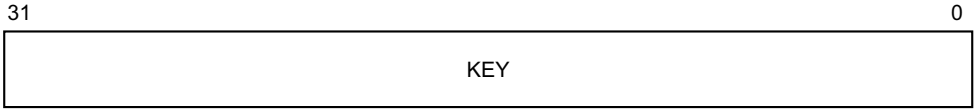
Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

**Attributes**

PMLAR is a 32-bit register.

#### Field descriptions

The PMLAR bit assignments are:




**KEY, bits [31:0]**

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

#### Accessing the PMLAR:

PMLAR can be accessed through the memory-mapped interface:

| Component | Offset |
|---|---|
| PMU | `0xFB0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.24 PMLSR, Performance Monitors Lock Status Register

The PMLSR characteristics are:

**Purpose**

Indicates the current status of the software lock for Performance Monitors registers.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

PMLSR is in the Debug power domain.

If OPTIONAL memory-mapped access to the external debug interface is supported then an OPTIONAL Software lock can be implemented as part of CoreSight compliance.

PMLSR is RAZ if the Software lock is not implemented and is RAZ for other accesses to the external debug interface.

The Software lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

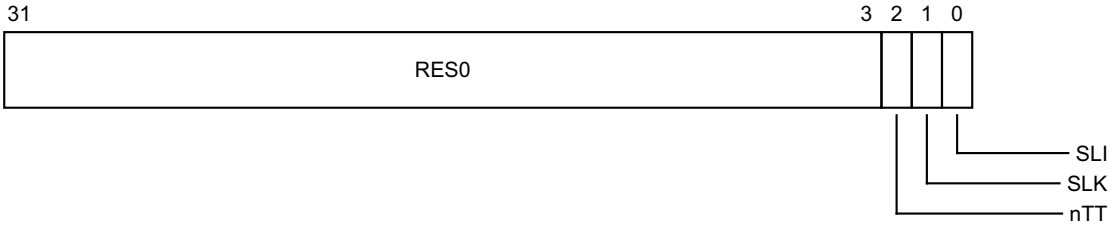
Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

**Attributes**

PMLSR is a 32-bit register.

#### Field descriptions

The PMLSR bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**nTT, bit [2]**

Not thirty-two bit access required. RAZ.

**SLK, bit [1]**

Software lock status for this component. For an access to LSR that is not a memory-mapped access, or when the software lock is not implemented, this field is RES0.

For memory-mapped accesses when the software lock is implemented, possible values of this field are:

`0` Lock clear. Writes are permitted to this component's registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 Lock set. Writes to this component's registers are ignored, and reads have no side effects.

When this register has an architecturally-defined reset value, this field resets to 1.

This field resets to its defined reset value on External debug reset.

**SLI, bit [0]**

Software lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ. For memory-mapped accesses, the value of this field is IMPLEMENTATION DEFINED. Permitted values are:

0 Software lock not implemented or not memory-mapped access.

1 Software lock implemented and memory-mapped access.

### Accessing the PMLSR:

PMLSR can be accessed through the memory-mapped interface:

| Component | Offset |
|---|---|
| PMU | 0xFB4 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.25 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear register

The PMOVSCLR_EL0 characteristics are:

**Purpose**

Contains the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<x>. Writing to this register clears these bits.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register PMOVSCLR_EL0 is architecturally mapped to AArch64 System register PMOVSCLR_EL0.

External register PMOVSCLR_EL0 is architecturally mapped to AArch32 System register PMOVSR.

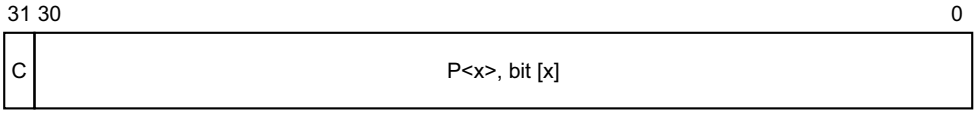
PMOVSCLR_EL0 is in the Core power domain.

**Attributes**

PMOVSCLR_EL0 is a 32-bit register.

## Field descriptions

The PMOVSCLR_EL0 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 overflow bit. Possible values are:

`0` When read, means the cycle counter has not overflowed. When written, has no effect.

`1` When read, means the cycle counter has overflowed. When written, clears the overflow bit to 0.

PMCR_EL0.LC is used to control from which bit of PMCCNTR_EL0 (bit 31 or bit 63) an overflow is detected.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow clear bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI. N is the value in PMCR_EL0.N.

Possible values of each bit are:

`0` When read, means that PMEVCNTR<x> has not overflowed. When written, has no effect.

`1` When read, means that PMEVCNTR<x> has overflowed. When written, clears the PMEVCNTR<x> overflow bit to 0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMOVSCLR_EL0:

PMOVSCLR_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xC80 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.26 PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register

The PMOVSSET_EL0 characteristics are:

**Purpose**

Sets the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<x>.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | RO | RW |

**Configurations**

External register PMOVSSET_EL0 is architecturally mapped to AArch64 System register PMOVSSET_EL0.

External register PMOVSSET_EL0 is architecturally mapped to AArch32 System register PMOVSSET.

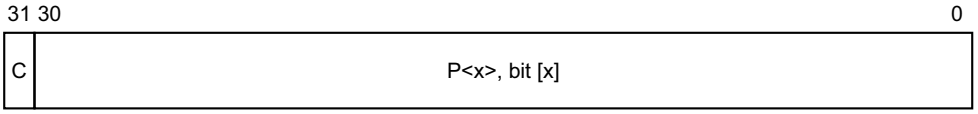
PMOVSSET_EL0 is in the Core power domain.

**Attributes**

PMOVSSET_EL0 is a 32-bit register.

## Field descriptions

The PMOVSSET_EL0 bit assignments are:




**C, bit [31]**

PMCCNTR_EL0 overflow bit. Possible values are:

`0` When read, means the cycle counter has not overflowed. When written, has no effect.

`1` When read, means the cycle counter has overflowed. When written, sets the overflow bit to 1.

**P<x>, bit [x], for x = 0 to 30**

Event counter overflow set bit for PMEVCNTR<x>.

Bits [30:N] are RAZ/WI. N is the value in PMCR_EL0.N.

Possible values are:

`0` When read, means that PMEVCNTR<x> has not overflowed. When written, has no effect.

`1` When read, means that PMEVCNTR<x> has overflowed. When written, sets the PMEVCNTR<x> overflow bit to 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMOVSSET_EL0:

PMOVSSET_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xCC0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.27 PMPIDR0, Performance Monitors Peripheral Identification Register 0

The PMPIDR0 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMPIDR0 is in the Debug power domain.

PMPIDR0 is optional to implement in the external register interface.

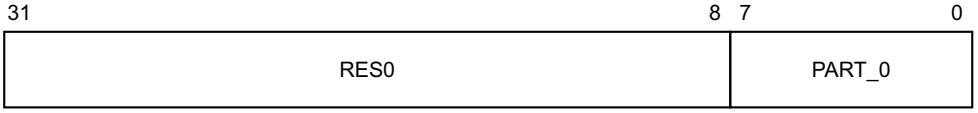
This register is required for CoreSight compliance.

**Attributes**

PMPIDR0 is a 32-bit register.

#### Field descriptions

The PMPIDR0 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PART_0, bits [7:0]**

Part number, least significant byte.

#### Accessing the PMPIDR0:

PMPIDR0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xFE0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.28 PMPIDR1, Performance Monitors Peripheral Identification Register 1

The PMPIDR1 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMPIDR1 is in the Debug power domain.

PMPIDR1 is optional to implement in the external register interface.

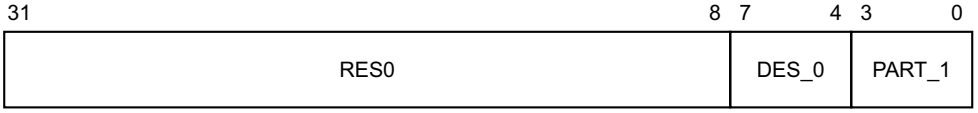
This register is required for CoreSight compliance.

**Attributes**

PMPIDR1 is a 32-bit register.

#### Field descriptions

The PMPIDR1 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**DES_0, bits [7:4]**

Designer, least significant nibble of JEP106 ID code. For ARM Limited, this field is `0b1011`.

**PART_1, bits [3:0]**

Part number, most significant nibble.

#### Accessing the PMPIDR1:

PMPIDR1 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xFE4` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.29 PMPIDR2, Performance Monitors Peripheral Identification Register 2

The PMPIDR2 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMPIDR2 is in the Debug power domain.

PMPIDR2 is optional to implement in the external register interface.

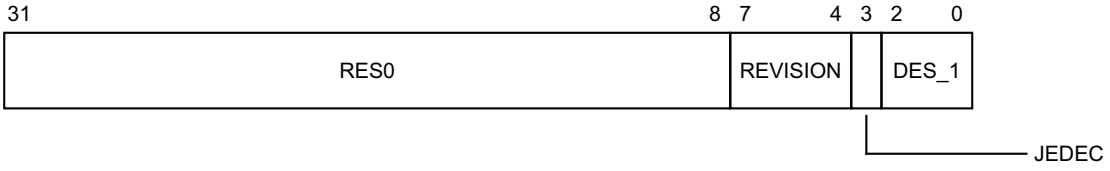
This register is required for CoreSight compliance.

**Attributes**

PMPIDR2 is a 32-bit register.

#### Field descriptions

The PMPIDR2 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**REVISION, bits [7:4]**

Part major revision. Parts can also use this field to extend Part number to 16-bits.

**JEDEC, bit [3]**

RAO. Indicates a JEP106 identity code is used.

**DES_1, bits [2:0]**

Designer, most significant bits of JEP106 ID code. For ARM Limited, this field is `0b011`.

#### Accessing the PMPIDR2:

PMPIDR2 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xFE8` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.30 PMPIDR3, Performance Monitors Peripheral Identification Register 3

The PMPIDR3 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMPIDR3 is in the Debug power domain.

PMPIDR3 is optional to implement in the external register interface.

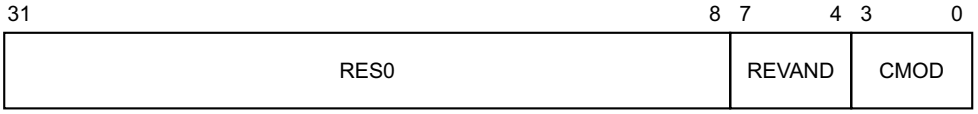
This register is required for CoreSight compliance.

**Attributes**

PMPIDR3 is a 32-bit register.

#### Field descriptions

The PMPIDR3 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**REVAND, bits [7:4]**

Part minor revision. Parts using PMPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

**CMOD, bits [3:0]**

Customer modified. Indicates someone other than the Designer has modified the component.

#### Accessing the PMPIDR3:

PMPIDR3 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xFEC |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.31 PMPIDR4, Performance Monitors Peripheral Identification Register 4

The PMPIDR4 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

**Usage constraints**

This register is accessible as follows:

| SLK | Default |
|---|---|
| RO | RO |

**Configurations**

PMPIDR4 is in the Debug power domain.

PMPIDR4 is optional to implement in the external register interface.

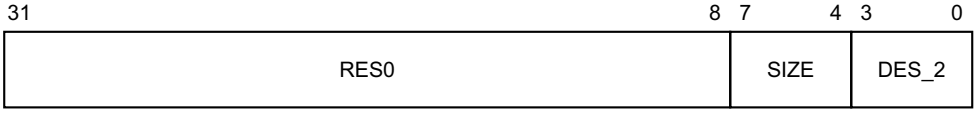
This register is required for CoreSight compliance.

**Attributes**

PMPIDR4 is a 32-bit register.

#### Field descriptions

The PMPIDR4 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**SIZE, bits [7:4]**

Size of the component. RAZ. $Log_2$ of the number of 4KB pages from the start of the component to the end of the component ID registers.

**DES_2, bits [3:0]**

Designer, JEP106 continuation code, least significant nibble. For ARM Limited, this field is `0b0100`.

#### Accessing the PMPIDR4:

PMPIDR4 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | `0xFD0` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.3.32 PMSWINC_EL0, Performance Monitors Software Increment register

The PMSWINC_EL0 characteristics are:

**Purpose**

Increments a counter that is configured to count the Software increment event, event `0x00`.

**Usage constraints**

This register is accessible as follows:

| Off | DLK | OSLK | EPMAD | SLK | Default |
|---|---|---|---|---|---|
| Error | Error | Error | Error | WI | WO |

**Configurations**

External register PMSWINC_EL0 is architecturally mapped to AArch64 System register PMSWINC_EL0.

External register PMSWINC_EL0 is architecturally mapped to AArch32 System register PMSWINC.

PMSWINC_EL0 is in the Core power domain.

PMSWINC_EL0 is optional to implement in the external register interface.

If this register is implemented, use of it is deprecated.

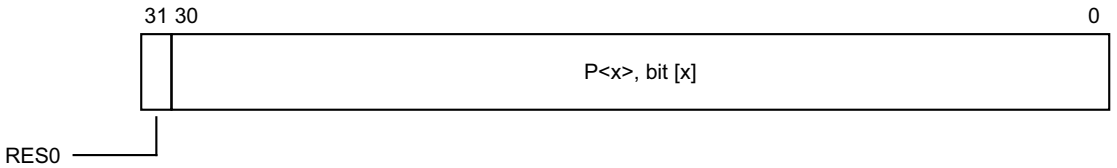
If 1 is written to bit [x] from the external debug interface, it is CONSTRAINED UNPREDICTABLE whether or not a SW_INCR event is created for counter x. This is consistent with not implementing the register in the external debug interface.

**Attributes**

PMSWINC_EL0 is a 32-bit register.

## Field descriptions

The PMSWINC_EL0 bit assignments are:




**Bit [31]**

Reserved, RES0.

**P<x>, bit [x], for x = 0 to 30**

Event counter software increment bit for PMEVCNTR<x>.

P<x> is WI if x >= PMCR_EL0.N, the number of implemented counters.

Otherwise, the effects of writing to this bit are:

`0` No action. The write to this bit is ignored.

`1` It is CONSTRAINED UNPREDICTABLE whether a SW_INCR event is generated for event counter x.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the PMSWINC_EL0:

PMSWINC_EL0 can be accessed through the memory-mapped interface and the external debug interface:

| Component | Offset |
|---|---|
| PMU | 0xCA0 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I3.4 Generic Timer memory-mapped registers overview

The Generic Timer memory-mapped registers are implemented as multiple register frames, with each register frame having its own base address, as follows:

- A single CNTCTLBase register frame, at base address CNTCTLBase.
- Between one and seven CNTBase*N* register frames, each with its own base address CNTBase*N*.
- For each CNTBase*N* register frame, if required, a CNTEL0Base*N* register frame, at base address CNTEL0Base*N*, that provides an EL0 view of the CNTBase*N* register frame.

For more information, see:

- *Memory-mapped timer components* on page I1-5104.
- *The CNTBaseN and CNTEL0BaseN frames* on page I1-5105. This section includes the memory map of the CNTBase*N* and CNTBase*N* register frames.
- *The CNTCTLBase frame* on page I1-5107. This section includes the memory map of the CNTCTLBase register frame.
- *Providing a complete set of counter and timer features* on page I1-5108.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## I3.5 Generic Timer memory-mapped register descriptions

This section describes the Generic Timer registers. *Generic Timer memory-mapped registers overview* on page I3-5172 gives an overview of these registers, and includes links to their memory maps.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.1 CNTACR<n>, Counter-timer Access Control Registers, n = 0 - 7

The CNTACR<n> characteristics are:

**Purpose**

Provides top-level access controls for the elements of a timer frame. CNTACR<n> provides the controls for frame CNTBaseN.

In addition to the CNTACR<n> control:

- CNTNSAR controls whether CNTACR<n> is accessible from Non-secure state.
- If frame CNTEL0BaseN is implemented, the CNTEL0ACR in frame CNTBaseN provides additional control of accesses to frame CNTEL0BaseN.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RW |

In a system that implements both Secure and Non-secure states:

- CNTACR<n> is always accessible in Secure state.
- CNTNSAR.NS<n> determines whether CNTACR<n> is accessible in Non-secure state.

**Configurations**

Implemented only if the value of CNTTIDR.Frame<n> is 1.

An implementation of the counters might not provide configurable access to some or all of the features. In this case, the associated field in the CNTACR<n> register is:

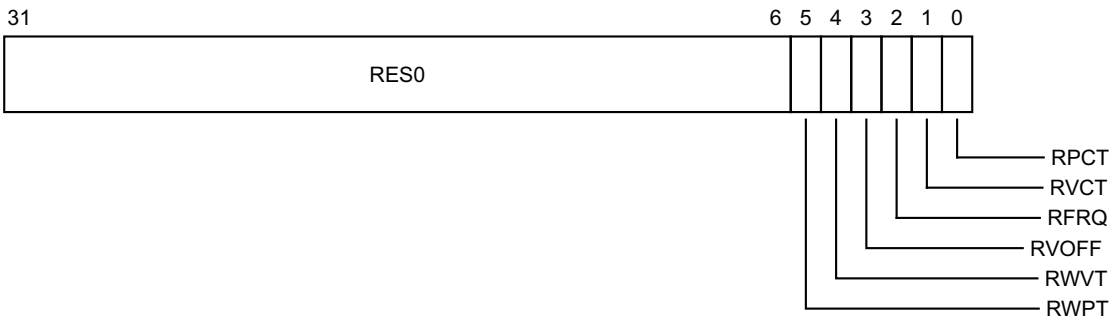
- RAZ/WI if access is always denied.
- RAO/WI if access is always permitted.

**Attributes**

CNTACR<n> is a 32-bit register.

#### Field descriptions

The CNTACR<n> bit assignments are:




**Bits [31:6]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**RWPT, bit [5]**

Read/write access to the EL1 Physical Timer registers CNTP_CVAL, CNTP_TVAL, and CNTP_CTL, in frame <n>. The possible values of this bit are:

`0` No access to the EL1 Physical Timer registers in frame <n>. The registers are RES0.

`1` Read/write access to the EL1 Physical Timer registers in frame <n>.

**RWVT, bit [4]**

Read/write access to the Virtual Timer register CNTV_CVAL, CNTV_TVAL, and CNTV_CTL, in frame <n>. The possible values of this bit are:

`0` No access to the Virtual Timer registers in frame <n>. The registers are RES0.

`1` Read/write access to the Virtual Timer registers in frame <n>.

**RVOFF, bit [3]**

Read-only access to CNTVOFF, in frame <n>. The possible values of this bit are:

`0` No access to CNTVOFF in frame <n>. The register is RES0.

`1` Read-only access to CNTVOFF in frame <n>.

**RFRQ, bit [2]**

Read-only access to CNTFRQ, in frame <n>. The possible values of this bit are:

`0` No access to CNTFRQ in frame <n>. The register is RES0.

`1` Read-only access to CNTFRQ in frame <n>.

**RVCT, bit [1]**

Read-only access to CNTVCT, in frame <n>. The possible values of this bit are:

`0` No access to CNTVCT in frame <n>. The register is RES0.

`1` Read-only access to CNTVCT in frame <n>.

**RPCT, bit [0]**

Read-only access to CNTPCT, in frame <n>. The possible values of this bit are:

`0` No access to CNTPCT in frame <n>. The register is RES0.

`1` Read-only access to CNTPCT in frame <n>.

### Accessing the CNTACR<n>:

CNTACR<n> can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTCTLBase | `0x040` + 4n |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.2 CNTCR, Counter Control Register

The CNTCR characteristics are:

**Purpose**

Enables the counter, controls the counter frequency setting, and controls counter behavior during debug.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RW |

In a system that implements both Secure and Non-secure states, this register is only writable in Secure state.

**Configurations**

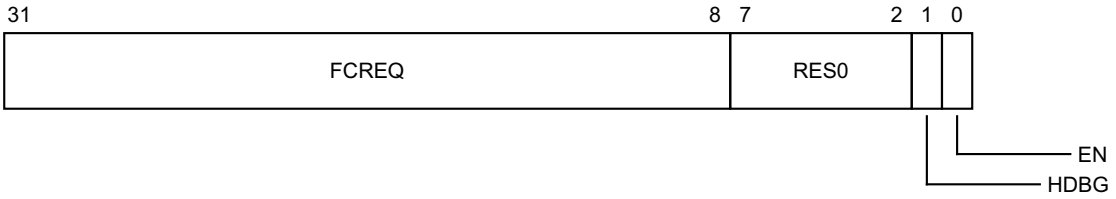
There are no configuration notes.

**Attributes**

CNTCR is a 32-bit register.

## Field descriptions

The CNTCR bit assignments are:




**FCREQ, bits [31:8]**

Frequency change request. Indicates the number of the entry in the frequency table to select.

Selecting an unimplemented entry, or an entry that contains 0, has no effect on the counter.

When this register has an architecturally-defined reset value, this field resets to `0`.

**Bits [7:2]**

Reserved, RES0.

**HDBG, bit [1]**

Halt-on-debug. Controls whether a Halt-on-debug signal halts the system counter:

`0` System counter ignores Halt-on-debug.

`1` Asserted Halt-on-debug signal halts system counter update.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**EN, bit [0]**

Enables the counter:

`0` System counter disabled.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

1 System counter enabled.

When this register has an architecturally-defined reset value, this field resets to `0`.

### Accessing the CNTCR:

CNTCR can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTControlBase | `0x000` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.3 CNTCV, Counter Count Value register

The CNTCV characteristics are:

**Purpose**

Indicates the current count value.

**Usage constraints**

This register is accessible as follows:

| Frame | Accessibility |
|---|---|
| CNTControlBase | RW |
| CNTReadBase | RO |

A write to CNTCV must be visible in the CNTPCT register of each running processor in a finite time.

For the writable copy of the register:

- If the counter is enabled, the effect if writing the register is UNPREDICTABLE.
- If the system implements both Secure and Non-secure states, the register is writable only by Secure writes.

In an implementation that supports 64-bit atomic memory accesses, this register must be accessible using a 64-bit atomic access.

**Configurations**

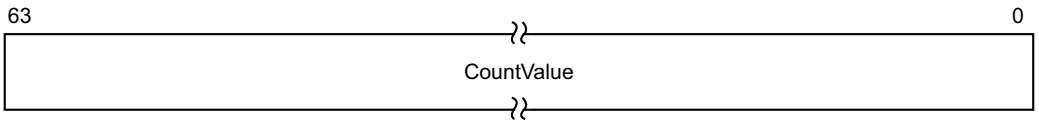
There are no configuration notes.

**Attributes**

CNTCV is a 64-bit register.

#### Field descriptions

The CNTCV bit assignments are:




**CountValue, bits [63:0]**

Indicates the counter value.

#### Accessing the CNTCV:

CNTCV[31:0] can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTControlBase | 0x008 |
| Timer | CNTReadBase | 0x000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

CNTCV[63:32] can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTControlBase | 0x00C |
| Timer | CNTReadBase | 0x004 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.4 CNTEL0ACR, Counter-timer EL0 Access Control Register

The CNTEL0ACR characteristics are:

**Purpose**

An implementation of CNTEL0ACR in the frame at CNTBaseN controls whether the CNTPCT, CNTVCT, CNTFRQ, EL1 Physical Timer, and Virtual Timer registers are visible in the frame at CNTEL0BaseN.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RW |

If CNTEL0ACR is implemented, it is implemented as a RW register in the CNTBaseN frame when the value of bit[0] of CNTTIDR.Frame<n> is 1, otherwise the encoding is RES0.

If CNTEL0ACR is not implemented:

- The register location is RAZ/WI.
- The registers CNTFRQ, CNTP_CTL, CNTP_CVAL, CNTP_TVAL, CNTPCT, CNTV_CTL, CNTV_CVAL, CNTV_TVAL, and CNTVCT are not visible in the corresponding CNTEL0BaseN frame.

**Configurations**

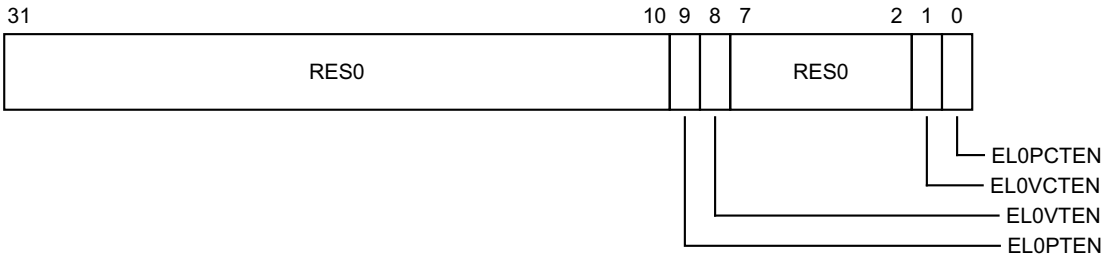
In each implemented CNTBaseN frame, CNTEL0ACR is optional.

**Attributes**

CNTEL0ACR is a 32-bit register.

#### Field descriptions

The CNTEL0ACR bit assignments are:




**Bits [31:10]**

Reserved, RES0.

**EL0PTEN, bit [9]**

Second view read/write access control for the EL1 Physical Timer registers. This bit controls whether the CNTP_CVAL, CNTP_TVAL, and CNTP_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame. The possible values of this bit are:

`0` No access. Registers are RES0 in the second view.

`1` Access permitted. If the registers are accessible in the current frame then they are accessible in the second view.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**EL0VTEN, bit [8]**

Second view read/write access control for the Virtual Timer registers. This bit controls whether the CNTV_CVAL, CNTV_TVAL, and CNTV_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame. The possible values of this bit are:

`0` No access. Registers are RES0 in the second view.

`1` Access permitted. If the registers are accessible in the current frame then they are accessible in the second view.

The definition of this bit means that, if the Virtual Timer registers are not implemented in the current CNTBaseN frame, then the Virtual Timer register addresses are RES0 in the corresponding CNTEL0BaseN frame, regardless of the value of this bit.

**Bits [7:2]**

Reserved, RES0.

**EL0VCTEN, bit [1]**

Second view read access control for CNTVCT and CNTFRQ. The possible values of this bit are:

`0` CNTVCT is not visible in the second view.
If EL0PCTEN is set to 0, CNTFRQ is not visible in the second view.

`1` Access permitted. If CNTVCT and CNTFRQ are visible in the current frame then they are visible in the second view.

**EL0PCTEN, bit [0]**

Second view read access control for CNTPCT and CNTFRQ. The possible values of this bit are:

`0` CNTPCT is not visible in the second view.
If EL0VCTEN is set to 0, CNTFRQ is not visible in the second view.

`1` Access permitted. If CNTPCT and CNTFRQ are visible in the current frame then they are visible in the second view.

## Accessing the CNTEL0ACR:

CNTEL0ACR can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | `0x014` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.5 CNTFID0, Counter Frequency ID

The CNTFID0 characteristics are:

**Purpose**

Indicates the base frequency of the system counter.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO or RW |

**Configurations**

The possible frequencies for the system counter are stored as 32-bit words starting with the base frequency, CNTFID0.

A 32-bit word of zero value after the final frequency mode entry marks the end of the frequency modes table.

Typically, the frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

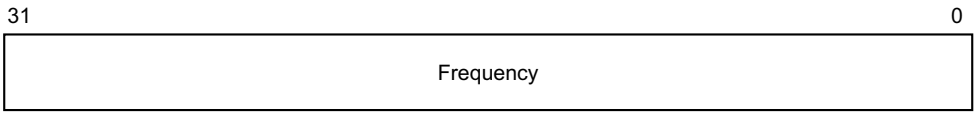
If the frequency modes table is in read/write memory, ARM strongly recommends that the frequency modes table is not updated once the system is running.

**Attributes**

CNTFID0 is a 32-bit register.

#### Field descriptions

The CNTFID0 bit assignments are:




**Frequency, bits [31:0]**

The base frequency of the system counter, in Hz.

#### Accessing the CNTFID0:

CNTFID0 can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTControlBase | 0x020 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.6 CNTFID<n>, Counter Frequency IDs, n = 1 - 23

The CNTFID<n> characteristics are:

**Purpose**

Indicates alternative system counter update frequencies.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO or RW |

**Configurations**

CNTFID<n> is optional to implement in the external register interface.

The possible frequencies for the system counter are stored as 32-bit words starting with the base frequency, CNTFID0.

A 32-bit word of zero value after the final frequency mode entry marks the end of the frequency modes table. The only required entry in the table is the entry for CNTFID0.

Typically, the frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialise the table entries as part of its start-up sequence.

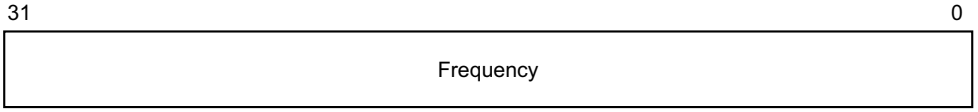
If the frequency modes table is in read/write memory, ARM strongly recommends that the frequency modes table is not updated once the system is running.

**Attributes**

CNTFID<n> is a 32-bit register.

#### Field descriptions

The CNTFID<n> bit assignments are:




**Frequency, bits [31:0]**

A system counter update frequency, in Hz. Must be an exact divisor of the base frequency. ARM strongly recommends that all frequency values in the table are integer power-of-two divisors of the base frequency.

When the system timer is operating at a lower frequency than the base frequency, the increment applied at each counter update is given by:

increment = (base frequency) / selected frequency)

#### Accessing the CNTFID<n>:

CNTFID<n> can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTControlBase | `0x020` + 4n |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.7 CNTFRQ, Counter-timer Frequency

The CNTFRQ characteristics are:

**Purpose**

Holds the clock frequency of the system counter.

**Usage constraints**

This register is accessible as follows:

| Frame | Accessibility |
|---|---|
| CNTCTLBase | RW |
| CNTBaseN | Config-RO |
| CNTEL0BaseN | Config-RO |

For the CNTBaseN and CNTEL0BaseN frames:

- A RO copy of CNTFRQ is implemented in the CNTBaseN frame when both:
  - CNTACR<n>.RFRQ is 1.
  - Bit[0] of CNTTIDR.Frame<n> is 1.

  Otherwise the encoding in CNTBaseN is RES0.
- When CNTFRQ is RO in the CNTBaseN frame, it is also RO in the CNTEL0BaseN frame if bit[2] of CNTTIDR.Frame<n> is 1 and either:
  - The value of CNTEL0ACR.EL0VCTEN is 1.
  - The value of CNTEL0ACR.EL0PCTEN is 1.

  Otherwise the CNTFRQ encoding in CNTEL0BaseN frame is RES0.

In a system that implements both Secure and Non-secure states, this register is only accessible in Secure state.

**Configurations**

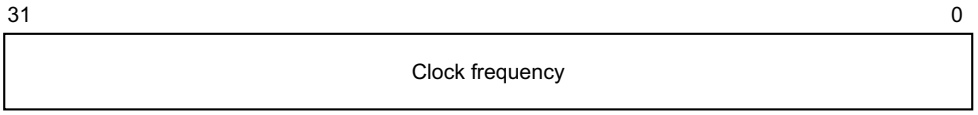
There are no configuration notes.

**Attributes**

CNTFRQ is a 32-bit register.

#### Field descriptions

The CNTFRQ bit assignments are:




**Bits [31:0]**

Clock frequency. Indicates the system counter clock frequency, in Hz.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTFRQ:

CNTFRQ can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x010 |
| Timer | CNTEL0BaseN | 0x010 |
| Timer | CNTCTLBase | 0x000 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.8 CNTNSAR, Counter-timer Non-secure Access Register

The CNTNSAR characteristics are:

**Purpose**

Provides the highest-level control of whether frames CNTBaseN and CNTEL0BaseN are accessible by Non-secure accesses.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RW |

In a system that implements both Secure and Non-secure states, this register is only accessible in Secure state.

**Configurations**

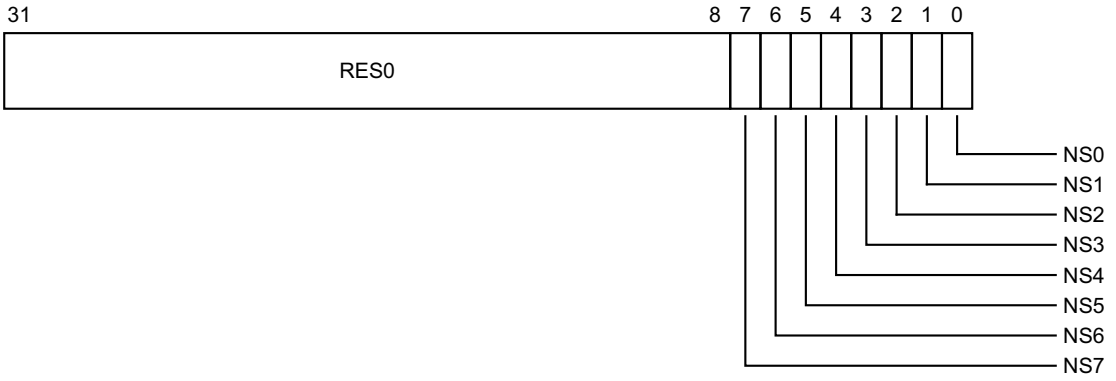
There are no configuration notes.

**Attributes**

CNTNSAR is a 32-bit register.

#### Field descriptions

The CNTNSAR bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**NS<n>, bit [n], for n = 0 to 7**

Non-secure access to frame n. The possible values of this bit are:

`0` Secure access only. Behaves as RES0 to Non-secure accesses.

`1` Secure and Non-secure accesses permitted.

This bit also determines whether, in the CNTCTLBase frame, CNTACR<n> and CNTVOFF are accessible to Non-secure accesses.

If frame CNTBase<n>:

- Is not implemented, then NS<n> is RES0.
- Is not Configurable access, and is accessible only by Secure accesses, then NS<n> is RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Is not Configurable access, and is accessible by both Secure and Non-secure accesses, then NS<n> is RES1.

### Accessing the CNTNSAR:

CNTNSAR can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTCTLBase | 0x004 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.9 CNTP_CTL, Counter-timer Physical Timer Control

The CNTP_CTL characteristics are:

**Purpose**

Control register for the physical timer.

**Usage constraints**

This register is accessible as follows:

| Frame | Accessibility |
|---|---|
| CNTBaseN | Config-RW |
| CNTEL0BaseN | Config-RW |

CNTP_CTL is implemented as a RW register in the CNTBaseN frame when both:

- CNTACR<n>.RWPT is 1.
- Bit[0] of CNTTIDR.Frame<n> is 1.

Otherwise the encoding in the CNTBaseN frame is RES0.

When CNTP_CTL is implemented as a RW register in the CNTBaseN frame, it is also implemented as a RW register in the CNTEL0BaseN frame if both:

- CNTEL0ACR.EL0PTEN is 1.
- Bit[2] of CNTTIDR.Frame<n> is 1.

Otherwise the CNTP_CTL register in the CNTEL0BaseN frame is RES0.

**Configurations**

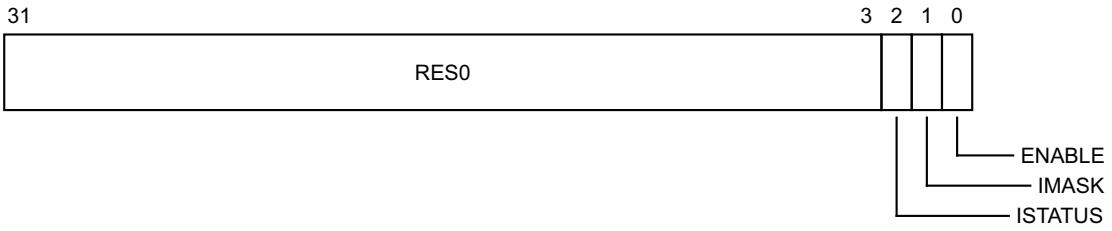
There are no configuration notes.

**Attributes**

CNTP_CTL is a 32-bit register.

#### Field descriptions

The CNTP_CTL bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is asserted:

0 Timer condition is not asserted.

1 Timer condition is asserted.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see *Operation of the CompareValue views of the timers* on page D6-1904 and *Operation of the TimerValue views of the timers* on page D6-1904.

This bit is read-only.

**IMASK, bit [1]**

Timer output signal mask bit. Permitted values are:

`0` Timer output signal is not masked.

`1` Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

`0` Timer disabled.

`1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL continues to count down.

**Note**

Disabling the output signal might be a power-saving option.

## Accessing the CNTP_CTL:

CNTP_CTL can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | `0x02C` |
| Timer | CNTEL0BaseN | `0x02C` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.10 CNTP_CVAL, Counter-timer Physical Timer CompareValue

The CNTP_CVAL characteristics are:

**Purpose**

Holds the 64-bit compare value for the EL1 physical timer.

**Usage constraints**

This register is accessible as follows:

| Frame | Accessibility |
|---|---|
| CNTBaseN | Config-RW |
| CNTEL0BaseN | Config-RW |

CNTP_CVAL is implemented as a RW register in the CNTBaseN frame when both:

- CNTACR<n>.RWPT is 1.
- Bit[0] of CNTTIDR.Frame<n> is 1.

Otherwise the encoding in the CNTBaseN frame is RES0.

When CNTP_CVAL is implemented as a RW register in the CNTBaseN frame, it is also implemented as a RW register in the CNTEL0BaseN frame if both:

- CNTEL0ACR.EL0PTEN is 1.
- Bit[2] of CNTTIDR.Frame<n> is 1.

Otherwise the CNTP_CVAL register in the CNTEL0BaseN frame is RES0.

If the implementation supports 64-bit atomic accesses, then the CNTP_CVAL register must be accessible as an atomic 64-bit value.

**Configurations**

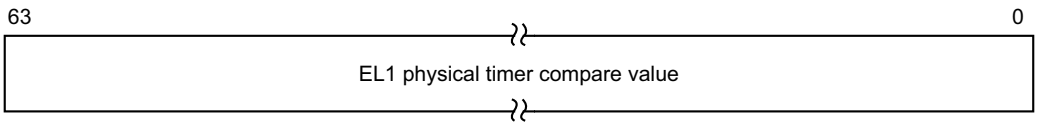
There are no configuration notes.

**Attributes**

CNTP_CVAL is a 64-bit register.

#### Field descriptions

The CNTP_CVAL bit assignments are:




**Bits [63:0]**

EL1 physical timer compare value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTP_CVAL:

CNTP_CVAL[31:0] can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x020 |
| Timer | CNTEL0BaseN | 0x020 |

CNTP_CVAL[63:32] can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x024 |
| Timer | CNTEL0BaseN | 0x024 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.11 CNTP_TVAL, Counter-timer Physical Timer TimerValue

The CNTP_TVAL characteristics are:

**Purpose**

Holds the timer value for the EL1 physical timer. This provides a 32-bit downcounter.

**Usage constraints**

This register is accessible as follows:

| Frame | Accessibility |
|---|---|
| CNTBaseN | Config-RW |
| CNTEL0BaseN | Config-RW |

CNTP_TVAL is implemented as a RW register in the CNTBaseN frame when both:

- CNTACR<n>.RWPT is 1.
- Bit[0] of CNTTIDR.Frame<n> is 1.

Otherwise the encoding in the CNTBaseN frame is RES0.

When CNTP_TVAL is implemented as a RW register in the CNTBaseN frame, it is also implemented as RW in the CNTEL0BaseN frame if both:

- CNTEL0ACR.EL0PTEN is 1.
- Bit[2] of CNTTIDR.Frame<n> is 1.

Otherwise the CNTP_TVAL register in the CNTEL0BaseN frame is RES0.

**Configurations**

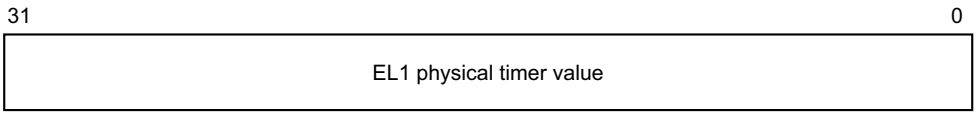
There are no configuration notes.

**Attributes**

CNTP_TVAL is a 32-bit register.

#### Field descriptions

The CNTP_TVAL bit assignments are:




**Bits [31:0]**

EL1 physical timer value.

#### Accessing the CNTP_TVAL:

CNTP_TVAL can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x028 |
| Timer | CNTEL0BaseN | 0x028 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.12 CNTPCT, Counter-timer Physical Count

The CNTPCT characteristics are:

**Purpose**

Holds the 64-bit physical count value.

**Usage constraints**

This register is accessible as follows:

| Frame | Accessibility |
|---|---|
| CNTBaseN | Config-RO |
| CNTEL0BaseN | Config-RO |

CNTPCT is implemented as a RO register in the CNTBaseN frame when both:

- CNTACR<n>.RPCT is 1.
- Bit[0] of CNTTIDR.Frame<n> is 1.

Otherwise the encoding in the CNTBaseN frame is RES0.

When CNTPCT is implemented as a RO register in the CNTBaseN frame, it is also implemented as a RO register in the CNTEL0BaseN frame if both:

- CNTEL0ACR.EL0PCTEN is 1.
- Bit[2] of CNTTIDR.Frame<n> is 1.

Otherwise the CNTPCT register in the CNTEL0BaseN frame is RES0.

If the implementation supports 64-bit atomic accesses, then the CNTPCT register must be accessible as an atomic 64-bit value.

**Configurations**

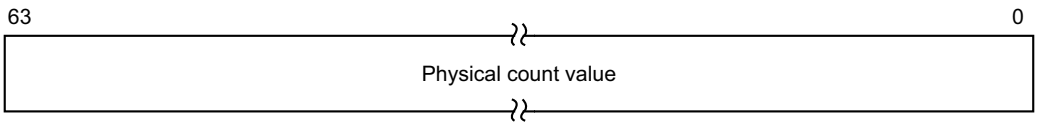
There are no configuration notes.

**Attributes**

CNTPCT is a 64-bit register.

## Field descriptions

The CNTPCT bit assignments are:




**Bits [63:0]**

Physical count value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTPCT:

CNTPCT[31:0] can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x000 |
| Timer | CNTEL0BaseN | 0x000 |

CNTPCT[63:32] can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x004 |
| Timer | CNTEL0BaseN | 0x004 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.13 CNTSR, Counter Status Register

The CNTSR characteristics are:

**Purpose**

Provides counter frequency status information.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

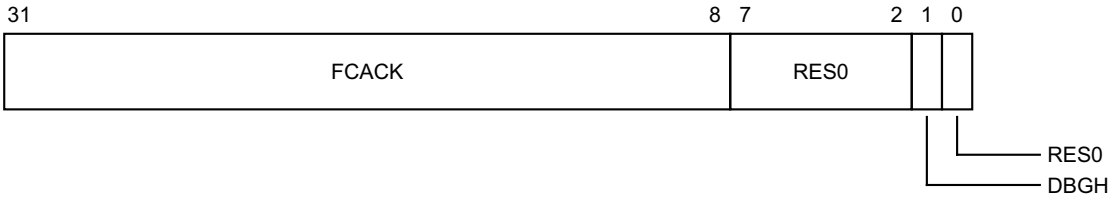
There are no configuration notes.

**Attributes**

CNTSR is a 32-bit register.

#### Field descriptions

The CNTSR bit assignments are:




**FCACK, bits [31:8]**

Frequency change acknowledge. Indicates the currently selected entry in the frequency table.

When this register has an architecturally-defined reset value, this field resets to `0`.

**Bits [7:2]**

Reserved, RES0.

**DBGH, bit [1]**

Indicates whether the counter is halted because the Halt-on-Debug signal is asserted:

`0` Counter is not halted.

`1` Counter is halted.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

**Bit [0]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTSR:

CNTSR can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTControlBase | 0x004 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.14 CNTTIDR, Counter-timer Timer ID Register

The CNTTIDR characteristics are:

**Purpose**

Indicates the implemented timers in the memory map, and their features. For each value of N from 0 to 7 it indicates whether:

- Frame CNTBaseN is a view of an implemented timer.
- Frame CNTBaseN has a second view, CNTEL0BaseN.
- Frame CNTBaseN has a virtual timer capability.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

This register is accessible in both Secure and Non-secure states.

**Configurations**

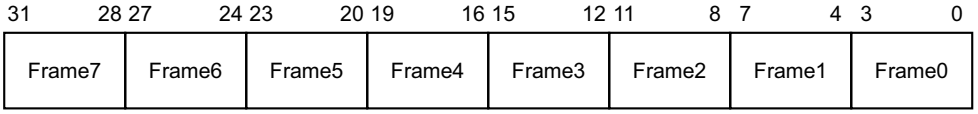
There are no configuration notes.

**Attributes**

CNTTIDR is a 32-bit register.

## Field descriptions

The CNTTIDR bit assignments are:




**Frame<n>, bits [4n+3:4n], for n = 0 to 7**

A 4-bit field indicating the features of frame CNTBase<n>.

Bit[3] of the field is RES0.

Bit[2] indicates whether frame CNTBase<n> has a second view, CNTEL0Base<n>. The possible values of this bit are:

| Bit[2] | Meaning |
|---|---|
| 0 | Frame <n> does not have a second view. CNTEL0Base<n> is RES0. |
| 1 | Frame <n> has a second view, CNTEL0Base<n>. |

If bit[0] is 0, bit[2] is RES0.

Bit[1] indicates whether both:

- Frame CNTBase<n> implements the virtual timer registers CNTV_CVAL, CNTV_TVAL, and CNTV_CTL.
- This CNTCTLBase frame implements the virtual timer offset register CNTVOFF<n>.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

The possible values of bit[1] are:

| Bit[1] | Meaning |
|---|---|
| 0 | Frame <n> does not have virtual capability. The virtual time and offset registers are RES0. |
| 1 | Frame <n> has virtual capability. The virtual time and offset registers are implemented. |

If bit[0] is 0, bit[1] is RES0.

Bit[0] indicates whether frame CNTBase<n> is implemented. The possible values of this bit are:

| Bit[0] | Meaning |
|---|---|
| 0 | Frame <n> not implemented. All registers associated with the frame are RES0. |
| 1 | Frame <n> is implemented. |

### Accessing the CNTTIDR:

CNTTIDR can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTCTLBase | 0x008 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.15 CNTV_CTL, Counter-timer Virtual Timer Control

The CNTV_CTL characteristics are:

**Purpose**

Control register for the virtual timer.

**Usage constraints**

This register is accessible as follows:

| Frame | Accessibility |
|---|---|
| CNTBaseN | Config-RW |
| CNTEL0BaseN | Config-RW |

If CNTV_CTL is implemented, it is implemented as a RW register in the CNTBaseN frame when both:

- The value of CNTACR<n>.RWVT is 1.
- Bits[1:0] of CNTTIDR.Frame<n> are 1.

Otherwise the encoding in the CNTBaseN frame is RES0.

When CNTV_CTL is implemented as a RW register in the CNTBaseN frame, if the value of CNTEL0ACR.EL0VTEN is 1 it is also implemented as a RW register in the CNTEL0BaseN frame if both:

- CNTEL0ACR.EL0VTEN is 1.
- Bit[2] of CNTTIDR.Frame<n> is 1.

Otherwise the CNTV_CTL register in the CNTEL0BaseN frame is RES0.

If CNTV_CTL is not implemented, the register location is RAZ/WI.

**Configurations**

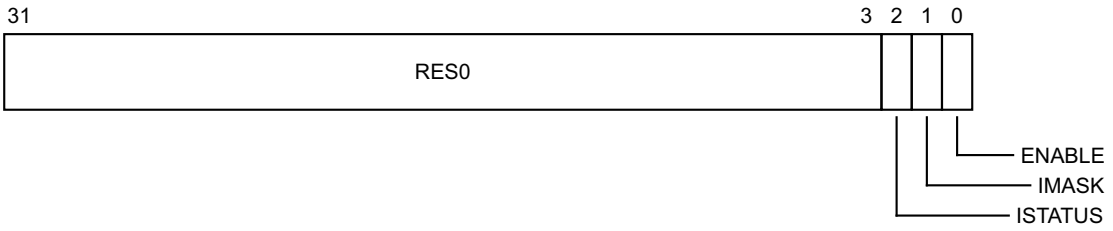
CNTV_CTL is optional to implement in the external register interface.

**Attributes**

CNTV_CTL is a 32-bit register.

#### Field descriptions

The CNTV_CTL bit assignments are:




**Bits [31:3]**

Reserved, RES0.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is asserted:

`0` Timer condition is not asserted.

`1` Timer condition is asserted.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer value meets the condition for the timer output to be asserted. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer output signal is asserted.

For more information see *Operation of the CompareValue views of the timers* on page D6-1904 and *Operation of the TimerValue views of the timers* on page D6-1904.

This bit is read-only.

**IMASK, bit [1]**

Timer output signal mask bit. Permitted values are:

`0` Timer output signal is not masked.

`1` Timer output signal is masked.

For more information, see the description of the ISTATUS bit.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

`0` Timer disabled.

`1` Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL continues to count down.

**Note**

Disabling the output signal might be a power-saving option.

## Accessing the CNTV_CTL:

CNTV_CTL can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | `0x03C` |
| Timer | CNTEL0BaseN | `0x03C` |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.16 CNTV_CVAL, Counter-timer Virtual Timer CompareValue

The CNTV_CVAL characteristics are:

**Purpose**

Holds the 64-bit compare value for the virtual timer.

**Usage constraints**

This register is accessible as follows:

| Frame | Accessibility |
|---|---|
| CNTBaseN | Config-RW |
| CNTEL0BaseN | Config-RW |

If CNTV_CVAL is implemented, it is implemented as a RW register in the CNTBaseN frame when both:

- The value of CNTACR<n>.RWVT is 1.
- Bits[1:0] of CNTTIDR.Frame<n> are 1.

Otherwise the encoding in the CNTBaseN frame is RES0.

When CNTV_CVAL is implemented as a RW register in the CNTBaseN frame, it is also implemented as a RW register in the CNTEL0BaseN frame if both:

- CNTEL0ACR.EL0VTEN is 1.
- Bit[2] of CNTTIDR.Frame<n> is 1.

Otherwise the CNTV_CVAL register in the CNTEL0BaseN frame is RES0.

If CNTV_CVAL is not implemented, the register location is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTV_CVAL register must be accessible as an atomic 64-bit value.

**Configurations**

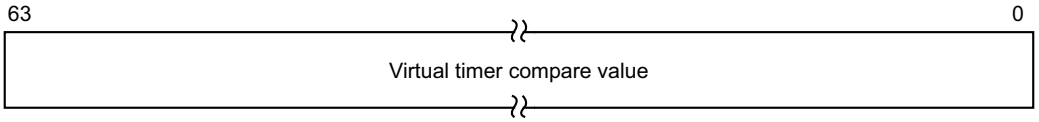
CNTV_CVAL is optional to implement in the external register interface.

**Attributes**

CNTV_CVAL is a 64-bit register.

#### Field descriptions

The CNTV_CVAL bit assignments are:




**Bits [63:0]**

Virtual timer compare value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTV_CVAL:

CNTV_CVAL[31:0] can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x030 |
| Timer | CNTEL0BaseN | 0x030 |

CNTV_CVAL[63:32] can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x034 |
| Timer | CNTEL0BaseN | 0x034 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.17 CNTV_TVAL, Counter-timer Virtual Timer TimerValue

The CNTV_TVAL characteristics are:

**Purpose**

Holds the timer value for the virtual timer.

**Usage constraints**

This register is accessible as follows:

| Frame | Accessibility |
|---|---|
| CNTBaseN | Config-RW |
| CNTEL0BaseN | Config-RW |

If CNTV_TVAL is implemented, it is implemented as a RW register in the CNTBaseN frame when both:

- The value of CNTACR<n>.RWVT is 1.
- Bits[1:0] of CNTTIDR.Frame<n> are 1.

Otherwise the encoding in the CNTBaseN frame is RES0.

When CNTV_TVAL is implemented as a RW register in the CNTBaseN frame, it is also implemented as RW in the CNTEL0BaseN frame if both:

- CNTEL0ACR.EL0VTEN is 1
- Bit[2] of CNTTIDR.Frame<n> is 1.

Otherwise the CNTV_TVAL register in the CNTEL0BaseN frame is RES0.

If CNTV_TVAL is not implemented, the register location is RAZ/WI.

**Configurations**

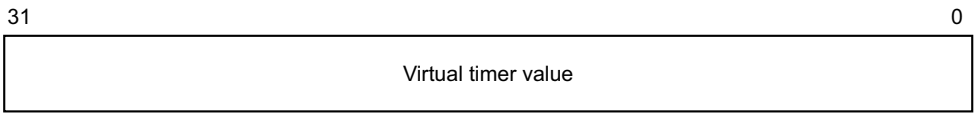
CNTV_TVAL is optional to implement in the external register interface.

**Attributes**

CNTV_TVAL is a 32-bit register.

## Field descriptions

The CNTV_TVAL bit assignments are:




**Bits [31:0]**

Virtual timer value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTV_TVAL:

CNTV_TVAL can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x038 |
| Timer | CNTEL0BaseN | 0x038 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.18 CNTVCT, Counter-timer Virtual Count

The CNTVCT characteristics are:

**Purpose**

Holds the 64-bit virtual count value.

**Usage constraints**

This register is accessible as follows:

| Frame | Accessibility |
|---|---|
| CNTBaseN | Config-RO |
| CNTEL0BaseN | Config-RO |

CNTVCT is implemented as a RO register in the CNTBaseN frame when both:

- CNTACR<n>.RVCT is 1
- Bit[0] of CNTTIDR.Frame<n> is 1.

Otherwise the encoding in the CNTBaseN frame is RES0.

When CNTVCT is implemented as a RO register in the CNTBaseN frame, it is also implemented as a RO register in the CNTEL0BaseN frame if both:

- CNTEL0ACR.EL0VCTEN is 1.
- Bit[2] of CNTTIDR.Frame<n> is 1.

Otherwise the CNTVCT register in the CNTEL0BaseN frame is RES0.

If the implementation supports 64-bit atomic accesses, then the CNTVCT register must be accessible as an atomic 64-bit value.

**Configurations**

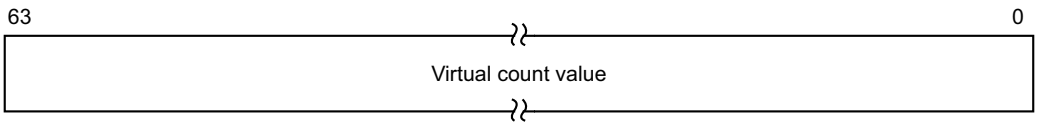
There are no configuration notes.

**Attributes**

CNTVCT is a 64-bit register.

#### Field descriptions

The CNTVCT bit assignments are:




**Bits [63:0]**

Virtual count value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTVCT:

CNTVCT[31:0] can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x008 |
| Timer | CNTEL0BaseN | 0x008 |

CNTVCT[63:32] can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTBaseN | 0x00C |
| Timer | CNTEL0BaseN | 0x00C |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.19 CNTVOFF{<n>}, Counter-timer Virtual Offset

The CNTVOFF{<n>} characteristics are:

**Purpose**

Holds the 64-bit virtual offset.

**Usage constraints**

This register is accessible as follows:

| Frame | Mnemonic | Accessibility |
|---|---|---|
| CNTCTLBase | CNTVOFF<n>, n = 1 to 7 | Config-RW |
| CNTBaseN | CNTVOFF | Config-RO |
| CNTEL0BaseN | CNTVOFF | Config-RO |

For each implemented CNTBaseN frame, it is IMPLEMENTATION DEFINED whether the timer distinguishes between real time and virtual time.

If the frame<n> timer supports this distinction and bits[1:0] of CNTTIDR.Frame<n> are 1, CNTVOFF<n> is implemented as a RW register in the CNTCTLBase frame. Otherwise the CNTVOFF<n> encoding in the CNTCTLBase frame is RES0.

In a system that implements both Secure and Non-secure states:

- CNTVOFF<n> is always accessible in Secure state.
- CNTNSAR.NS<n> determines whether CNTVOFF<n> is accessible in Non-secure state.

If CNTVOFF<n> is not implemented, the register location is RAZ/WI.

CNTVOFF is implemented as a RO register in the CNTBaseN frame when the value of CNTACR<n>.RVOFF is 1, otherwise the encoding in the CNTBaseN frame is RES0.

When CNTVOFF is implemented as a RO register in the CNTBaseN frame, and if bit[2] of CNTTIDR.Frame<n> is 1, CNTVOFF is also implemented as a RO register in the CNTEL0BaseN frame. When CNTVOFF is not implemented in the CNTBaseN frame, or when the value of bit[2] of CNTTIDR.Frame<n> is 0, the encoding in CNTEL0BaseN is RES0.

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF{<n>} register must be accessible as an atomic 64-bit value.

**Configurations**

CNTVOFF{<n>} is optional to implement in the external register interface.

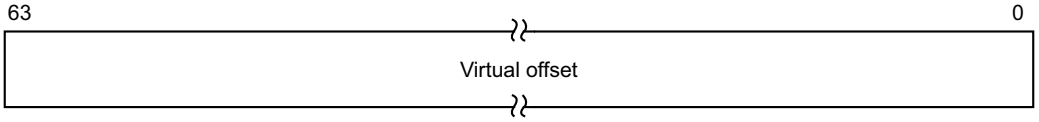
CNTACR<n>.RVOFF enables access to this register for frame CNTBaseN.

**Attributes**

CNTVOFF{<n>} is a 64-bit register.

#### Field descriptions

The CNTVOFF{<n>} bit assignments are:




**Bits [63:0]**

Virtual offset.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Accessing the CNTVOFF:

CNTVOFF{<n>} can be accessed through the memory-mapped interface:

| Component | Mnemonic | Frame | Offset |
|---|---|---|---|
| Timer | CNTVOFF<n>[31:0] | CNTCTLBase | 0x080 + 8n |
| Timer | CNTVOFF<n>[63:32] | CNTCTLBase | 0x084 + 8n |
| Timer | CNTVOFF[31:0] | CNTBaseN | 0x018 |
| Timer | CNTVOFF[63:32] | CNTBaseN | 0x01C |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### I3.5.20 CounterID<n>, Counter ID registers, n = 0 - 11

The CounterID<n> characteristics are:

**Purpose**

IMPLEMENTATION DEFINED identification registers 0 to 11 for the memory-mapped Generic Timer.

**Usage constraints**

This register is accessible as follows:

| Default |
|---|
| RO |

**Configurations**

These registers are implemented independently in each of the frames accessed through the different memory maps.

These registers are accessible in both Secure and Non-secure states.

If the implementation of the Counter ID registers requires an architecture version, the value for this version of the ARM Generic Timer is version 0.

The Counter ID registers can be implemented as a set of CoreSight ID registers, comprising Peripheral ID Registers and Component ID Registers. An implementation of these registers for the Generic Timer must use a Component class value of `0xF`.

**Attributes**

CounterID<n> is a 32-bit register.

#### Field descriptions

The CounterID<n> bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

#### Accessing the CounterID<n>:

CounterID<n> can be accessed through the memory-mapped interface:

| Component | Frame | Offset |
|---|---|---|
| Timer | CNTControlBase | `0xFD0` + 4n |
| Timer | CNTReadBase | `0xFD0` + 4n |
| Timer | CNTBaseN | `0xFD0` + 4n |
| Timer | CNTEL0BaseN | `0xFD0` + 4n |
| Timer | CNTCTLBase | `0xFD0` + 4n |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Part J

## Architectural Pseudocode

# Chapter J1
# ARMv8 Pseudocode

This chapter contains pseudocode that describes many features of the ARMv8 architecture. It contains the following sections:


- *Pseudocode for AArch64 operations* on page J1-5214.
- *Pseudocode for AArch32 operation* on page J1-5273.
- *Shared pseudocode* on page J1-5344.


**Note**

**Status of this appendix in the beta release document**

ARM is currently working to improve the organization and presentation of the pseudocode in this document, including providing improved linking within the pseudocode. Currently, this chapter contains the complete pseudocode library for ARMv8, split between:

- Functions, or versions of functions, that are specific to execution in AArch64 state.
- Functions, or versions of functions, that are specific to execution in AArch32state.
- Functions that apply to execution in either Execution state.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## J1.1 Pseudocode for AArch64 operations

This section holds the pseudocode for execution in AArch64 state. Functions listed in this section are identified as `AArch64.FunctionName`. Some of these functions have an equivalent AArch32 function, `AArch32.FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch64/debug/breakpoint`.

The top-level sections of the AArch64 pseudocode hierarchy are:

- *aarch64/debug*.
- *aarch64/exceptions* on page J1-5220.
- *aarch64/functions* on page J1-5237.
- *aarch64/instrs* on page J1-5250.
- *aarch64/translation* on page J1-5257.

### J1.1.1 aarch64/debug

This section includes the following pseudocode functions:

- *aarch64/debug/breakpoint/AArch64.BreakpointMatch*.
- *aarch64/debug/breakpoint/AArch64.BreakpointValueMatch* on page J1-5215.
- *aarch64/debug/breakpoint/AArch64.StateMatch* on page J1-5216.
- *aarch64/debug/enables/AArch64.GenerateDebugExceptions* on page J1-5217.
- *aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom* on page J1-5217.
- *aarch64/debug/pmu/AArch64.CheckForPMUOverflow* on page J1-5217.
- *aarch64/debug/pmu/AArch64.CountEvents* on page J1-5217.
- *aarch64/debug/pmu/AArch64.ProfilingProhibited* on page J1-5218.
- *aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState* on page J1-5218.
- *aarch64/debug/watchpoint/AArch64.WatchpointByteMatch* on page J1-5219.
- *aarch64/debug/watchpoint/AArch64.WatchpointMatch* on page J1-5220.

#### aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.BRPs);

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT == '0x01';
    linked_to = FALSE;

    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                     linked, DBGBCR_EL1[n].LBN, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAnyAArch32() && size == 4 then                 // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool();

    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // at the address DBGBVR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool();

    match = value_match && state_match && enabled;

    return match;
```

### aarch64/debug/breakpoint/AArch64.BreakpointValueMatch

```
// AArch64.BreakpointValueMatch()
// ==============================

boolean AArch64.BreakpointValueMatch(integer n, bits(64) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //    matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.

    // If a non-existant breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(ID_AA64DFR0_EL1.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(ID_AA64DFR0_EL1.BRPs));
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking.)
    if DBGBCR_EL1[n].E == '0' then return FALSE;

    context_aware = (n >= UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    type = DBGBCR_EL1[n].BT;
    if (type == 'x1xx' ||                                           // Reserved
          (type != '0x0x' && !context_aware) ||                     // Context matching
          (type == '1xxx' && !HaveEL(EL2))) then                    // VMID match
        (c, type) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = type == '0x0x';
    match_vmid = type == '10xx';
    match_cid  = type == 'x01x';
    linked     = type == 'xxx1';

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return FALSE;

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return FALSE;

    // Do the comparison.
    if match_addr then
        byte = UInt(vaddress<1:0>);
        if HaveAnyAArch32() then
            // T32 instructions can be executed at EL0 in an AArch64 translation regime.
            assert byte IN {0,2};                 // "vaddress" is halfword aligned.
            byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0;                     // "vaddress" is word aligned
            byte_select_match = TRUE;             // DBGBCR_EL1[n].BAS<byte> is RES1
        top = AddrTop(vaddress);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        BVR_match = vaddress<top:2> == DBGBVR_EL1[n]<top:2> && byte_select_match;
    elsif match_cid then
        BVR_match = (PSTATE.EL IN {EL0,EL1} && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
    if match_vmid then
        BXVR_match = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                      VTTBR_EL2.VMID == DBGBVR_EL1[n]<39:32>);

    match = (!match_vmid || BXVR_match) && (!(match_addr || match_cid) || BVR_match);
    return match;
```

### aarch64/debug/breakpoint/AArch64.StateMatch

```
// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean ispriv)
    // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
    // "linked" is TRUE if this is a linked breakpoint/watchpoint type.

    // If parameters are set to a reserved type, behaves either as disabled or not a reserved-type
    if ((HMC:SSC:PxC) IN {'0xx00','011xx','100x0','101x0','11010','11101','1111x'} || // Reserved
          (SSC IN {'01','10'} && !HaveEL(EL3)) ||                                     // No EL3
          (HMC:SSC != '000' && !HaveEL(EL3) && !HaveEL(EL2))) then                    // No EL3/EL2
        (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
    EL2_match = HaveEL(EL2) && HMC == '1';
    EL1_match = PxC<0> == '1';
    EL0_match = PxC<1> == '1';

    case PSTATE.EL of
        when EL3  priv_match = EL3_match;
        when EL2  priv_match = EL2_match;
        when EL1  priv_match = if ispriv then EL1_match else EL0_match;
        when EL0  priv_match = EL0_match;

    case SSC of
        when '00'  security_state_match = TRUE;         // Both
        when '01'  security_state_match = !IsSecure();  // Non-secure only
        when '10'  security_state_match = IsSecure();   // Secure only
        when '11'  security_state_match = TRUE;         // Both

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));
        last_ctx_cmp = UInt(ID_AA64DFR0_EL1.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(64) UNKNOWN;
        linked_to = TRUE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

### aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =================================

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);
```

### aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = HaveEL(EL2) && !secure && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    if HaveEL(EL3) && secure then
        enabled = MDCR_EL3.SDD == '0' && from != EL3;
    else
        enabled = TRUE;

    target = if route_to_el2 then EL2 else EL1;
    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';

    return enabled;
```

### aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```
// AArch64.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch64.CheckForPMUOverflow()

    pmuirq = (PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSSET_EL0<31> == '1');
    for n = 0 to UInt(PMCR_EL0.N) - 1
        if HaveEL(EL2) then
            E = (if n < UInt(MDCR_EL2.HPMN) then PMCR_EL0.E else MDCR_EL2.HPME);
        else
            E = PMCR_EL0.E;
        if E == '1' && PMINTENSET_EL1<n> == '1' && PMOVSSET_EL0<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

    return pmuirq;
```

### aarch64/debug/pmu/AArch64.CountEvents

```
// AArch64.CountEvents()
// =====================
// Return TRUE if counter "n" should count its event.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean AArch64.CountEvents(integer n)
    assert(n == 31 || n < UInt(PMCR_EL0.N));

    // Event counting is disabled in Debug state
    debug = Halted();

    // Event counting might be prohibited
    prohibited = AArch64.ProfilingProhibited(IsSecure(), PSTATE.EL);
    if prohibited && n == 31 && PMCR_EL0.DP == '0' then prohibited = FALSE;

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M} bits
    filter = (if n == 31 then PMCCFILTR_EL0<31:26> else PMEVTYPER_EL0[n]<31:26>);

    M = if !HaveEL(EL3) then '0' else (filter<5> EOR filter<0>);
    H = if !HaveEL(EL2) then '0' else filter<1>;
    P = filter<5>;  U = filter<4>;
    if !IsSecure() && HaveEL(EL3) then
        P = P EOR filter<3>;  U = U EOR filter<2>;

    case PSTATE.EL of
        when EL0  filtered = U == '1';
        when EL1  filtered = P == '1';
        when EL2  filtered = H == '0';
        when EL3  filtered = M == '1';

    if HaveEL(EL2) then
        E = (if n < UInt(MDCR_EL2.HPMN) then PMCR_EL0.E else MDCR_EL2.HPME);
    else
        E = PMCR_EL0.E;
    enabled = (E == '1' && PMCNTENSET_EL0<n> == '1');

    return !debug && !prohibited && !filtered && enabled;
```

## aarch64/debug/pmu/AArch64.ProfilingProhibited

```
// AArch64.ProfilingProhibited()
// =============================
// Determine whether event counting is prohibited in the current state.

boolean AArch64.ProfilingProhibited(boolean secure, bits(2) el)

    // Events are always counted in Non-secure state.
    if !secure then return FALSE;

    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    if !HaveEL(EL3) then return FALSE;

    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    if MDCR_EL3.SPME == '1' then return FALSE;

    // * Allowed by the IMPLEMENTATION DEFINED authentication interface
    if ExternalSecureNoninvasiveDebugEnabled() then return FALSE;

    return TRUE;
```

## aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception Level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;  PSTATE.nRW = '0';  PSTATE.SP = '1';

    SPSR[] = bits(32) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then                                 // Coming from AArch32
        PSTATE.IT = '00000000';  PSTATE.T = '0';    // PSTATE.J is RES0

    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.
    EndOfInstruction();
```

## aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// =============================

boolean AArch64.WatchpointByteMatch(integer n, bits(64) vaddress)

    top = AddrTop(vaddress);
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool();
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                      // Not contiguous
            byte_select_match = ConstrainUnpredictableBool();
            bottom = 3;                                         // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;             // Disabled
            when Constraint_NONE      mask = 0;                 // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool();
    else
        WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

    return WVR_match && byte_select_match;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
                                boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.WRPs);

    // "ispriv" is FALSE for LDTR/STTR instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';

    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, DBGWCR_EL1[n].LBN, ispriv);

    ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

## J1.1.2 aarch64/exceptions

This section includes the following pseudocode functions:


- *aarch64/exceptions/aborts/AArch64.Abort* on page J1-5221.
- *aarch64/exceptions/aborts/AArch64.AbortSyndrome* on page J1-5221.
- *aarch64/exceptions/aborts/AArch64.CheckPCAlignment* on page J1-5222.
- *aarch64/exceptions/aborts/AArch64.DataAbort* on page J1-5222.
- *aarch64/exceptions/aborts/AArch64.InstructionAbort* on page J1-5222.
- *aarch64/exceptions/aborts/AArch64.PCAlignmentFault* on page J1-5222.
- *aarch64/exceptions/aborts/AArch64.SPAlignmentFault* on page J1-5223.
- *aarch64/exceptions/asynch/AArch64.TakePhysicalFIQException* on page J1-5223.
- *aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException* on page J1-5223.
- *aarch64/exceptions/asynch/AArch64.TakePhysicalSystemErrorException* on page J1-5224.
- *aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException* on page J1-5224.
- *aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException* on page J1-5224.
- *aarch64/exceptions/asynch/AArch64.TakeVirtualSystemErrorException* on page J1-5225.
- *aarch64/exceptions/debug/AArch64.BreakpointException* on page J1-5225.
- *aarch64/exceptions/debug/AArch64.SoftwareBreakpoint* on page J1-5225.
- *aarch64/exceptions/debug/AArch64.SoftwareStepException* on page J1-5226.
- *aarch64/exceptions/debug/AArch64.VectorCatchException* on page J1-5226.
- *aarch64/exceptions/debug/AArch64.WatchpointException* on page J1-5226.
- *aarch64/exceptions/exceptions/AArch64.ExceptionClass* on page J1-5227.
- *aarch64/exceptions/exceptions/AArch64.ReportException* on page J1-5227.
- *aarch64/exceptions/exceptions/AArch64.ResetControlRegisters* on page J1-5228.
- *aarch64/exceptions/exceptions/AArch64.TakeReset* on page J1-5228.
- *aarch64/exceptions/ieeefp/AArch64.FPTrappedException* on page J1-5229.
- *aarch64/exceptions/syscalls/AArch64.CallHypervisor* on page J1-5229.
- *aarch64/exceptions/syscalls/AArch64.CallSecureMonitor* on page J1-5229.
- *aarch64/exceptions/syscalls/AArch64.CallSupervisor* on page J1-5230.
- *aarch64/exceptions/takeexception/AArch64.TakeException* on page J1-5230.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap* on page J1-5231.
- *aarch64/exceptions/traps/AArch64.CPRegTrap* on page J1-5231.
- *aarch64/exceptions/traps/AArch64.CPRegTrapSyndrome* on page J1-5231.
- *aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps* on page J1-5232.
- *aarch64/exceptions/traps/AArch64.CheckCoprocInstr* on page J1-5233.
- *aarch64/exceptions/traps/AArch64.CheckCoprocInstrTraps* on page J1-5234.
- *aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled* on page J1-5234.
- *aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap* on page J1-5234.
- *aarch64/exceptions/traps/AArch64.CheckForSMCTrap* on page J1-5234.
- *aarch64/exceptions/traps/AArch64.CheckForWFxTrap* on page J1-5235.
- *aarch64/exceptions/traps/AArch64.CheckIllegalState* on page J1-5235.
- *aarch64/exceptions/traps/AArch64.MonitorModeTrap* on page J1-5235.
- *aarch64/exceptions/traps/AArch64.SystemRegisterTrap* on page J1-5236.
- *aarch64/exceptions/traps/AArch64.UndefinedFault* on page J1-5236.
- *aarch64/exceptions/traps/AArch64.WFxTrap* on page J1-5236.
- *aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64* on page J1-5237.

## aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

    if IsDebugException(fault) then
        if fault.acctype == AccType_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(vaddress, fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
```

## aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception type, FaultRecord fault, bits(64) vaddress)

    exception = ExceptionSyndrome(type);

    d_side = type IN {Exception_DataAbort, Exception_Watchpoint};

    exception.syndrome = FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.ipaddress = fault.ipaddress;
    else
        exception.ipavalid = FALSE;

    return exception;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// ==========================

AArch64.CheckPCAlignment()

    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

### aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// ===================

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || IsSecondStage(fault)));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || IsSecondStage(fault)));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// ==========================
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PCAlignment);
    exception.vaddress = ThisInstrAddr();

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SPAlignment);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/asynch/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// ==================================

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
    exception = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0,EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0,EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/asynch/AArch64.TakePhysicalSystemErrorException

```
// AArch64.TakePhysicalSystemErrorException()
// ==========================================

AArch64.TakePhysicalSystemErrorException(boolean syndrome_valid, bits(24) syndrome)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    if syndrome_valid then
        exception.syndrome<24> = '1';
        exception.syndrome<23:0> = syndrome;

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =================================

AArch64.TakeVirtualFIQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;

    exception = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =================================

AArch64.TakeVirtualIRQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/asynch/AArch64.TakeVirtualSystemErrorException

```
// AArch64.TakeVirtualSystemErrorException()
// =========================================

AArch64.TakeVirtualSystemErrorException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';  // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    if boolean IMPLEMENTATION_DEFINED "Virtual System Error syndrome valid" then
        exception.syndrome<24> = '1';
        exception.syndrome<23:0> = bits(24) IMPLEMENTATION_DEFINED "Virtual System Error syndrome";

    HCR_EL2.VSE = '0';
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/debug/AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =============================

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// ============================

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    exception.syndrome<15:0> = immediate;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// ===============================

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/debug/AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert HaveEL(EL2) && !IsSecure() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## aarch64/exceptions/exceptions/AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// ========================
// Return the Exception Class and Instruction Length fields for reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception type, bits(2) target_el)

    il = if ThisInstrLength() == 32 then '1' else '0';
    from_32 = UsingAArch32();
    assert from_32 || il == '1';            // AArch64 instructions always 32-bit

    case type of
        when Exception_Uncategorized        ec = 0x00; il = '1';
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03;          assert from_32;
        when Exception_CP15RRTTrap          ec = 0x04;          assert from_32;
        when Exception_CP14RTTrap           ec = 0x05;          assert from_32;
        when Exception_CP14DTTrap           ec = 0x06;          assert from_32;
        when Exception_AdvSIMDFPAccessTrap  ec = 0x07;
        when Exception_FPIDTrap             ec = 0x08;
        when Exception_CP14RRTTrap          ec = 0x0C;          assert from_32;
        when Exception_IllegalState         ec = 0x0E; il = '1';
        when Exception_SupervisorCall       ec = 0x11;
        when Exception_HypervisorCall       ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_SystemRegisterTrap   ec = 0x18;          assert !from_32;
        when Exception_InstructionAbort     ec = 0x20; il = '1';
        when Exception_PCAlignment          ec = 0x22; il = '1';
        when Exception_DataAbort            ec = 0x24;
        when Exception_SPAlignment          ec = 0x26; il = '1'; assert !from_32;
        when Exception_FPTrappedException   ec = 0x28;
        when Exception_SError               ec = 0x2F; il = '1';
        when Exception_Breakpoint           ec = 0x30; il = '1';
        when Exception_SoftwareStep         ec = 0x32; il = '1';
        when Exception_Watchpoint           ec = 0x34; il = '1';
        when Exception_SoftwareBreakpoint   ec = 0x38;
        when Exception_VectorCatch          ec = 0x3A; il = '1'; assert from_32;
        otherwise                           Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    return (ec,il);
```

## aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception type = exception.type;

    (ec,il) = AArch64.ExceptionClass(type, target_el);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = ec<5:0>:il:iss;

    if type IN {Exception_InstructionAbort, Exception_PCAlignment, Exception_DataAbort,
                Exception_Watchpoint} then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if target_el == EL2 then
        if exception.ipavalid then
            HPFAR_EL2<39:4> = exception.ipaddress<47:12>;
        else
            HPFAR_EL2<39:4> = bits(36) UNKNOWN;

    return;
```

## aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

```
AArch64.ResetControlRegisters(boolean cold_reset);
```

## aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// ===================
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert !HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset the system registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';              // Select stack pointer
    PSTATE.<D,A,I,F>  = '1111';   // All asynchronous exceptions masked
    PSTATE.SS = '0';              // Clear software step bit
    PSTATE.IL = '0';              // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;                      // IMPLEMENTATION DEFINED reset vector
    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:PAMax()>) && IsZero(rv<1:0>);

    BranchTo(rv, BranchType_UNKNOWN);
```

### aarch64/exceptions/ieeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// ============================

AArch64.FPTrappedException(boolean is_ase, integer element, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome(Exception_FPTrappedException);
    exception.syndrome<23> = '1';                                  // TFV
    if is_ase then exception.syndrome<10:8> = element<2:0>;    // VECITR
    exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF

    route_to_el2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/syscalls/AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// ========================
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_HypervisorCall);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// ===========================

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_MonitorCall);
    exception.syndrome<15:0> = immediate;

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/syscalls/AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// ========================
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();

    route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/takeexception/AArch64.TakeException

```
// AArch64.TakeException()
// =======================
// Take an exception to an Exception Level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception,
                      bits(64) preferred_exception_return, integer vect_offset)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    if UInt(target_el) > UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == EL3 then
            if !IsSecure() && HaveEL(EL2) then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
        else
            lower_32 = ELUsingAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

    elsif PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    spsr = GetPSRFromPSTATE();

    if !(exception.type IN {Exception_IRQ, Exception_FIQ}) then
        AArch64.ReportException(exception, target_el);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    PSTATE.EL = target_el;  PSTATE.nRW = '0';  PSTATE.SP = '1';

    SPSR[] = spsr;
    ELR[] = preferred_exception_return;

    PSTATE.SS = '0';
    PSTATE.<D,A,I,F> = '1111';
    PSTATE.IL = '0';
    if from_32 then                                 // Coming from AArch32
        PSTATE.IT = '00000000';  PSTATE.T = '0';    // PSTATE.J is RES0

    BranchTo(VBAR[] + vect_offset, BranchType_EXCEPTION);
    EndOfInstruction();
```

### aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// ============================
// Trapped access to Advanced SIMD or FP registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        exception = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        // No syndrome information when taken to AArch64 state
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
```

### aarch64/exceptions/traps/AArch64.CPRegTrap

```
// AArch64.CPRegTrap()
// ===================
// Trapped AArch32 CP14 and CP15 access other than due to CPTR_EL2 or CPACR_EL1.

AArch64.CPRegTrap(bits(2) target_el, bits(32) aarch32_instr)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.CPRegTrapSyndrome(aarch32_instr);

    if target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/traps/AArch64.CPRegTrapSyndrome

```
// AArch64.CPRegTrapSyndrome()
// ===========================
// Return the syndrome information for coprocessor register traps other than
// due to HCPTR or CPACR

ExceptionRecord AArch64.CPRegTrapSyndrome(bits(32) instr)

    ExceptionRecord exception;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    cpnum = UInt(instr<11:8>);

    bits(20) iss = Zeros();
    if instr<27:24> == '1110' && instr<4> == '1' && instr<31:28> != '1111' then
        // MRC/MCR
        case cpnum of
            when 10    exception = ExceptionSyndrome(Exception_FPIDTrap);
            when 14    exception = ExceptionSyndrome(Exception_CP14RTTrap);
            when 15    exception = ExceptionSyndrome(Exception_CP15RTTrap);
            otherwise  Unreachable();
        iss<19:17> = instr<7:5>;    // opc2
        iss<16:14> = instr<23:21>;  // opc1
        iss<13:10> = instr<19:16>;  // CRn
        iss<9:5>   = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>; // Rt
        iss<4:1>   = instr<3:0>;    // CRm
    elsif instr<27:21> == '1100010' && instr<31:28> != '1111' then
        // MRRC/MCRR
        case cpnum of
            when 14    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
            when 15    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
            otherwise  Unreachable();
        iss<19:16> = instr<7:4>;    // opc1
        iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rt2
        iss<9:5>   = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>; // Rt
        iss<4:1>   = instr<3:0>;    // CRm
    elsif instr<27:25> == '110' && instr<31:28> != '1111' then
        // LDC/STC
        assert cpnum == 14;
        exception = ExceptionSyndrome(Exception_CP14DTTrap);
        iss<19:12> = instr<7:0>;    // imm8
        iss<4>     = instr<23>;     // U
        iss<2:1>   = instr<24,21>;  // P,W
        if instr<19:16> == '1111' then    // Literal addressing
            iss<9:5> = bits(5) UNKNOWN;
            iss<3>   = '1';
        else
            iss<9:5> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rn
            iss<3>   = '0';
    else
        Unreachable();
    iss<0> = instr<20>;     // Direction
    exception.syndrome<19:0> = iss;

    return exception;
```

## aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32 CP15 traps in HSTR_EL2 and HCR_EL2.

boolean AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR_EL2<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !(major IN {4,14}) && HSTR_EL2<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR_EL2.TIDCP
        if (HCR_EL2.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
             (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            return TRUE;

    return FALSE;
```

### aarch64/exceptions/traps/AArch64.CheckCoprocInstr

```
// AArch64.CheckCoprocInstr()
// ==========================
// Check AArch32 coprocessor instruction for enables and disables

AArch64.CheckCoprocInstr(bits(32) instr)
    assert instr<11:8> != '101x';

    cp_num = UInt(instr<11:8>);

    // Decode the AArch32 coprocessor instruction
    if instr<27:24> == '1110' && instr<4> == '1' then  // MRC/MCR
        cprt = TRUE;  cpdt = FALSE;  nreg = 1;
        opc1 = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn  = UInt(instr<19:16>);
        CRm  = UInt(instr<3:0>);
    elsif instr<27:21> == '1100010' then               // MRRC/MCRR
        cprt = TRUE;  cpdt = FALSE;  nreg = 2;
        opc1 = UInt(instr<7:4>);
        CRm  = UInt(instr<3:0>);
    elsif instr<27:25> == '110' then                   // LDC/STC
        cprt = FALSE;  cpdt = TRUE;  nreg = 0;
        lng  = instr<22> == '1';
        opc1 = 0;
        CRn  = UInt(instr<15:12>);
    else                                               // CDP
        cprt = FALSE;  cpdt = FALSE;  nreg = 0;
        opc1 = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn  = UInt(instr<19:16>);
        CRm  = UInt(instr<3:0>);
    two = instr<31:28> == '1111';                      // MRC2/MCR2/etc.

    // Coarse-grain decode into CP14 or CP15 operations. Each of the CPxxxInstrDecode functions
    // returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
    if cp_num == 14 then
        // LDC and STC only supported for c5, CDP, and MRC2, MCR2, etc. not supported for CP14
        if (cpdt && (CRn != 5 || lng)) || (!cprt && !cpdt) || two then
            allocated = FALSE;
        else
            // Coarse-grained decode of CP14 based on opc1 field
            case opc1 of
                when 0     allocated = CP14DebugInstrDecode(instr);
                when 1     allocated = CP14TraceInstrDecode(instr);
                when 7     allocated = CP14JazelleInstrDecode(instr);    // JIDR only
                otherwise  allocated = FALSE;          // All other values are unallocated

    elsif cp_num == 15 then
        // LDC, STC, and CDP, and MRC2, MCR2, etc. not supported by CP15
        if !cprt || two then
            allocated = FALSE;
        else
            allocated = CP15InstrDecode(instr);

            // Coarse-grain traps to EL2 have a higher priority than Undefined Instruction
            if AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm) then
                // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
                // Non-secure User mode is UNDEFINED when the trap is disabled, then it is
                // IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
                // when the trap is enabled.
                if PSTATE.EL == EL0 && !IsSecure() && !allocated then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                    if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at NS EL0" then
                        UNDEFINED;
                AArch64.CPRegTrap(EL2, instr);

    else
        allocated = FALSE;                              // All other coprocessors are unallocated

    if !allocated then
        UNDEFINED;

    // If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
    AArch64.CheckCoprocInstrTraps(instr);

    return;
```

### aarch64/exceptions/traps/AArch64.CheckCoprocInstrTraps

```
AArch64.CheckCoprocInstrTraps(bits(32) instr);
```

### aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// ===============================
// Check against CPACR_EL1.

AArch64.CheckFPAdvSIMDEnabled()

    if PSTATE.EL IN {EL0, EL1} then
        // Check if access disabled in CPACR_EL1
        case CPACR_EL1.FPEN of
            when 'x0'  disabled = TRUE;
            when '01'  disabled = PSTATE.EL == EL0;
            when '11'  disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap();               // Also check against CPTR_EL2 and CPTR_EL3
```

### aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()

    if HaveEL(EL2) && !IsSecure() then
        // Check if access disabled in CPTR_EL2
        if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

    return;
```

### aarch64/exceptions/traps/AArch64.CheckForSMCTrap

```
// AArch64.CheckForSMCTrap()
// =========================
// Check for trap on SMC instruction

AArch64.CheckForSMCTrap(bits(16) imm)

    route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} && HCR_EL2.TSC == '1';
    if route_to_el2 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/traps/AArch64.CheckForWFxTrap

```
// AArch64.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    case target_el of
        when EL1  trap = (if is_wfe then SCTLR_EL1.nTWE else SCTLR_EL1.nTWI) == '0';
        when EL2  trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3  trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';

    if trap then
        AArch64.WFxTrap(target_el, is_wfe);
```

### aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()

    if PSTATE.IL == '1' then
        route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;

        exception = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/exceptions/traps/AArch64.SystemRegisterTrap

```
// AArch64.SystemRegisterTrap()
// ============================
// Trapped system register access other than due to CPTR_EL2 and CPACR_EL1

AArch64.SystemRegisterTrap(bits(2) target_el, bits(2) op0, bits(3) op2, bits(3) op1, bits(4) crn,
                           bits(5) rt, bits(4) crm, bit dir)
    assert UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
    exception.syndrome<21:20> = op0;
    exception.syndrome<19:17> = op2;
    exception.syndrome<16:14> = op1;
    exception.syndrome<13:10> = crn;
    exception.syndrome<9:5>   = rt;
    exception.syndrome<4:1>   = crm;
    exception.syndrome<0>     = dir;

    if target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/traps/AArch64.UndefinedFault

```
// AArch64.UndefinedFault()
// ========================

AArch64.UndefinedFault()

    route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =================

AArch64.WFxTrap(bits(2) target_el, boolean is_wfe)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<0> = if is_wfe then '1' else '0';

    if target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =========================
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

## J1.1.3 aarch64/functions

This section includes the following pseudocode functions:


- *aarch64/functions/aborts/AArch64.CreateFaultRecord* on page J1-5238.
- *aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass* on page J1-5238.
- *aarch64/functions/exclusive/AArch64.IsExclusiveVA* on page J1-5239.
- *aarch64/functions/exclusive/AArch64.MarkExclusiveVA* on page J1-5239.
- *aarch64/functions/exclusive/AArch64.SetExclusiveMonitors* on page J1-5239.
- *aarch64/functions/fusedrstep/FPRSqrtStepFused* on page J1-5239.
- *aarch64/functions/fusedrstep/FPRecipStepFused* on page J1-5240.
- *aarch64/functions/memory/AArch64.CheckAlignment* on page J1-5240.
- *aarch64/functions/memory/AArch64.MemSingle* on page J1-5240.
- *aarch64/functions/memory/CheckSPAlignment* on page J1-5241.
- *aarch64/functions/memory/Mem* on page J1-5241.
- *aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers* on page J1-5242.
- *aarch64/functions/registers/AArch64.ResetGeneralRegisters* on page J1-5243.
- *aarch64/functions/registers/AArch64.ResetSIMDFPRegisters* on page J1-5243.
- *aarch64/functions/registers/AArch64.ResetSpecialRegisters* on page J1-5243.
- *aarch64/functions/registers/AArch64.ResetSystemRegisters* on page J1-5244.
- *aarch64/functions/registers/PC* on page J1-5244.
- *aarch64/functions/registers/SP* on page J1-5244.
- *aarch64/functions/registers/V* on page J1-5244.
- *aarch64/functions/registers/Vpart* on page J1-5245.
- *aarch64/functions/registers/X* on page J1-5245.
- *aarch64/functions/sysregisters/ELR* on page J1-5246.
- *aarch64/functions/sysregisters/ESR* on page J1-5246.
- *aarch64/functions/sysregisters/ESRType* on page J1-5247.
- *aarch64/functions/sysregisters/FAR* on page J1-5247.
- *aarch64/functions/sysregisters/MAIR* on page J1-5247.
- *aarch64/functions/sysregisters/MAIRType* on page J1-5248.
- *aarch64/functions/sysregisters/SCTLR* on page J1-5248.
- *aarch64/functions/sysregisters/SCTLRType* on page J1-5248.
- *aarch64/functions/sysregisters/VBAR* on page J1-5248.
- *aarch64/functions/system/AArch64.CheckAdvSIMDFPSystemRegisterTraps* on page J1-5248.
- *aarch64/functions/system/AArch64.CheckSystemRegisterTraps* on page J1-5249.
- *aarch64/functions/system/AArch64.CheckUnallocatedSystemAccess* on page J1-5249.
- *aarch64/functions/system/CheckSystemAccess* on page J1-5249.
- *aarch64/functions/system/SysOp_R* on page J1-5250.
- *aarch64/functions/system/SysOp_W* on page J1-5250.
- *aarch64/functions/system/System_Get* on page J1-5250.
- *aarch64/functions/system/System_Put* on page J1-5250.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/functions/aborts/AArch64.CreateFaultRecord

```
// AArch64.CreateFaultRecord()
// ===========================

FaultRecord AArch64.CreateFaultRecord(Fault type, bits(48) ipaddress,
                                      integer level, AccType acctype, boolean write, bit extflag,
                                      boolean secondstage, boolean s2fs1walk)

    FaultRecord fault;
    fault.type = type;
    fault.domain = bits(4) UNKNOWN;         // Not used from AArch64
    fault.debugmoe = bits(4) UNKNOWN;       // Not used from AArch64
    fault.ipaddress = ipaddress;
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fs1walk = s2fs1walk;

    return fault;
```

### aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// ===============================

// Return TRUE if the Exclusive Monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusive Monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/functions/exclusive/AArch64.IsExclusiveVA

```
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

### aarch64/functions/exclusive/AArch64.MarkExclusiveVA

```
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

### aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// ==============================

// Sets the Exclusive Monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = (address != Align(address, size));

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

### aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// ==================

bits(N) FPRSqrtStepFused(bits(N) op1, bits(N) op2)
    assert N IN {32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1,sign1,value1) = FPUnpack(op1, FPCR);
    (type2,sign2,value2) = FPUnpack(op2, FPCR);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// ==================

bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
    assert N IN {32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1,sign1,value1) = FPUnpack(op1, FPCR);
    (type2,sign2,value2) = FPUnpack(op2, FPCR);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```

### aarch64/functions/memory/AArch64.CheckAlignment

```
// AArch64.CheckAlignment()
// ========================

boolean AArch64.CheckAlignment(bits(64) address, integer size, AccType acctype, boolean iswrite)

    aligned = (address == Align(address, size));
    A = SCTLR[].A;

    if !aligned && (acctype == AccType_ATOMIC || acctype == AccType_ORDERED || A == '1') then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

### aarch64/functions/memory/AArch64.MemSingle

```
// AArch64.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    value = _Mem[memaddrdesc, size, acctype];
    return value;

// AArch64.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8)
value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    _Mem[memaddrdesc, size, acctype] = value;
    return;
```

## aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// ==================
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];

    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR_EL1.SA0 != '0');
    else
        stack_align_check = (SCTLR[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```

## aarch64/functions/memory/Mem

```
// Mem[] - non-assignment (read) form
// ==================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    integer i;
    boolean iswrite = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    atomic = (aligned && !(acctype IN {AccType_VEC, AccType_VECSTREAM})) || size == 1;

    if !atomic then
        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch64.MemSingle[address, size, acctype, aligned];

    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem[] - assignment (write) form
// ===============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
    integer i;
    boolean iswrite = TRUE;

    if BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    atomic = (aligned && !(acctype IN {AccType_VEC, AccType_VECSTREAM})) || size == 1;

    if !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    else
        AArch64.MemSingle[address, size, acctype, aligned] = value;
    return;
```

## aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```
// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to  AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32();         // Always called from AArch32 state before entering AArch64 state

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        first = 0;  last = 14;  include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0,EL1} && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool() then
            _R[n]<63:32> = Zeros();

    return;
```

### aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// ===============================

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i] = bits(64) UNKNOWN;

    return;
```

### aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// ==============================

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;

    return;
```

### aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// ===============================

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;
    ELR_EL1  = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(32) UNKNOWN;
        ELR_EL2  = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(32) UNKNOWN;
        ELR_EL3  = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq = bits(32) UNKNOWN;
        SPSR_irq = bits(32) UNKNOWN;
        SPSR_abt = bits(32) UNKNOWN;
        SPSR_und = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    DSPSR_EL0 = bits(32) UNKNOWN;

    return;
```

### aarch64/functions/registers/AArch64.ResetSystemRegisters

```
AArch64.ResetSystemRegisters(boolean cold_reset);
```

### aarch64/functions/registers/PC

```
// PC - non-assignment form
// ========================
// Read program counter.

bits(64) PC[]
    return _PC;
```

### aarch64/functions/registers/SP

```
// SP[] - assignment form
// ======================
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
    assert width IN {32,64};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value);
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = ZeroExtend(value);
            when EL1  SP_EL1 = ZeroExtend(value);
            when EL2  SP_EL2 = ZeroExtend(value);
            when EL3  SP_EL3 = ZeroExtend(value);
    return;

// SP[] - non-assignment form
// ==========================
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
    assert width IN {8,16,32,64};
    if PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0<width-1:0>;
            when EL1  return SP_EL1<width-1:0>;
            when EL2  return SP_EL2<width-1:0>;
            when EL3  return SP_EL3<width-1:0>;
```

### aarch64/functions/registers/V

```
// V[] - assignment form
// =====================
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    _V[n] = ZeroExtend(value);
    return;

// V[] - non-assignment form
// =========================
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _V[n]<width-1:0>;
```

### aarch64/functions/registers/Vpart

```
// Vpart[] - non-assignment form
// =============================
// Reads a 128-bit SIMD&FP register in up to two parts:
//  part 0 returns the bottom 8, 16, 32 or 64 bits of the register;
//  part 1 returns only the top 64 bits of the register.

bits(width) Vpart[integer n, integer part]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        return _V[n]<width-1:0>;
    else
        assert width == 64;
        return _V[n]<127:64>;

// Vpart[] - assignment form
// =========================
// Write a 128-bit SIMD&FP register in up to two parts:
//  part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
//  part 1 inserts a 64-bit value into the top 64 bits of the register.

Vpart[integer n, integer part] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        _V[n] = ZeroExtend(value);
    else
        assert width == 64;
        _V[n]<127:64> = value<63:0>;
```

### aarch64/functions/registers/X

```
// X[] - assignment form
// =====================
// Write to general-purpose register from either a 32-bit or a 64-bit value.

X[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value);
    return;

// X[] - non-assignment form
// =========================
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// ===========================

bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        when EL3  r = ELR_EL3;
        otherwise Unreachable();
    return r;

// ELR[] - non-assignment form
// ===========================

bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// =======================

ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1  ELR_EL1 = r;
        when EL2  ELR_EL2 = r;
        when EL3  ELR_EL3 = r;
        otherwise Unreachable();
    return;

// ELR[] - assignment form
// =======================

ELR[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR[PSTATE.EL] = value;
    return;
```

### aarch64/functions/sysregisters/ESR

```
// ESR[] - non-assignment form
// ===========================

ESRType ESR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        when EL3  r = ESR_EL3;
        otherwise Unreachable();
    return r;

// ESR[] - non-assignment form
// ===========================

ESRType ESR[]
    return ESR[S1TranslationRegime()];

// ESR[] - assignment form
// =======================

ESR[bits(2) regime] = ESRType value
    bits(32) r = value;
    case regime of
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        when EL3  ESR_EL3 = r;
        otherwise Unreachable();
    return;

// ESR[] - assignment form
// =======================

ESR[] = ESRType value
    ESR[S1TranslationRegime()] = value;
```

### aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

### aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// ===========================

bits(64) FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = FAR_EL1;
        when EL2  r = FAR_EL2;
        when EL3  r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR[] - non-assignment form
// ===========================

bits(64) FAR[]
    return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// =======================

FAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1  FAR_EL1 = r;
        when EL2  FAR_EL2 = r;
        when EL3  FAR_EL3 = r;
        otherwise Unreachable();
    return;

// FAR[] - assignment form
// =======================

FAR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;
```

### aarch64/functions/sysregisters/MAIR

```
// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = MAIR_EL1;
        when EL2  r = MAIR_EL2;
        when EL3  r = MAIR_EL3;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        otherwise Unreachable();
    return r;

// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[]
    return MAIR[S1TranslationRegime()];
```

### aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

### aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// =============================

SCTLRType SCTLR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        when EL3  r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// =============================

SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];
```

### aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

### aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
```

### aarch64/functions/system/AArch64.CheckAdvSIMDFPSystemRegisterTraps

```
(boolean, bits(2)) AArch64.CheckAdvSIMDFPSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn,
bits(4) crm, bits(3) op2, bit read);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/functions/system/AArch64.CheckSystemRegisterTraps

```
 (boolean, bits(2)) AArch64.CheckSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
bits(3) op2, bit read);
```

### aarch64/functions/system/AArch64.CheckUnallocatedSystemAccess

```
 boolean AArch64.CheckUnallocatedSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3)
op2, bit read);
```

### aarch64/functions/system/CheckSystemAccess

```
// CheckSystemAccess()
// ===================

CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bits(5) rt, bit read)
    // Checks if an AArch64 MSR/MRS/SYS instruction is UNALLOCATED or trapped at the current
    // exception level, security state and configuration, based on the opcode's encoding.
    boolean unallocated = FALSE;
    boolean need_secure = FALSE;
    bits(2) min_EL;

    // Check for traps by HCR_EL2.TIDCP
    if HaveEL(EL2) && !IsSecure() && HCR_EL2.TIDCP == 1 && op0 == 'x1' && crn == '1x11' then
        // At Non-secure EL0, it is IMPLEMENTATION_DEFINED whether attempts to execute system
        // register access instructions with reserved encodings are trapped to EL2 or UNDEFINED
        rcs_el0_trap = boolean IMPLEMENTATION_DEFINED "Reserved Control Space EL0 Trapped";
        if PSTATE.EL == EL0 && rcs_el0_trap then
            AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);
        elsif PSTATE.EL == EL1 then
            AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);

    // Check for unallocated encodings
    case op1 of
        when '00x', '010'
            min_EL = EL1;
        when '011'
            min_EL = EL0;
        when '100'
            min_EL = EL2;
        when '101'
            UnallocatedEncoding();
        when '110'
            min_EL = EL3;
        when '111'
            min_EL = EL1;
            need_secure = TRUE;
    if UInt(PSTATE.EL) < UInt(min_EL) then
        UnallocatedEncoding();
    elsif need_secure && !IsSecure() then
        UnallocatedEncoding();
    elsif AArch64.CheckUnallocatedSystemAccess(op0, op1, crn, crm, op2, read) then
        UnallocatedEncoding();

    // Check for traps on accesses to SIMD or floating-point registers
    (take_trap, target_el) = AArch64.CheckAdvSIMDFPSystemRegisterTraps(op0, op1, crn, crm, op2);
    if take_trap then
        AArch64.AdvSIMDFPAccessTrap(target_el);

    // Check for traps on access to all other system registers
    (take_trap, target_el) = AArch64.CheckSystemRegisterTraps(op0, op1, crn, crm, op2, read);
    if take_trap then
        AArch64.SystemRegisterTrap(target_el, op0, op2, op1, crn, rt, crm, read);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/functions/system/SysOp_R

```
bits(64) SysOp_R(integer op0, integer op1, integer crn, integer crm, integer op2);
```

### aarch64/functions/system/SysOp_W

```
SysOp_W(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

### aarch64/functions/system/System_Get

```
bits(64) System_Get(integer op0, integer op1, integer crn, integer crm, integer op2);
```

### aarch64/functions/system/System_Put

```
System_Put(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

## J1.1.4 aarch64/instrs

This section includes the following pseudocode functions:


- *aarch64/instrs/branch/eret/AArch64.ExceptionReturn*.
- *aarch64/instrs/countop/CountOp* on page J1-5251.
- *aarch64/instrs/extendreg/DecodeRegExtend* on page J1-5251.
- *aarch64/instrs/extendreg/ExtendReg* on page J1-5251.
- *aarch64/instrs/extendreg/ExtendType* on page J1-5252.
- *aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp* on page J1-5252.
- *aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp* on page J1-5252.
- *aarch64/instrs/float/convert/fpconvop/FPConvOp* on page J1-5252.
- *aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred* on page J1-5252.
- *aarch64/instrs/integer/bitmasks/DecodeBitMasks* on page J1-5253.
- *aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp* on page J1-5253.
- *aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred* on page J1-5253.
- *aarch64/instrs/integer/shiftreg/DecodeShift* on page J1-5254.
- *aarch64/instrs/integer/shiftreg/ShiftReg* on page J1-5254.
- *aarch64/instrs/integer/shiftreg/ShiftType* on page J1-5254.
- *aarch64/instrs/logicalop/LogicalOp* on page J1-5254.
- *aarch64/instrs/memory/memop/MemOp* on page J1-5254.
- *aarch64/instrs/memory/prefetch/Prefetch* on page J1-5254.
- *aarch64/instrs/system/barriers/barrierop/MemBarrierOp* on page J1-5255.
- *aarch64/instrs/system/hints/syshintop/SystemHintOp* on page J1-5255.
- *aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField* on page J1-5255.
- *aarch64/instrs/system/sysops/sysop/SysOp* on page J1-5255.
- *aarch64/instrs/system/sysops/sysop/SystemOp* on page J1-5256.
- *aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp* on page J1-5256.
- *aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp* on page J1-5256.
- *aarch64/instrs/vector/crypto/enabled/CheckCryptoEnabled64* on page J1-5256.
- *aarch64/instrs/vector/logical/immediateop/ImmediateOp* on page J1-5256.
- *aarch64/instrs/vector/reduce/reduceop/Reduce* on page J1-5256.
- *aarch64/instrs/vector/reduce/reduceop/ReduceOp* on page J1-5257.


### aarch64/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =========================
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AArch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    EventRegisterSet();

    if spsr<4> == '1' then
        // For an attempted to change to AArch32 state, align PC[1:0] according
        // to the target instruction set state. If the exception return is illegal,
        // it is IMPLEMENTATION DEFINED  whether this alignment takes place.
        align_pc = boolean IMPLEMENTATION_DEFINED "align PC on illegal exception return";
        if PSTATE.IL == '0' || align_pc then
            if spsr<5> == '1' then              // T32
                new_pc = Align(new_pc, 2);
            else                                // A32
                new_pc = Align(new_pc, 4);

        // If the return was illegal, the 32 MSBs of the target PC might be zeroed
        if PSTATE.IL == '1' && ConstrainUnpredictableBool() then
            new_pc<63:32> = Zeros();

    if UsingAArch32() then
        // 32 most significant bits are ignored
        BranchTo(new_pc<31:0>, BranchType_UNKNOWN);
    else
        BranchTo(new_pc, BranchType_ERET);
```

## aarch64/instrs/countop/CountOp

```
enumeration CountOp     {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

## aarch64/instrs/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =================
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```

## aarch64/instrs/extendreg/ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType type, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;

    case type of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SXTX unsigned = FALSE; len = 64;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when ExtendType_UXTB unsigned = TRUE;  len = 8;
        when ExtendType_UXTH unsigned = TRUE;  len = 16;
        when ExtendType_UXTW unsigned = TRUE;  len = 32;
        when ExtendType_UXTX unsigned = TRUE;  len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    //   [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    //   LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

### aarch64/instrs/extendreg/ExtendType

```
enumeration ExtendType  {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                         ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

### aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```
enumeration FPMaxMinOp  {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                         FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

### aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp

```
enumeration FPUnaryOp   {FPUnaryOp_ABS, FPUnaryOp_MOV,
                         FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

### aarch64/instrs/float/convert/fpconvop/FPConvOp

```
enumeration FPConvOp    {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                         FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF};
```

### aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// ==============
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // must be UBFX/SBFX alias
    return TRUE;
```

### aarch64/instrs/integer/bitmasks/DecodeBitMasks

```
// DecodeBitMasks()
// ================

// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)
    bits(M) tmask, wmask;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then ReservedValue();
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        ReservedValue();

    S = UInt(imms AND levels);
    R = UInt(immr AND levels);
    diff = S - R;    // 6-bit subtract with borrow

    esize = 1 << len;
    d = UInt(diff<len-1:0>);
    welem = ZeroExtend(Ones(S + 1), esize);
    telem = ZeroExtend(Ones(d + 1), esize);
    wmask = Replicate(ROR(welem, R));
    tmask = Replicate(telem);
    return (wmask, tmask);
```

### aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp

```
enumeration MoveWideOp  {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

### aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// ===================
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && immN:imms != '1xxxxxx' then
        return FALSE;
    if sf == '0' && immN:imms != '00xxxxx' then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if S < 16 then
        // ones must not span halfword boundary when rotated
        return (-R MOD 16) <= (15 - S);

    // for MOVN must contain no more than 16 zeros
    if S >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (R MOD 16) <= (S - (width - 15));

    return FALSE;
```

### aarch64/instrs/integer/shiftreg/DecodeShift

```
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00'  return ShiftType_LSL;
        when '01'  return ShiftType_LSR;
        when '10'  return ShiftType_ASR;
        when '11'  return ShiftType_ROR;
```

### aarch64/instrs/integer/shiftreg/ShiftReg

```
// ShiftReg()
// ==========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType type, integer amount)
    bits(N) result = X[reg];
    case type of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

### aarch64/instrs/integer/shiftreg/ShiftType

```
enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

### aarch64/instrs/logicalop/LogicalOp

```
enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

### aarch64/instrs/memory/memop/MemOp

```
enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

### aarch64/instrs/memory/prefetch/Prefetch

```
// Prefetch()
// ==========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    case prfop<4:3> of
        when '00' hint = Prefetch_READ;         // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;         // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;        // PST: prepare for store
        when '11' return;                       // unallocated hint
    target = UInt(prfop<2:1>);                  // target cache level
    stream = (prfop<0> != '0');                 // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

### aarch64/instrs/system/barriers/barrierop/MemBarrierOp

```
enumeration MemBarrierOp   {MemBarrierOp_DSB, MemBarrierOp_DMB, MemBarrierOp_ISB};
```

### aarch64/instrs/system/hints/syshintop/SystemHintOp

```
enumeration SystemHintOp {SystemHintOp_NOP, SystemHintOp_YIELD,
                          SystemHintOp_WFE, SystemHintOp_WFI,
                          SystemHintOp_SEV, SystemHintOp_SEVL};
```

### aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_SP};
```

### aarch64/instrs/system/sysops/sysop/SysOp

```
// SysOp()
// =======

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;   // S1E1R
        when '100 0111 1000 000' return Sys_AT;   // S1E2R
        when '110 0111 1000 000' return Sys_AT;   // S1E3R
        when '000 0111 1000 001' return Sys_AT;   // S1E1W
        when '100 0111 1000 001' return Sys_AT;   // S1E2W
        when '110 0111 1000 001' return Sys_AT;   // S1E3W
        when '000 0111 1000 010' return Sys_AT;   // S1E0R
        when '000 0111 1000 011' return Sys_AT;   // S1E0W
        when '100 0111 1000 100' return Sys_AT;   // S12E1R
        when '100 0111 1000 101' return Sys_AT;   // S12E1W
        when '100 0111 1000 110' return Sys_AT;   // S12E0R
        when '100 0111 1000 111' return Sys_AT;   // S12E0W
        when '011 0111 0100 001' return Sys_DC;   // ZVA
        when '000 0111 0110 001' return Sys_DC;   // IVAC
        when '000 0111 0110 010' return Sys_DC;   // ISW
        when '011 0111 1010 001' return Sys_DC;   // CVAC
        when '000 0111 1010 010' return Sys_DC;   // CSW
        when '011 0111 1011 001' return Sys_DC;   // CVAU
        when '011 0111 1110 001' return Sys_DC;   // CIVAC
        when '000 0111 1110 010' return Sys_DC;   // CISW
        when '000 0111 0001 000' return Sys_IC;   // IALLUIS
        when '000 0111 0101 000' return Sys_IC;   // IALLU
        when '011 0111 0101 001' return Sys_IC;   // IVAU
        when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
        when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
        when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
        when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
        when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
        when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
        when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
        when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
        when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
        when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
        when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
        when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
        when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
        when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
        when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
        when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
        when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
        when '100 1000 0111 000' return Sys_TLBI; // ALLE2
        when '110 1000 0111 000' return Sys_TLBI; // ALLE3
        when '000 1000 0111 001' return Sys_TLBI; // VAE1
        when '100 1000 0111 001' return Sys_TLBI; // VAE2
        when '110 1000 0111 001' return Sys_TLBI; // VAE3
        when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
        when '000 1000 0111 011' return Sys_TLBI; // VAAE1
        when '100 1000 0111 100' return Sys_TLBI; // ALLE1
        when '000 1000 0111 101' return Sys_TLBI; // VALE1
        when '100 1000 0111 101' return Sys_TLBI; // VALE2
        when '110 1000 0111 101' return Sys_TLBI; // VALE3
        when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
        when '000 1000 0111 111' return Sys_TLBI; // VAALE1
    return Sys_SYS;
```

### aarch64/instrs/system/sysops/sysop/SystemOp

```
enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```

### aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```
enumeration VBitOp      {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

### aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```
enumeration CompareOp   {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                         CompareOp_LE, CompareOp_LT};
```

### aarch64/instrs/vector/crypto/enabled/CheckCryptoEnabled64

```
// CheckCryptoEnabled64()
// ======================

CheckCryptoEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
    return;
```

### aarch64/instrs/vector/logical/immediateop/ImmediateOp

```
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                         ImmediateOp_ORR, ImmediateOp_BIC};
```

### aarch64/instrs/vector/reduce/reduceop/Reduce

```
// Reduce()
// ========

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return input;

half = N DIV 2;
hi = Reduce(op, input<N-1:half>, esize);
lo = Reduce(op, input<half-1:0>, esize);

case op of
    when ReduceOp_FMINNUM
        result = FPMinNum(lo, hi, FPCR);
    when ReduceOp_FMAXNUM
        result = FPMaxNum(lo, hi, FPCR);
    when ReduceOp_FMIN
        result = FPMin(lo, hi, FPCR);
    when ReduceOp_FMAX
        result = FPMax(lo, hi, FPCR);
    when ReduceOp_FADD
        result = FPAdd(lo, hi, FPCR);
    when ReduceOp_ADD
        result = lo + hi;

return result;
```

### aarch64/instrs/vector/reduce/reduceop/ReduceOp

```
enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                      ReduceOp_FMIN, ReduceOp_FMAX,
                      ReduceOp_FADD, ReduceOp_ADD};
```

## J1.1.5 aarch64/translation

This section includes the following pseudocode functions:


- *aarch64/translation/attrs/AArch64.InstructionDevice* on page J1-5258.
- *aarch64/translation/attrs/AArch64.S1AttrDecode* on page J1-5258.
- *aarch64/translation/attrs/AArch64.TranslateAddressS1Off* on page J1-5259.
- *aarch64/translation/checks/AArch64.CheckPermission* on page J1-5260.
- *aarch64/translation/checks/AArch64.CheckS2Permission* on page J1-5260.
- *aarch64/translation/debug/AArch64.CheckBreakpoint* on page J1-5261.
- *aarch64/translation/debug/AArch64.CheckDebug* on page J1-5261.
- *aarch64/translation/debug/AArch64.CheckWatchpoint* on page J1-5262.
- *aarch64/translation/faults/AArch64.AccessFlagFault* on page J1-5262.
- *aarch64/translation/faults/AArch64.AddressSizeFault* on page J1-5262.
- *aarch64/translation/faults/AArch64.AlignmentFault* on page J1-5262.
- *aarch64/translation/faults/AArch64.AsynchExternalAbort* on page J1-5263.
- *aarch64/translation/faults/AArch64.DebugFault* on page J1-5263.
- *aarch64/translation/faults/AArch64.NoFault* on page J1-5263.
- *aarch64/translation/faults/AArch64.PermissionFault* on page J1-5263.
- *aarch64/translation/faults/AArch64.TranslationFault* on page J1-5264.
- *aarch64/translation/translation/AArch64.FirstStageTranslate* on page J1-5264.
- *aarch64/translation/translation/AArch64.FullTranslate* on page J1-5265.
- *aarch64/translation/translation/AArch64.SecondStageTranslate* on page J1-5265.
- *aarch64/translation/translation/AArch64.SecondStageWalk* on page J1-5266.
- *aarch64/translation/translation/AArch64.TranslateAddress* on page J1-5266.
- *aarch64/translation/walk/AArch64.TranslationTableWalk* on page J1-5266.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/translation/attrs/AArch64.InstructionDevice

```
// AArch64.InstructionDevice()
// ===========================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch64.InstructionDevice(AddressDescriptor addrdesc, bits(64) vaddress,
                                            bits(48) ipaddress, integer level,
                                            AccType acctype, boolean iswrite, boolean secondstage,
                                            boolean s2fs1walk)

    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch64.PermissionFault(ipaddress, level, acctype, iswrite,
                                                 secondstage, s2fs1walk);
    else
        addrdesc.mattrs.type = MemType_Normal;
        addrdesc.mattrs.inner.attrs = MemAttr_NC;
        addrdesc.mattrs.inner.hints = MemHint_No;
        addrdesc.mattrs.outer = addrdesc.mattrs.inner;
        addrdesc.mattrs = MemAttrDefaults(addrdesc.mattrs);

    return addrdesc;
```

### aarch64/translation/attrs/AArch64.S1AttrDecode

```
// AArch64.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch64.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes mattrs;

    mair = MAIR[];
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();

    if attrfield<7:4> == '0000' then            // Device
        mattrs.type = MemType_Device;
        case attrfield<3:0> of
            when '0000'  mattrs.device = DeviceType_nGnRnE;
            when '0100'  mattrs.device = DeviceType_nGnRE;
            when '1000'  mattrs.device = DeviceType_nGRE;
            when '1100'  mattrs.device = DeviceType_GRE;
            otherwise    Unreachable();         // Reserved, handled above

    elsif attrfield<3:0> != '0000'  then        // Normal
        mattrs.type = MemType_Normal;
        mattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        mattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        mattrs.shareable = SH<1> == '1';
        mattrs.outershareable = SH == '10';

    else
        Unreachable();                          // Reserved, handled above

    return MemAttrDefaults(mattrs);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/translation/attrs/AArch64.TranslateAddressS1Off

```
// AArch64.TranslateAddressS1Off()
// ===============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch64.TranslateAddressS1Off(bits(64) vaddress, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    Top = AddrTop(vaddress);
    if !IsZero(vaddress<Top:PAMax()>) then
        level = 0;
        ipaddress = bits(48) UNKNOWN;
        secondstage = FALSE;
        s2fs1walk = FALSE;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    default_cacheable = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} && HCR_EL2.DC == '1');

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.type = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;      // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
    elsif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.type = MemType_Device;
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
    else
        // Instruction cacheability controlled by SCTLR_ELx.I
        cacheable = SCTLR[].I == '1';
        result.addrdesc.memattrs.type = MemType_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            result.addrdesc.memattrs.inner.hints = MemHint_No;
        result.addrdesc.memattrs.shareable = TRUE;
        result.addrdesc.memattrs.outershareable = TRUE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.physicaladdress = vaddress<47:0>;
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch64.NoFault();

    return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/translation/checks/AArch64.CheckPermission

```
// AArch64.CheckPermission()
// =========================
// Function used for permission checking from AArch64 stage 1 translations

FaultRecord AArch64.CheckPermission(Permissions perms, bits(64) vaddress, integer level,
                                    bit NS, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    wxn = SCTLR[].WXN == '1';

    if PSTATE.EL IN {EL0,EL1} then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';
        user_xn = perms.xn == '1' || (user_w && wxn);
        priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;
        ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2 or EL3
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
        xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(48) UNKNOWN;
        return AArch64.PermissionFault(ipaddress, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

### aarch64/translation/checks/AArch64.CheckS2Permission

```
// AArch64.CheckS2Permission()
// ===========================
// Function used for permission checking from AArch64 stage 2 translations

FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, bits(48) ipaddress,
                                      integer level, AccType acctype, boolean iswrite,
                                      boolean s2fs1walk)
    assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && PSTATE.EL != EL2;

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    xn = perms.xn == '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
    elsif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch64.PermissionFault(ipaddress, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

### aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    match = FALSE;

    for i = 0 to UInt(ID_AA64DFR0_EL1.BRPs)
        match_i = AArch64.BreakpointMatch(i, vaddress, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

### aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch64.NoFault();

    d_side = (acctype != AccType_IFETCH);
    generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            fault = AArch64.CheckBreakpoint(vaddress, size);

    return fault;
```

### aarch64/translation/debug/AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = PSTATE.EL != EL0 && !(PSTATE.EL == EL1 && acctype == AccType_UNPRIV);

    for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
        match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

### aarch64/translation/faults/AArch64.AccessFlagFault

```
// AArch64.AccessFlagFault()
// =========================

FaultRecord AArch64.AccessFlagFault(bits(48) ipaddress, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

### aarch64/translation/faults/AArch64.AddressSizeFault

```
// AArch64.AddressSizeFault()
// ==========================

FaultRecord AArch64.AddressSizeFault(bits(48) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

### aarch64/translation/faults/AArch64.AlignmentFault

```
// AArch64.AlignmentFault()
// ========================

FaultRecord AArch64.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    s2fs1walk = boolean UNKNOWN;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return AArch64.CreateFaultRecord(Fault_Alignment, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

### aarch64/translation/faults/AArch64.AsynchExternalAbort

```
// AArch64.AsynchExternalAbort()
// =============================
// Wrapper function for asynchronous external aborts

FaultRecord AArch64.AsynchExternalAbort(boolean parity, bit extflag)

    type = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(type, ipaddress,  level, acctype, iswrite, extflag,
                                     secondstage, s2fs1walk);
```

### aarch64/translation/faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// ====================

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)

    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_Debug, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

### aarch64/translation/faults/AArch64.NoFault

```
// AArch64.NoFault()
// =================

FaultRecord AArch64.NoFault()

    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_None, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

### aarch64/translation/faults/AArch64.PermissionFault

```
// AArch64.PermissionFault()
// =========================

FaultRecord AArch64.PermissionFault(bits(48) ipaddress, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Permission, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

### aarch64/translation/faults/AArch64.TranslationFault

```
// AArch64.TranslationFault()
// ==========================

FaultRecord AArch64.TranslationFault(bits(48) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Translation, ipaddress,  level, acctype, iswrite,
                                     extflag, secondstage, s2fs1walk);
```

### aarch64/translation/translation/AArch64.FirstStageTranslate

```
// AArch64.FirstStageTranslate()
// =============================
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.FirstStageTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                              boolean wasaligned, integer size)

    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';
    else
        s1_enabled = SCTLR[].M == '1';

    ipaddress = bits(48) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                          // First stage enabled
        S1 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);
        permissioncheck = TRUE;
    else
        S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;

    // Check for unaligned data accesses to Device memory
    if (!wasaligned && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
        acctype != AccType_IFETCH) then
        S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level,
                                                    S1.addrdesc.paddress.NS,
                                                    acctype, iswrite);

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
        acctype == AccType_IFETCH) then
        S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                                acctype, iswrite,
                                                secondstage, s2fs1walk);

    return S1.addrdesc;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch64/translation/translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch64.FullTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch64.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !IsFault(S1) && HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        s2fs1walk = FALSE;
        result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                              size);
    else
        result = S1;

    return result;
```

### aarch64/translation/translation/AArch64.SecondStageTranslate

```
// AArch64.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.SecondStageTranslate(AddressDescriptor S1, bits(64) vaddress,
                                               AccType acctype, boolean iswrite, boolean wasaligned,
                                               boolean s2fs1walk, integer size)
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};

    s2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';
    secondstage = TRUE;

    if s2_enabled then                          // Second stage enabled
        ipaddress = S1.paddress.physicaladdress<47:0>;
        S2 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if (!wasaligned && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype != AccType_IFETCH) then
            S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                          acctype, iswrite, s2fs1walk);

        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype == AccType_IFETCH) then
            S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    acctype, iswrite,
                                                    secondstage, s2fs1walk);

        // Check for protected table walk
        if (s2fs1walk && !IsFault(S2.addrdesc) && HCR_EL2.PTW == '1' &&
            S2.addrdesc.memattrs.type == MemType_Device) then
            S2.addrdesc.fault = AArch64.PermissionFault(ipaddress, S2.level, acctype,
                                                        iswrite, secondstage, s2fs1walk);
        result = CombineS1S2Desc(S1, S2.addrdesc);
    else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result = S1;

    return result;
```

### aarch64/translation/translation/AArch64.SecondStageWalk

```
// AArch64.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype,
                                          boolean iswrite, integer size)
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                        size);
```

### aarch64/translation/translation/AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)

    result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
```

### aarch64/translation/walk/AArch64.TranslationTableWalk

```
// AArch64.TranslationTableWalk()
// ==============================
// Returns a result of a translation table walk
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch64.TranslationTableWalk(bits(48) ipaddress, bits(64) vaddress,
                                       AccType acctype, boolean iswrite, boolean secondstage,
                                       boolean s2fs1walk, integer size)
    if !secondstage then
        assert !ELUsingAArch32(S1TranslationRegime());
    else
        assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && PSTATE.EL != EL2;

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(64) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2

    descaddr.memattrs.type = MemType_Normal;

    // Derived parameters for the page table walk:
    //  grainsize = Log2(Size of Table)         - Size of Table is 4KB, 16KB or 64KB in AArch64
    //  stride = Log2(Address per Level)        - Bits of address consumed at each level
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
//  firstblocklevel = First level where a block entry is allowed
//  ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTCR_EL2.PS
//  inputsize = Log2(Size of Input Address) - Input Address size in bits
//  level = Level to start walk from
// This means that the number of levels after start level = 3-level

if !secondstage then
    // First stage translation
    inputaddr = ZeroExtend(vaddress);
    top = AddrTop(inputaddr);

    if PSTATE.EL == EL3 then
        inputsize = 64 - UInt(TCR_EL3.T0SZ);
        if inputsize > 48 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 48;
        if inputsize < 25 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 25;
        largegrain = TCR_EL3.TG0 == '01';
        midgrain = TCR_EL3.TG0 == '10';
        ps = TCR_EL3.PS;
        basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<top:inputsize>);
        disabled = FALSE;
        baseregister = TTBR0_EL3;
        descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0);
        reversedescriptors = SCTLR_EL3.EE == '1';
        lookupsecure = TRUE;
        singlepriv = TRUE;

    elsif PSTATE.EL == EL2 then
        inputsize = 64 - UInt(TCR_EL2.T0SZ);
        if inputsize > 48 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 48;
        if inputsize < 25 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 25;
        largegrain = TCR_EL2.TG0 == '01';
        midgrain = TCR_EL2.TG0 == '10';
        ps = TCR_EL2.PS;
        basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<top:inputsize>);
        disabled = FALSE;
        baseregister = TTBR0_EL2;
        descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0);
        reversedescriptors = SCTLR_EL2.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;

    else
        if inputaddr<top> == '0' then
            inputsize = 64 - UInt(TCR_EL1.T0SZ);
            if inputsize > 48 then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = 48;
            if inputsize < 25 then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = 25;
            largegrain = TCR_EL1.TG0 == '01';
            midgrain = TCR_EL1.TG0 == '10';
            basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<top:inputsize>);
            disabled = TCR_EL1.EPD0 == '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                baseregister = TTBR0_EL1;
                descaddr.mematTrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORGN0, TCR_EL1.IRGN0);
            else
                inputsize = 64 - UInt(TCR_EL1.T1SZ);
                if inputsize > 48 then
                    c = ConstrainUnpredictable();
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
                    if c == Constraint_FORCE then inputsize = 48;
                if inputsize < 25 then
                    c = ConstrainUnpredictable();
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
                    if c == Constraint_FORCE then inputsize = 25;
                largegrain = TCR_EL1.TG1 == '11';       // TG1 and TG0 encodings differ
                midgrain = TCR_EL1.TG1 == '01';
                basefound = inputsize >= 25 && inputsize <= 48 && IsOnes(inputaddr<top:inputsize>);
                disabled = TCR_EL1.EPD1 == '1';
                baseregister = TTBR1_EL1;
                descaddr.mematTrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGN1, TCR_EL1.IRGN1);
            ps = TCR_EL1.IPS;
            reversedescriptors = SCTLR_EL1.EE == '1';
            lookupsecure = IsSecure();
            singlepriv = FALSE;

        if largegrain then
            grainsize = 16;                             // Log2(64KB page size)
            firstblocklevel = 2;                        // Largest block is 512MB (2^29 bytes)
        elsif midgrain then
            grainsize = 14;                             // Log2(16KB page size)
            firstblocklevel = 2;                        // Largest block is 32MB (2^25 bytes)
        else // Small grain
            grainsize = 12;                             // Log2(4KB page size)
            firstblocklevel = 1;                        // Largest block is 1GB (2^30 bytes)
        stride = grainsize - 3;                         // Log2(page size / 8 bytes)
        // The starting level is the number of strides needed to consume the input address
        level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));

    else
        // Second stage translation
        inputaddr = ZeroExtend(ipaddress);
        inputsize = 64 - UInt(VTCR_EL2.T0SZ);
        if inputsize > 48 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 48;
        if inputsize < 25 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = 25;
        largegrain = VTCR_EL2.TG0 == '01';
        midgrain = VTCR_EL2.TG0 == '10';
        ps = VTCR_EL2.PS;
        basefound = inputsize >= 25 && inputsize <= 48 && IsZero(inputaddr<63:inputsize>);
        disabled = FALSE;
        baseregister = VTTBR_EL2;
        descaddr.mematTrs = WalkAttrDecode(VTCR_EL2.IRGN0, VTCR_EL2.ORGN0, VTCR_EL2.SH0);
        reversedescriptors = SCTLR_EL2.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;

        startlevel = UInt(VTCR_EL2.SL0);
        if largegrain then
            grainsize = 16;                             // Log2(64KB page size)
            level = 3 - startlevel;
            firstblocklevel = 2;                        // Largest block is 512MB (2^29 bytes)
        elsif midgrain then
            grainsize = 14;                             // Log2(16KB page size)
            level = 3 - startlevel;
            firstblocklevel = 2;                        // Largest block is 32MB (2^25 bytes)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else // Small grain
        grainsize = 12;                             // Log2(4KB page size)
        level = 2 - startlevel;
        firstblocklevel = 1;                        // Largest block is 1GB (2^30 bytes)
    stride = grainsize - 3;                         // Log2(page size / 8 bytes)

    // Limits on IPA controls based on implemented PA size. Level 0 is only
    // supported by small grain translations
    if largegrain then                              // 64KB pages
        // Level 1 only supported if implemented PA size is greater than 2^42 bytes
        if level == 0 || (level == 1 && PAMax() <= 42) then basefound = FALSE;
    elsif midgrain then                             // 16KB pages
        // Level 1 only supported if implemented PA size is greater than 2^40 bytes
        if level == 0 || (level == 1 && PAMax() <= 40) then basefound = FALSE;
    else                                            // Small grain, 4KB pages
        // Level 0 only supported if implemented PA size is greater than 2^42 bytes
        if level < 0 || (level == 0 && PAMax() <= 42) then basefound = FALSE;

    // If the inputsize exceeds the PAMax value, the behavior is CONSTRAINED UNPREDICTABLE
    inputsizecheck = inputsize;
    if inputsize > PAMax() && (!ELUsingAArch32(EL1) || inputsize > 40) then
        case ConstrainUnpredictable() of
            when Constraint_FORCE
                // Restrict the inputsize to the PAMax value
                inputsize = PAMax();
                inputsizecheck = PAMax();
            when Constraint_FORCENOSLCHECK
                // As FORCE, except use the configured inputsize in the size checks below
                inputsize = PAMax();
            when Constraint_FAULT
                // Generate a translation fault
                basefound = FALSE;
            otherwise
                Unreachable();

    // Number of entries in the starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    startsizecheck = inputsizecheck - ((3 - level)*stride + grainsize); // Log2(Num of entries)

    // Check for starting level table with fewer than 2 entries or longer than 16 pages.
    // Lower bound check is:  startsizecheck < Log2(2 entries)
    // Upper bound check is:  startsizecheck > Log2(pagesize/8*16)
    if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

if !basefound || disabled then
    level = 0;           // AArch32 reports this as a level 1 fault
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

case ps of
    when '000'  outputsize = 32;
    when '001'  outputsize = 36;
    when '010'  outputsize = 40;
    when '011'  outputsize = 42;
    when '100'  outputsize = 44;
    when '101'  outputsize = 48;
    otherwise   outputsize = 48;

if outputsize > PAMax() then outputsize = PAMax();

if outputsize != 48 && !IsZero(baseregister<47:outputsize>) then
    level = 0;
    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

// Bottom bound of the Base address is:
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
//     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
//     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
baseaddress = baseregister<47:baselowerbound>:Zeros(baselowerbound);

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

addrselecttop = inputsize - 1;

repeat
    addrselectbottom = (3-level)*stride + grainsize;

    bits(48) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
    descaddr.paddress.physicaladdress = baseaddress OR index;
    descaddr.paddress.NS = ns_table;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if !HaveEL(EL2) || secondstage || IsSecure() || PSTATE.EL == EL2 then
        descaddr2 = descaddr;
    else
        descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
        // Check for a fault on the stage 2 walk
        if IsFault(descaddr2) then
            result.addrdesc.fault = descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    descaddr2.vaddress = ZeroExtend(vaddress);

    desc = _Mem[descaddr2, 8, AccType_PTW];
    if reversedescriptors then desc = BigEndianReverse(desc);

    if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
         // Fault (00), Reserved (10), or Block (01) at level 3
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Valid Block, Page, or Table entry
    if desc<1:0> == '01' || level == 3 then                 // Block (01) or Page (11)
        blocktranslate = TRUE;
    else                                                    // Table (11)
        if outputsize != 48 && !IsZero(desc<47:outputsize>) then
            result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        baseaddress = desc<47:grainsize>:Zeros(grainsize);

        if !secondstage then
            // Unpack the upper and lower table attributes
            ns_table    = ns_table    OR desc<63>;
            ap_table<1> = ap_table<1> OR desc<62>;       // read-only
            xn_table    = xn_table    OR desc<60>;
            // pxn_table and ap_table[0] apply only in EL1&0 translation regimes
            if !singlepriv then
                ap_table<0> = ap_table<0> OR desc<61>;   // privileged
                pxn_table   = pxn_table   OR desc<59>;

        level = level + 1;
        addrselecttop = addrselectbottom - 1;
        blocktranslate = FALSE;
until blocktranslate;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// Check block size is supported at this level
if level < firstblocklevel then
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                     iswrite, secondstage, s2fs1walk);
    return result;

// Check for misprogramming of the contiguous bit
if largegrain then
    contiguousbitcheck = level == 2 && inputsize < 34;
elsif midgrain then
    contiguousbitcheck = level == 2 && inputsize < 30;
else
    contiguousbitcheck = level == 1 && inputsize < 34;

if contiguousbitcheck && desc<52> == '1' then
    if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

// Check the output address is inside the supported range
if outputsize != 48 && !IsZero(desc<47:outputsize>) then
    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                     iswrite, secondstage, s2fs1walk);
    return result;

// Check the access flag
if desc<10> == '0' then
    result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);
    return result;

// Unpack the descriptor into address and upper and lower block attributes
outputaddress = desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
xn = desc<54>;
pxn = desc<53>;
contiguousbit = desc<52>;
nG = desc<11>;
sh = desc<9:8>;
ap = desc<7:6>:'1';
mematttr = desc<5:2>;                                   // AttrIndx and NS bit in stage 1

result.domain = bits(4) UNKNOWN;                        // Domains not used
result.level = level;
result.blocksize = 2^((3-level)*stride + grainsize);

// Stage 1 translation regimes also inherit attributes from the tables
if !secondstage then
    result.perms.xn      = xn OR xn_table;
    result.perms.ap<2>   = ap<2> OR ap_table<1>;        // Force read-only

    // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
    if !singlepriv then
        result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
        result.perms.pxn   = pxn OR pxn_table;
        // Pages from Non-secure tables are marked non-global in Secure EL1&0
        if IsSecure() then
            result.nG = nG OR ns_table;
        else
            result.nG = nG;
    else
        result.perms.ap<1> = '1';
        result.perms.pxn   = '0';
        result.nG          = '0';
    result.perms.ap<0>   = '1';
    result.addrdesc.mematttrs = AArch64.S1AttrDecode(sh, mematttr<2:0>, acctype);
    result.addrdesc.paddress.NS = mematttr<3> OR ns_table;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
else
    result.perms.ap<2:1> = ap<2:1>;
    result.perms.ap<0>   = '1';
    result.perms.xn      = xn;
    result.perms.pxn     = '0';
    result.nG            = '0';
    result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
    result.addrdesc.paddress.NS = '1';

result.addrdesc.paddress.physicaladdress = outputaddress;
result.addrdesc.fault = AArch64.NoFault();
result.contiguous = contiguousbit == '1';

return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## J1.2 Pseudocode for AArch32 operation

This section holds the pseudocode for execution in AArch32 state. Functions listed in this section are identified as `AArch32.FunctionName`. Some of these functions have an equivalent AArch64 function, `AArch64.FunctionName`. This section is organised by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch32/debug/breakpoint`.

The top-level sections of the AArch32 pseudocode hierarchy are:

- *aarch32/debug*.
- *aarch32/exceptions* on page J1-5281.
- *aarch32/functions* on page J1-5299.
- *aarch32/translation* on page J1-5324.

### J1.2.1 aarch32/debug

This section includes the following pseudocode functions:

- *aarch32/debug/VCRMatch/AArch32.VCRMatch*.
- *aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled* on page J1-5274.
- *aarch32/debug/breakpoint/AArch32.BreakpointMatch* on page J1-5274.
- *aarch32/debug/breakpoint/AArch32.BreakpointValueMatch* on page J1-5275.
- *aarch32/debug/breakpoint/AArch32.StateMatch* on page J1-5276.
- *aarch32/debug/enables/AArch32.GenerateDebugExceptions* on page J1-5277.
- *aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom* on page J1-5277.
- *aarch32/debug/pmu/AArch32.CheckForPMUOverflow* on page J1-5277.
- *aarch32/debug/pmu/AArch32.CountEvents* on page J1-5278.
- *aarch32/debug/pmu/AArch32.ProfilingProhibited* on page J1-5278.
- *aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState* on page J1-5279.
- *aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState* on page J1-5279.
- *aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState* on page J1-5280.
- *aarch32/debug/watchpoint/AArch32.WatchpointByteMatch* on page J1-5280.
- *aarch32/debug/watchpoint/AArch32.WatchpointMatch* on page J1-5281.

#### aarch32/debug/VCRMatch/AArch32.VCRMatch

```
// AArch32.VCRMatch()
// ==================

boolean AArch32.VCRMatch(bits(32) vaddress)

    if UsingAArch32() && ELUsingAArch32(EL1) && IsZero(vaddress<1:0>) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && !IsSecure() then
                match_word<UInt(vaddress<4:2>) + 24> = '1';     // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1';      // Secure vectors (or no EL3)
        if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() && vaddress<31:5> == MVBAR<31:5> then
            match_word<UInt(vaddress<4:2>) + 8> = '1';          // Monitor vectors

        // Mask out bits not corresponding to vectors.
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
        elsif !ELUsingAArch32(EL3) then
            mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
        else
            mask = '11011110':'00000000':'11011100':'11011110';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        match_word = match_word AND DBGVCR AND mask;
        match = !IsZero(match_word);

        // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
        if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
            match = ConstrainUnpredictableBool();
    else
        match = FALSE;

    return match;
```

## aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// ========================================================

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // In the recommended interface, SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;
```

## aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIDR.BRPs);

    enabled = DBGBCR[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then                  // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool();
        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool();

    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool();
        if !value_mismatch then value_mismatch = ConstrainUnpredictableBool();

    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## aarch32/debug/breakpoint/AArch32.BreakpointValueMatch

```
// AArch32.BreakpointValueMatch()
// ==============================
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n, bits(32) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //    matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.

    // If a non-existant breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(DBGDIDR.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(DBGDIDR.BRPs));
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking.)
    if DBGBCR[n].E == '0' then return (FALSE,FALSE);

    context_aware = (n >= UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    type = DBGBCR[n].BT;
    if (type IN {'011x','11xx'} ||                                  // Reserved
          (type == '010x' && HaltOnBreakpointOrWatchpoint()) ||     // Address mismatch
          (type != '0x0x' && !context_aware) ||                     // Context matching
          (type == '1xxx' && !HaveEL(EL2))) then                    // VMID match
        (c, type) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = type == '0x0x';
    mismatch   = type == '010x';
    match_vmid = type == '10xx';
    match_cid  = type == 'x01x';
    linked     = type == 'xxx1';

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return (FALSE,FALSE);

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return (FALSE,FALSE);

    // Do the comparison.
    if match_addr then
        byte = UInt(vaddress<1:0>);
        assert byte IN {0,2};                     // "vaddress" is halfword aligned.
        byte_select_match = (DBGBCR[n].BAS<byte> == '1');
        BVR_match = vaddress<31:2> == DBGBVR[n]<31:2> && byte_select_match;
    elsif match_cid then
        BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
    if match_vmid then
        vmid = (if ELUsingAArch32(EL2) then VTTBR.VMID else VTTBR_EL2.VMID);
        BXVR_match = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                      vmid == DBGBXVR[n]<7:0>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    match = (!match_vmid || BXVR_match) && (!(match_addr || match_cid) || BVR_match);
    return (match && !mismatch, !match && mismatch);
```

## aarch32/debug/breakpoint/AArch32.StateMatch

```
// AArch32.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreakpnt, boolean ispriv)
    // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
    // "linked" is TRUE if this is a linked breakpoint/watchpoint type.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.

    // If parameters are set to a reserved type, behaves either as disabled or not a reserved-type
    if ((HMC:SSC:PxC) IN {'011xx','100x0','101x0','110x0','11101','1111x'} ||         // Reserved
          (HMC == '0' && PxC == '00' && !isbreakpnt) ||                               // Usr/Svc/Sys
          (SSC IN {'01','10'} && !HaveEL(EL3)) ||                                     // No EL3
          (HMC:SSC != '000' && !HaveEL(EL3) && !HaveEL(EL2))) then                    // No EL3/EL2
        (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits();
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    PL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
    PL2_match = HaveEL(EL2) && HMC == '1';
    PL1_match = PxC<0> == '1';
    PL0_match = PxC<1> == '1';
    SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';

    if SSU_match then
        priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
    else
        case PSTATE.EL of
            when EL3, EL1  priv_match = if ispriv then PL1_match else PL0_match;
            when EL2       priv_match = PL2_match;
            when EL0       priv_match = PL0_match;

    case SSC of
        when '00'  security_state_match = TRUE;         // Both
        when '01'  security_state_match = !IsSecure();  // Non-secure only
        when '10'  security_state_match = IsSecure();   // Secure only
        when '11'  security_state_match = TRUE;         // Both

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));
        last_ctx_cmp = UInt(DBGDIDR.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(32) UNKNOWN;
        linked_to = TRUE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

### aarch32/debug/enables/AArch32.GenerateDebugExceptions

```
// AArch32.GenerateDebugExceptions()
// =================================

boolean AArch32.GenerateDebugExceptions()
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());
```

### aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```
// AArch32.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)

    if from == EL0 && !ELStateUsingAArch32(EL1, secure) then
        mask = bit UNKNOWN;                          // PSTATE.D mask, unused for EL0 case
        return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    if HaveEL(EL3) && secure then
        spd = (if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32);
        if spd<1> == '1' then
            enabled = spd<0> == '1';
        else
            // SPD == 0b01 is reserved, but behaves the same as 0b00.
            enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from != EL2;

    return enabled;
```

### aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```
// AArch32.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch32.CheckForPMUOverflow()

    if !ELUsingAArch32(EL1) then return AArch64.CheckForPMUOverflow();

    pmuirq = (PMCR.E == '1' && PMINTENSET<31> == '1' && PMOVSSET<31> == '1');
    for n = 0 to UInt(PMCR.N) - 1
        if HaveEL(EL2) then
            hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
            hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
            E = (if n < UInt(hpmn) then PMCR.E else hpme);
        else
            E = PMCR.E;
        if E == '1' && PMINTENSET<n> == '1' && PMOVSSET<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

    return pmuirq;
```

## aarch32/debug/pmu/AArch32.CountEvents

```
// AArch32.CountEvents()
// =====================
// Return TRUE if counter "n" should count its event.

boolean AArch32.CountEvents(integer n)
    assert(n == 31 || n < UInt(PMCR.N));

    if !ELUsingAArch32(EL1) then return AArch64.CountEvents(n);

    // Event counting is disabled in Debug state
    debug = Halted();

    // Event counting might be prohibited
    prohibited = AArch32.ProfilingProhibited(IsSecure(), PSTATE.EL);
    if prohibited && n == 31 && PMCR_EL0.DP == '0' then prohibited = FALSE;

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
    filter = (if n == 31 then PMCCFILTR<31:27> else PMEVTYPER[n]<31:27>);

    H = if !HaveEL(EL2) then '0' else filter<0>;
    P = filter<4>;  U = filter<3>;
    if !IsSecure() && HaveEL(EL3) then
        P = P EOR filter<2>;  U = U EOR filter<1>;

    case PSTATE.EL of
        when EL0      filtered = U == '1';
        when EL1,EL3  filtered = P == '1';
        when EL2      filtered = H == '0';

    if HaveEL(EL2) then
        hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
        hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
        E = (if n < UInt(hpmn) then PMCR.E else hpme);
    else
        E = PMCR.E;
    enabled = (E == '1' && PMCNTENSET<n> == '1');

    return !debug && !prohibited && !filtered && enabled;
```

## aarch32/debug/pmu/AArch32.ProfilingProhibited

```
// AArch32.ProfilingProhibited()
// =============================
// Determine whether event counting is prohibited in the current state.

boolean AArch32.ProfilingProhibited(boolean secure, bits(2) el)

    if (el == EL0 && !ELUsingAArch32(EL1)) || !ELUsingAArch32(el) then
        return AArch64.ProfilingProhibited(secure, el);

    // Events are always counted in Non-secure state.
    if !secure then return FALSE;

    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    if !HaveEL(EL3) then return FALSE;

    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    // * EL3 is using AArch32 and SDCR.SPME == 1
    spme = (if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if spme == '1' then return FALSE;

    // * Allowed by the IMPLEMENTATION DEFINED authentication interface
    if ExternalSecureNoninvasiveDebugEnabled() then return FALSE;

    // * EL3 or EL1 is using AArch32, executing at EL0, and SDER.SUNIDEN == 1.
    if el == EL0 && ELUsingAArch32(EL1) && SDER.SUNIDEN == '1' then return FALSE;

    return TRUE;
```

### aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// ==================================
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    EDSCR.ERR = '1';
    UpdateEDSCRFields();
    EndOfInstruction();
```

### aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// ===============================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.
    EndOfInstruction();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// ======================================
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.
    EndOfInstruction();
```

### aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =============================

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)

    bottom = if DBGWVR[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool();
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));  MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                    // Not contiguous
            byte_select_match = ConstrainUnpredictableBool();
            bottom = 3;                                       // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;           // Disabled
            when Constraint_NONE      mask = 0;               // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<31:mask> == DBGWVR[n]<31:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool();
    else
        WVR_match = vaddress<31:bottom> == DBGWVR[n]<31:bottom>;

    return WVR_match && byte_select_match;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
                                boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIDR.WRPs);

    // "ispriv" is FALSE for LDRT/STRT instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, DBGWCR[n].LBN, isbreakpnt, ispriv);

    ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

## J1.2.2 aarch32/exceptions

This section includes the following pseudocode functions:


- *aarch32/exceptions/aborts/AArch32.Abort* on page J1-5282.
- *aarch32/exceptions/aborts/AArch32.AbortSyndrome* on page J1-5282.
- *aarch32/exceptions/aborts/AArch32.CheckPCAlignment* on page J1-5283.
- *aarch32/exceptions/aborts/AArch32.ReportDataAbort* on page J1-5283.
- *aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort* on page J1-5284.
- *aarch32/exceptions/aborts/AArch32.TakeDataAbortException* on page J1-5284.
- *aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException* on page J1-5284.
- *aarch32/exceptions/asynch/AArch32.TakePhysicalAsynchAbortException* on page J1-5285.
- *aarch32/exceptions/asynch/AArch32.TakePhysicalFIQException* on page J1-5286.
- *aarch32/exceptions/asynch/AArch32.TakePhysicalIRQException* on page J1-5286.
- *aarch32/exceptions/asynch/AArch32.TakeVirtualAsynchAbortException* on page J1-5287.
- *aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException* on page J1-5287.
- *aarch32/exceptions/asynch/AArch32.TakeVirtualIRQException* on page J1-5287.
- *aarch32/exceptions/debug/AArch32.SoftwareBreakpoint* on page J1-5288.
- *aarch32/exceptions/debug/DebugException* on page J1-5288.
- *aarch32/exceptions/exceptions/AArch32.ExceptionClass* on page J1-5288.
- *aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64* on page J1-5289.
- *aarch32/exceptions/exceptions/AArch32.ReportHypEntry* on page J1-5289.
- *aarch32/exceptions/exceptions/AArch32.ResetControlRegisters* on page J1-5289.
- *aarch32/exceptions/exceptions/AArch32.TakeReset* on page J1-5289.
- *aarch32/exceptions/exceptions/ExcVectorBase* on page J1-5290.
- *aarch32/exceptions/ieeefp/AArch32.FPTrappedException* on page J1-5290.
- *aarch32/exceptions/syscalls/AArch32.CallHypervisor* on page J1-5291.
- *aarch32/exceptions/syscalls/AArch32.CallSupervisor* on page J1-5291.
- *aarch32/exceptions/syscalls/AArch32.TakeHVCException* on page J1-5291.
- *aarch32/exceptions/syscalls/AArch32.TakeSMCException* on page J1-5291.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *aarch32/exceptions/syscalls/AArch32.TakeSVCException* on page J1-5292.
- *aarch32/exceptions/takeexception/AArch32.EnterHypMode* on page J1-5292.
- *aarch32/exceptions/takeexception/AArch32.EnterMode* on page J1-5293.
- *aarch32/exceptions/takeexception/AArch32.EnterMonitorMode* on page J1-5293.
- *aarch32/exceptions/traps/AArch32.CPRegTrap* on page J1-5293.
- *aarch32/exceptions/traps/AArch32.CPRegTrapSyndrome* on page J1-5294.
- *aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEnabled* on page J1-5295.
- *aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap* on page J1-5295.
- *aarch32/exceptions/traps/AArch32.CheckForSMCTrap* on page J1-5296.
- *aarch32/exceptions/traps/AArch32.CheckForWFxTrap* on page J1-5296.
- *aarch32/exceptions/traps/AArch32.CheckITEnabled* on page J1-5296.
- *aarch32/exceptions/traps/AArch32.CheckIllegalState* on page J1-5297.
- *aarch32/exceptions/traps/AArch32.CheckSETENDEnabled* on page J1-5297.
- *aarch32/exceptions/traps/AArch32.TakeHypTrapException* on page J1-5298.
- *aarch32/exceptions/traps/AArch32.TakeMonitorTrapException* on page J1-5298.
- *aarch32/exceptions/traps/AArch32.TakeUndefInstrException* on page J1-5298.
- *aarch32/exceptions/traps/AArch32.UndefinedFault* on page J1-5298.


### aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress), fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
```

### aarch32/exceptions/aborts/AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort  exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception type, FaultRecord fault, bits(32) vaddress)

    exception = ExceptionSyndrome(type);

    d_side = type == Exception_DataAbort;

    exception.syndrome = FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.ipaddress = ZeroExtend(fault.ipaddress);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        exception.ipavalid = FALSE;

    return exception;
```

### aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// ==========================

AArch32.CheckPCAlignment()

    bits(32) pc = ThisInstrAddr();
    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

        // Generate an Alignment fault Prefetch Abort exception
        vaddress = pc;
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        secondstage = FALSE;
        AArch32.Abort(vaddress, AArch32.AlignmentFault(acctype, iswrite, secondstage));
```

### aarch32/exceptions/aborts/AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

    // If "route_to_monitor" is TRUE then use the Secure Banked system registers because EL3 is
    // implemented and using AArch32, and the Abort might be taken from Non-secure state.
    // Otherwise use the system registers for the current security state, which might not be banked.
    eae = (if route_to_monitor then TTBCR_S.EAE else TTBCR.EAE);
    d_side = TRUE;
    if eae == '1' then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType_IC then
        if (eae == '0' &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// =============================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

    // If "route_to_monitor" is TRUE then use the Secure Banked system registers because EL3 is
    // implemented and using AArch32, and the Abort might be taken from Non-secure state.
    // Otherwise use the system registers for the current security state, which might not be banked.
    eae = (if route_to_monitor then TTBCR_S.EAE else TTBCR.EAE);
    d_side = FALSE;
    if eae == '1' then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;
```

### aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// ================================

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || IsSecondStage(fault) ||
                     (IsDebugException(fault) && HDCR.TDE == '1')));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;

    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

### aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// ====================================

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || IsSecondStage(fault) ||
                     (IsDebugException(fault) && HDCR.TDE == '1')));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0C;
    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;

    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        if fault.type == Fault_Alignment then             // PC Alignment fault
            exception = ExceptionSyndrome(Exception_PCAlignment);
            exception.vaddress = ThisInstrAddr();
        else
            exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## aarch32/exceptions/asynch/AArch32.TakePhysicalAsynchAbortException

```
// AArch32.TakePhysicalAsynchAbortException()
// ==========================================

AArch32.TakePhysicalAsynchAbortException(boolean parity, bit extflag,
                                         boolean syndrome_valid, bits(24) full_syndrome)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSystemErrorException(syndrome_valid, full_syndrome);

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || HCR.AMO == '1'));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    fault = AArch32.AsynchExternalAbort(parity, extflag);
    vaddress = bits(32) UNKNOWN;

    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

### aarch32/exceptions/asynch/AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// ==================================

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1';

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalFIQException();

    route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || HCR.FMO == '1'));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;

    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_FIQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

### aarch32/exceptions/asynch/AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1';

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || HCR.IMO == '1'));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;

    if route_to_monitor then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_IRQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

### aarch32/exceptions/asynch/AArch32.TakeVirtualAsynchAbortException

```
// AArch32.TakeVirtualAsynchAbortException()
// =========================================

AArch32.TakeVirtualAsynchAbortException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    if ELUsingAArch32(EL2) then   // Virtual Asynchronous Abort enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualSystemErrorException();

    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    extflag = bit IMPLEMENTATION_DEFINED "Virtual Asynchronous Abort ExT bit";
    fault = AArch32.AsynchExternalAbort(parity, extflag);

    if ELUsingAArch32(EL2) then HCR.VA = '0'; else HCR_EL2.VSE = '0';

    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

### aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// =================================

AArch32.TakeVirtualFIQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    if ELUsingAArch32(EL2) then  // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR.TGE == '0' && HCR.FMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

### aarch32/exceptions/asynch/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// =================================

AArch32.TakeVirtualIRQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if ELUsingAArch32(EL2) then   // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

## aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// ============================

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);

    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH;           // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;

    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```

## aarch32/exceptions/debug/DebugException

```
constant bits(4) DebugException_Breakpoint  = '0001';
constant bits(4) DebugException_BKPT        = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint  = '1010';
```

## aarch32/exceptions/exceptions/AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// ========================
// Return the Exception Class and Instruction Length fields for reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception type)

    il = if ThisInstrLength() == 32 then '1' else '0';

    case type of
        when Exception_Uncategorized        ec = 0x00; il = '1';
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03;
        when Exception_CP15RRTTrap          ec = 0x04;
        when Exception_CP14RTTrap           ec = 0x05;
        when Exception_CP14DTTrap           ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap  ec = 0x07;
        when Exception_FPIDTrap             ec = 0x08;
        when Exception_CP14RRTTrap          ec = 0x0C;
        when Exception_IllegalState         ec = 0x0E; il = '1';
        when Exception_SupervisorCall       ec = 0x11;
        when Exception_HypervisorCall       ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_InstructionAbort     ec = 0x20; il = '1';
        when Exception_PCAlignment          ec = 0x22; il = '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when Exception_DataAbort            ec = 0x24;
        when Exception_FPTrappedException   ec = 0x28;
        otherwise                           Unreachable();

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;

    return (ec,il);
```

### aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

```
// AArch32.GeneralExceptionsToAArch64()
// ====================================
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
```

### aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception type = exception.type;

    (ec,il) = AArch32.ExceptionClass(type);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if type IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif type == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

### aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

```
AArch32.ResetControlRegisters(boolean cold_reset);
```

### aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// ===================
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    assert HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                     // Secure state
    elsif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset the CP14 and CP15 registers and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111';       // All asynchronous exceptions masked
    PSTATE.IT = '00000000';       // IT block state reset
    PSTATE.T = SCTLR.TE;          // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
    PSTATE.E = SCTLR.EE;          // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0';              // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv;                      // IMPLEMENTATION DEFINED reset vector
    if HaveEL(EL3) then
        if MVBAR<0> == '1' then           // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    BranchTo(rv, BranchType_UNKNOWN);
```

### aarch32/exceptions/exceptions/ExcVectorBase

```
// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then  // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR;
```

### aarch32/exceptions/ieeefp/AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// ============================

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        AArch64.FPTrappedException(is_ase, element, accumulated_exceptions);

    FPEXC.DEX    = '1';
    FPEXC.TFV    = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF

    AArch32.TakeUndefInstrException();
```

### aarch32/exceptions/syscalls/AArch32.CallHypervisor

```
// AArch32.CallHypervisor()
// ========================
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCException(immediate);
```

### aarch32/exceptions/syscalls/AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// ========================
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate)

    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;

    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCException(immediate);
```

### aarch32/exceptions/syscalls/AArch32.TakeHVCException

```
// AArch32.TakeHVCException()
// ==========================

AArch32.TakeHVCException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32.ITAdvance();
    SSAdvance();

    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;

    exception = ExceptionSyndrome(Exception_HypervisorCall);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
```

### aarch32/exceptions/syscalls/AArch32.TakeSMCException

```
// AArch32.TakeSMCException()
// ==========================
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AArch32.TakeSMCException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    AArch32.ITAdvance();
    SSAdvance();

    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

### aarch32/exceptions/syscalls/AArch32.TakeSVCException

```
// AArch32.TakeSVCException()
// ==========================

AArch32.TakeSVCException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();

    route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_SupervisorCall);
        exception.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```

### aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// ======================
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return,
                     integer vect_offset)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    spsr = GetPSRFromPSTATE();
    if !(exception.type IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE;                       // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    BranchTo(HVBAR + vect_offset, BranchType_UNKNOWN);
    EndOfInstruction();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// ===================
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                  integer vect_offset)
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    spsr = GetPSRFromPSTATE();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    BranchTo(ExcVectorBase() + vect_offset, BranchType_UNKNOWN);
    EndOfInstruction();
```

### aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// ==========================
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                         integer vect_offset)
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    spsr = GetPSRFromPSTATE();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    BranchTo(MVBAR + vect_offset, BranchType_UNKNOWN);
    EndOfInstruction();
```

### aarch32/exceptions/traps/AArch32.CPRegTrap

```
// AArch32.CPRegTrap()
// ===================
// Trapped AArch32 CP14 and CP15 access other than due to CPTR_EL2 or CPACR_EL1.

AArch32.CPRegTrap(bits(2) target_el, bits(32) instr)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if !ELUsingAArch32(target_el) || AArch32.GeneralExceptionsToAArch64() then
        AArch64.CPRegTrap(target_el, instr);

    assert target_el IN {EL1,EL2};
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if target_el == EL2 then
        exception = AArch32.CPRegTrapSyndrome(instr);
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
```

### aarch32/exceptions/traps/AArch32.CPRegTrapSyndrome

```
// AArch32.CPRegTrapSyndrome()
// ===========================
// Return the syndrome information for coprocessor register traps other than
// due to HCPTR or CPACR

ExceptionRecord AArch32.CPRegTrapSyndrome(bits(32) instr)

    ExceptionRecord exception;
    cpnum = UInt(instr<11:8>);

    bits(20) iss = Zeros();
    if instr<27:24> == '1110' && instr<4> == '1' && instr<31:28> != '1111' then
        // MRC/MCR
        case cpnum of
            when 10    exception = ExceptionSyndrome(Exception_FPIDTrap);
            when 14    exception = ExceptionSyndrome(Exception_CP14RTTrap);
            when 15    exception = ExceptionSyndrome(Exception_CP15RTTrap);
            otherwise  Unreachable();
        iss<19:17> = instr<7:5>;    // opc2
        iss<16:14> = instr<23:21>;  // opc1
        iss<13:10> = instr<19:16>;  // CRn
        iss<8:5>   = instr<15:12>;  // Rt
        iss<4:1>   = instr<3:0>;    // CRm
    elsif instr<27:21> == '1100010' && instr<31:28> != '1111' then
        // MRRC/MCRR
        case cpnum of
            when 14    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
            when 15    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
            otherwise  Unreachable();
        iss<19:16> = instr<7:4>;    // opc1
        iss<13:10> = instr<19:16>;  // Rt2
        iss<8:5>   = instr<15:12>;  // Rt
        iss<4:1>   = instr<3:0>;    // CRm
    elsif instr<27:25> == '110' && instr<31:28> != '1111' then
        // LDC/STC
        assert cpnum == 14;
        exception = ExceptionSyndrome(Exception_CP14DTTrap);
        iss<19:12> = instr<7:0>;    // imm8
        iss<4>     = instr<23>;     // U
        iss<2:1>   = instr<24,21>;  // P,W
        if instr<19:16> == '1111' then    // Literal addressing
            iss<8:5> = bits(4) UNKNOWN;
            iss<3>   = '1';
        else
            iss<8:5> = instr<19:16>;  // Rn
            iss<3>   = '0';
    else
        Unreachable();
    iss<0> = instr<20>;     // Direction
    exception.syndrome<19:0> = iss;

    return exception;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEnabled

```
// AArch32.CheckAdvSIMDOrFPEnabled()
// =================================
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        AArch64.CheckFPAdvSIMDEnabled();
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then UNDEFINED;

            // Check if access disabled in CPACR
            case cpacr_cp10 of
                when 'x0'  disabled = TRUE;
                when '01'  disabled = PSTATE.EL == EL0;
                when '11'  disabled = FALSE;
            if disabled then UNDEFINED;

        // If required, check FPEXC enabled bit.
        if fpexc_check && FPEXC.EN == '0' then UNDEFINED;

        AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
```

### aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)

    if HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if HaveEL(EL2) && !IsSecure() then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;

            if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in HCPTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                if advsimd then
                    exception.syndrome<5> = '1';
                else
                    exception.syndrome<5> = '0';
                    exception.syndrome<3:0> = '1010';         // coproc field, always 0xA

                if PSTATE.EL == EL2 then
                    AArch32.TakeUndefInstrException(exception);
                else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                    AArch32.TakeHypTrapException(exception);

        if HaveEL(EL3) && !ELUsingAArch32(EL3) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

    return;
```

### aarch32/exceptions/traps/AArch32.CheckForSMCTrap

```
// AArch32.CheckForSMCTrap()
// =========================
// Check for trap on SMC instruction

AArch32.CheckForSMCTrap()

    if HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCTrap(Zeros(16));
    else
        route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} && HCR.TSC == '1';
        if route_to_hyp then
            exception = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(exception);
```

### aarch32/exceptions/traps/AArch32.CheckForWFxTrap

```
// AArch32.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        AArch64.CheckForWFxTrap(target_el, is_wfe);
        return;

    case target_el of
        when EL1  trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2  trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3  trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

    if trap then
        if (target_el == EL1 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) &&
            HCR_EL2.TGE == '1') then
            AArch64.WFxTrap(target_el, is_wfe);

        if target_el == EL3 then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            exception = ExceptionSyndrome(Exception_WFxTrap);
            exception.syndrome<0> = if is_wfe then '1' else '0';
            AArch32.TakeHypTrapException(exception);
        else
            AArch32.TakeUndefInstrException();
```

### aarch32/exceptions/traps/AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)

    if PSTATE.EL == EL2 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR_EL1.ITD);

    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

        // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
        next_instr = AArch32.MemSingle[NextInstrAddr(), 2, AccType_IFETCH, TRUE];

        if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx',
                          '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
            // taken on the IT instruction or the next instruction. This is not reflected in
            // the pseudocode, which always takes the exception on the IT instruction. This
            // also does not take into account cases where the next instruction is UNPREDICTABLE.
            UNDEFINED;

    return;
```

## aarch32/exceptions/traps/AArch32.CheckIllegalState

```
// AArch32.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()

    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
```

## aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// ============================
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()

    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR_EL1.SED);

    if setend_disabled == '1' then
        UNDEFINED;

    return;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x14;

    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
```

### aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// ==================================
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

### aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException()
    exception = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException(ExceptionRecord exception)

    route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
```

### aarch32/exceptions/traps/AArch32.UndefinedFault

```
// AArch32.UndefinedFault()
// ========================

AArch32.UndefinedFault()

    if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefinedFault();

    AArch32.TakeUndefInstrException();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### J1.2.3 aarch32/functions

This section includes the following pseudocode functions:


- *aarch32/functions/aborts/AArch32.CreateFaultRecord* on page J1-5301.
- *aarch32/functions/aborts/AArch32.DomainValid* on page J1-5301.
- *aarch32/functions/aborts/AArch32.FaultStatusLD* on page J1-5301.
- *aarch32/functions/aborts/AArch32.FaultStatusSD* on page J1-5302.
- *aarch32/functions/aborts/EncodeSDFSC* on page J1-5302.
- *aarch32/functions/common/A32ExpandImm* on page J1-5303.
- *aarch32/functions/common/A32ExpandImm_C* on page J1-5303.
- *aarch32/functions/common/DecodeImmShift* on page J1-5303.
- *aarch32/functions/common/DecodeRegShift* on page J1-5303.
- *aarch32/functions/common/RRX* on page J1-5304.
- *aarch32/functions/common/RRX_C* on page J1-5304.
- *aarch32/functions/common/SRType* on page J1-5304.
- *aarch32/functions/common/Shift* on page J1-5304.
- *aarch32/functions/common/Shift_C* on page J1-5304.
- *aarch32/functions/common/T32ExpandImm* on page J1-5305.
- *aarch32/functions/common/T32ExpandImm_C* on page J1-5305.
- *aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps* on page J1-5305.
- *aarch32/functions/coproc/AArch32.CheckCoprocInstrTraps* on page J1-5305.
- *aarch32/functions/coproc/CP14DebugInstrDecode* on page J1-5306.
- *aarch32/functions/coproc/CP14JazelleInstrDecode* on page J1-5306.
- *aarch32/functions/coproc/CP14TraceInstrDecode* on page J1-5306.
- *aarch32/functions/coproc/CP15InstrDecode* on page J1-5306.
- *aarch32/functions/coproc/Coproc_CanWriteAPSR* on page J1-5306.
- *aarch32/functions/coproc/Coproc_CheckInstr* on page J1-5306.
- *aarch32/functions/coproc/Coproc_DoneLoading* on page J1-5307.
- *aarch32/functions/coproc/Coproc_DoneStoring* on page J1-5307.
- *aarch32/functions/coproc/Coproc_GetOneWord* on page J1-5307.
- *aarch32/functions/coproc/Coproc_GetTwoWords* on page J1-5308.
- *aarch32/functions/coproc/Coproc_GetWordToStore* on page J1-5308.
- *aarch32/functions/coproc/Coproc_InternalOperation* on page J1-5308.
- *aarch32/functions/coproc/Coproc_SendLoadedWord* on page J1-5308.
- *aarch32/functions/coproc/Coproc_SendOneWord* on page J1-5308.
- *aarch32/functions/coproc/Coproc_SendTwoWords* on page J1-5308.
- *aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass* on page J1-5308.
- *aarch32/functions/exclusive/AArch32.IsExclusiveVA* on page J1-5309.
- *aarch32/functions/exclusive/AArch32.MarkExclusiveVA* on page J1-5309.
- *aarch32/functions/exclusive/AArch32.SetExclusiveMonitors* on page J1-5309.
- *aarch32/functions/float/CheckAdvSIMDEnabled* on page J1-5309.
- *aarch32/functions/float/CheckAdvSIMDOrVFPEnabled* on page J1-5309.
- *aarch32/functions/float/CheckCryptoEnabled32* on page J1-5310.
- *aarch32/functions/float/CheckVFPEnabled* on page J1-5310.
- *aarch32/functions/float/FPHalvedSub* on page J1-5310.
- *aarch32/functions/float/FPRSqrtStep* on page J1-5310.
- *aarch32/functions/float/FPRecipStep* on page J1-5311.
- *aarch32/functions/float/StandardFPSCRValue* on page J1-5311.
- *aarch32/functions/memory/AArch32.CheckAlignment* on page J1-5311.
- *aarch32/functions/memory/AArch32.MemSingle* on page J1-5311.
- *aarch32/functions/memory/GenerateAlignmentException* on page J1-5312.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *aarch32/functions/memory/Hint_PreloadData* on page J1-5312.
- *aarch32/functions/memory/Hint_PreloadDataForWrite* on page J1-5312.
- *aarch32/functions/memory/Hint_PreloadInstr* on page J1-5312.
- *aarch32/functions/memory/MemA* on page J1-5313.
- *aarch32/functions/memory/MemO* on page J1-5313.
- *aarch32/functions/memory/MemU* on page J1-5313.
- *aarch32/functions/memory/MemU_unpriv* on page J1-5313.
- *aarch32/functions/memory/Mem_with_type* on page J1-5314.
- *aarch32/functions/registers/AArch32.ResetGeneralRegisters* on page J1-5315.
- *aarch32/functions/registers/AArch32.ResetSIMDFPRegisters* on page J1-5315.
- *aarch32/functions/registers/AArch32.ResetSpecialRegisters* on page J1-5315.
- *aarch32/functions/registers/AArch32.ResetSystemRegisters* on page J1-5316.
- *aarch32/functions/registers/ALUExceptionReturn* on page J1-5316.
- *aarch32/functions/registers/ALUWritePC* on page J1-5316.
- *aarch32/functions/registers/BXWritePC* on page J1-5316.
- *aarch32/functions/registers/BranchWritePC* on page J1-5316.
- *aarch32/functions/registers/D* on page J1-5316.
- *aarch32/functions/registers/Din* on page J1-5317.
- *aarch32/functions/registers/LR* on page J1-5317.
- *aarch32/functions/registers/LoadWritePC* on page J1-5317.
- *aarch32/functions/registers/LookUpRIndex* on page J1-5317.
- *aarch32/functions/registers/Monitor_mode_registers* on page J1-5318.
- *aarch32/functions/registers/PC* on page J1-5318.
- *aarch32/functions/registers/PCStoreValue* on page J1-5318.
- *aarch32/functions/registers/Q* on page J1-5318.
- *aarch32/functions/registers/Qin* on page J1-5318.
- *aarch32/functions/registers/R* on page J1-5318.
- *aarch32/functions/registers/RBankSelect* on page J1-5319.
- *aarch32/functions/registers/Rmode* on page J1-5319.
- *aarch32/functions/registers/S* on page J1-5320.
- *aarch32/functions/registers/SP* on page J1-5320.
- *aarch32/functions/registers/_Dclone* on page J1-5320.
- *aarch32/functions/system/AArch32.ExceptionReturn* on page J1-5320.
- *aarch32/functions/system/AArch32.ITAdvance* on page J1-5321.
- *aarch32/functions/system/AArch32.WriteMode* on page J1-5321.
- *aarch32/functions/system/BadMode* on page J1-5321.
- *aarch32/functions/system/BankedRegisterAccessValid* on page J1-5321.
- *aarch32/functions/system/CPSRWriteByInstr* on page J1-5322.
- *aarch32/functions/system/ConditionPassed* on page J1-5322.
- *aarch32/functions/system/CurrentCond* on page J1-5322.
- *aarch32/functions/system/GenerateIntegerZeroDivide* on page J1-5323.
- *aarch32/functions/system/InITBlock* on page J1-5323.
- *aarch32/functions/system/IntegerZeroDivideTrappingEnabled* on page J1-5323.
- *aarch32/functions/system/LastInITBlock* on page J1-5323.
- *aarch32/functions/system/SPSRWriteByInstr* on page J1-5323.
- *aarch32/functions/system/SPSRaccessValid* on page J1-5323.
- *aarch32/functions/system/SelectInstrSet* on page J1-5324.
- *aarch32/functions/v6simd/Sat* on page J1-5324.
- *aarch32/functions/v6simd/SignedSat* on page J1-5324.
- *aarch32/functions/v6simd/UnsignedSat* on page J1-5324.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/functions/aborts/AArch32.CreateFaultRecord

```
// AArch32.CreateFaultRecord()
// ===========================

FaultRecord AArch32.CreateFaultRecord(Fault type, bits(40) ipaddress,  bits(4) domain,
                                      integer level, AccType acctype, boolean write, bit extflag,
                                      bits(4) debugmoe, boolean secondstage, boolean s2fs1walk)

    FaultRecord fault;
    fault.type = type;
    if (type != Fault_None && PSTATE.EL != EL2 && TTBCR.EAE == '0' && !secondstage && !s2fs1walk &&
        AArch32.DomainValid(type, level)) then
        fault.domain = domain;
    else
        fault.domain = bits(4) UNKNOWN;
    fault.debugmoe = debugmoe;
    fault.ipaddress = ZeroExtend(ipaddress);
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fs1walk = s2fs1walk;

    return fault;
```

### aarch32/functions/aborts/AArch32.DomainValid

```
// AArch32.DomainValid()
// =====================
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault type, integer level)
    assert type != Fault_None;

    case type of
        when Fault_Domain
            return TRUE;
        when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;
```

### aarch32/functions/aborts/AArch32.FaultStatusLD

```
// AArch32.FaultStatusLD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;

    bits(32) fsr = Zeros();
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '1';
    fsr<5:0> = EncodeLDFSC(fault.type, fault.level);

    return fsr;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/functions/aborts/AArch32.FaultStatusSD

```
// AArch32.FaultStatusSD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;

    bits(32) fsr = Zeros();
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '0';
    fsr<10,3:0> = EncodeSDFSC(fault.type, fault.level);
    if d_side then
        fsr<7:4> = fault.domain;               // Domain field (data fault only)

    return fsr;
```

### aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault type, integer level)

    bits(5) result;
    case type of
        when Fault_AccessFlag
            assert level IN {1,2};
            result = if level == 1 then '00011' else '00110';
        when Fault_Alignment
            result = '00001';
        when Fault_Permission
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault_Domain
            assert level IN {1,2};
            result = if level == 1 then '01001' else '01011';
        when Fault_Translation
            assert level IN {1,2};
            result = if level == 1 then '00101' else '00111';
        when Fault_SyncExternal
            result = '01000';
        when Fault_SyncExternalOnWalk
            assert level IN {1,2};
            result = if level == 1 then '01100' else '01110';
        when Fault_SyncParity
            result = '11001';
        when Fault_SyncParityOnWalk
            assert level IN {1,2};
            result = if level == 1 then '11100' else '11110';
        when Fault_AsyncParity
            result = '11000';
        when Fault_AsyncExternal
            result = '10110';
        when Fault_Debug
            result = '00010';
        when Fault_TLBConflict
            result = '10000';
        when Fault_Lockdown
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result = '10100';   // IMPLEMENTATION DEFINED
        when Fault_Exclusive
            result = '10101';   // IMPLEMENTATION DEFINED
        when Fault_ICacheMaint
            result = '00100';
        otherwise
            Unreachable();

    return result;
```

### aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

### aarch32/functions/common/A32ExpandImm_C

```
// A32ExpandImm_C()
// ================

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRType_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

### aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

### aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) type)
    case type of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;
```

### aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

### aarch32/functions/common/RRX_C

```
// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

### aarch32/functions/common/SRType

```
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

### aarch32/functions/common/Shift

```
// Shift()
// =======

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, type, amount, carry_in);
    return result;
```

### aarch32/functions/common/Shift_C

```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// ==============

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

### aarch32/functions/common/T32ExpandImm_C

```
// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

### aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained CP15 traps in HSTR and HCR.

boolean AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        if PSTATE.EL == EL0 && !ELUsingAArch32(EL2) then
            return AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !(major IN {4,14}) && HSTR<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR.TIDCP
        if (HCR.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
             (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```

### aarch32/functions/coproc/AArch32.CheckCoprocInstrTraps

```
AArch32.CheckCoprocInstrTraps(bits(32) instr);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/functions/coproc/CP14DebugInstrDecode

```
boolean CP14DebugInstrDecode(bits(32) instr);
```

### aarch32/functions/coproc/CP14JazelleInstrDecode

```
boolean CP14JazelleInstrDecode(bits(32) instr);
```

### aarch32/functions/coproc/CP14TraceInstrDecode

```
boolean CP14TraceInstrDecode(bits(32) instr);
```

### aarch32/functions/coproc/CP15InstrDecode

```
boolean CP15InstrDecode(bits(32) instr);
```

### aarch32/functions/coproc/Coproc_CanWriteAPSR

```
// Coproc_CanWriteAPSR()
// =====================
// Determines whether the AArch32 CP14 or CP15 coprocessor instruction can write to APSR flags.

boolean Coproc_CanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert !(cp_num IN {10,11});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

### aarch32/functions/coproc/Coproc_CheckInstr

```
// Coproc_CheckInstr()
// ===================
// Check coprocessor instruction for enables and disables

Coproc_CheckInstr(integer cp_num, bits(32) instr)
    assert cp_num == UInt(instr<11:8>) && !(cp_num IN {10,11});

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        AArch64.CheckCoprocInstr(instr);
        return;

    // Decode the AArch32 coprocessor instruction
    if instr<27:24> == '1110' && instr<4> == '1' then  // MRC/MCR
        cprt = TRUE;  cpdt = FALSE;  nreg = 1;
        opc1 = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn  = UInt(instr<19:16>);
        CRm  = UInt(instr<3:0>);
    elsif instr<27:21> == '1100010' then               // MRRC/MCRR
        cprt = TRUE;  cpdt = FALSE;  nreg = 2;
        opc1 = UInt(instr<7:4>);
        CRm  = UInt(instr<3:0>);
    elsif instr<27:25> == '110' then                   // LDC/STC
        cprt = FALSE;  cpdt = TRUE;  nreg = 0;
        lng  = instr<22> == '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        opc1 = 0;
        CRn  = UInt(instr<15:12>);
    else                                                // CDP
        cprt = FALSE;  cpdt = FALSE;  nreg = 0;
        opc1 = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn  = UInt(instr<19:16>);
        CRm  = UInt(instr<3:0>);
    two = instr<31:28> == '1111';                       // MRC2/MCR2/etc.

    // Coarse-grain decode into CP14 or CP15 operations. Each of the CPxxxInstrDecode functions
    // returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
    if cp_num == 14 then
        // LDC and STC only supported for c5, CDP, and MRC2, MCR2, etc. not supported for CP14
        if (cpdt && (CRn != 5 || lng)) || (!cprt && !cpdt) || two then
            allocated = FALSE;
        else
            // Coarse-grained decode of CP14 based on opc1 field
            case opc1 of
                when 0     allocated = CP14DebugInstrDecode(instr);
                when 1     allocated = CP14TraceInstrDecode(instr);
                when 7     allocated = CP14JazelleInstrDecode(instr);    // JIDR only
                otherwise  allocated = FALSE;          // All other values are unallocated

    elsif cp_num == 15 then
        // LDC, STC, and CDP, and MRC2, MCR2, etc. not supported by CP15
        if !cprt || two then
            allocated = FALSE;
        else
            allocated = CP15InstrDecode(instr);

            // Coarse-grain traps to EL2 have a higher priority than Undefined Instruction
            if AArch32.CheckCP15InstrCoarseTraps(CRn, nreg, CRm) then
                // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
                // Non-secure User mode is UNDEFINED when the trap is disabled, then it is
                // IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
                // when the trap is enabled.
                if PSTATE.EL == EL0 && !IsSecure() && !allocated then
                    if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at NS EL0" then
                        UNDEFINED;
                AArch32.CPRegTrap(EL2, instr);

    else
        allocated = FALSE;                              // All other coprocessors are unallocated

    if !allocated then
        UNDEFINED;

    // If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
    AArch32.CheckCoprocInstrTraps(instr);

    return;
```

## aarch32/functions/coproc/Coproc_DoneLoading

```
boolean Coproc_DoneLoading(integer cp_num, bits(32) instr);
```

## aarch32/functions/coproc/Coproc_DoneStoring

```
boolean Coproc_DoneStoring(integer cp_num, bits(32) instr);
```

## aarch32/functions/coproc/Coproc_GetOneWord

```
bits(32) Coproc_GetOneWord(integer cp_num, bits(32) instr);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/functions/coproc/Coproc_GetTwoWords

```
(bits(32), bits(32)) Coproc_GetTwoWords(integer cp_num, bits(32) instr);
```

### aarch32/functions/coproc/Coproc_GetWordToStore

```
bits(32) Coproc_GetWordToStore(integer cp_num, bits(32) instr);
```

### aarch32/functions/coproc/Coproc_InternalOperation

```
Coproc_InternalOperation(integer cp_num, bits(32) instr);
```

### aarch32/functions/coproc/Coproc_SendLoadedWord

```
Coproc_SendLoadedWord(bits(32) word, integer cp_num, bits(32) instr);
```

### aarch32/functions/coproc/Coproc_SendOneWord

```
Coproc_SendOneWord(bits(32) word, integer cp_num, bits(32) instr);
```

### aarch32/functions/coproc/Coproc_SendTwoWords

```
Coproc_SendTwoWords(bits(32) word2, bits(32) word1, integer cp_num, bits(32) instr);
```

### aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// ===============================

// Return TRUE if the Exclusive Monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusive Monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());
        if memaddrdesc.memattrs.shareable then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

### aarch32/functions/exclusive/AArch32.IsExclusiveVA

```
boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

### aarch32/functions/exclusive/AArch32.MarkExclusiveVA

```
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```

### aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// ==============================

// Sets the Exclusive Monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = (address != Align(address, size));

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

### aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====================

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```

### aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

```
// CheckAdvSIMDOrVFPEnabled()
// ==========================
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

### aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// ======================

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```

### aarch32/functions/float/CheckVFPEnabled

```
// CheckVFPEnabled()
// =================

CheckVFPEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

### aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =============

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

### aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =============

bits(32) FPRSqrtStep(bits(32) op1, bits(32) op2)
    FPCRType fpcr = StandardFPSCRValue();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(32) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        result = FPHalvedSub(FPThree('0'), product, fpcr);
    return result;
```

### aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =============

bits(32) FPRecipStep(bits(32) op1, bits(32) op2)
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(32) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        result = FPSub(FPTwo('0'), product, fpcr);
    return result;
```

### aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// ====================

FPCRType StandardFPSCRValue()
    return '00000' : FPSCR.AHP : '1100000000000000000000000';
```

### aarch32/functions/memory/AArch32.CheckAlignment

```
// AArch32.CheckAlignment()
// ========================

boolean AArch32.CheckAlignment(bits(32) address, integer size, AccType acctype, boolean iswrite)

    aligned = (address == Align(address, size));
    A = (if PSTATE.EL == EL2 then HSCTLR.A else SCTLR.A);

    if !aligned && (acctype == AccType_ATOMIC || acctype == AccType_ORDERED || A == '1') then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

### aarch32/functions/memory/AArch32.MemSingle

```
// AArch32.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    value = _Mem[memaddrdesc, size, acctype];
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8)
value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    _Mem[memaddrdesc, size, acctype] = value;
    return;
```

### aarch32/functions/memory/GenerateAlignmentException

```
GenerateAlignmentException();
```

### aarch32/functions/memory/Hint_PreloadData

```
Hint_PreloadData(bits(32) address);
```

### aarch32/functions/memory/Hint_PreloadDataForWrite

```
Hint_PreloadDataForWrite(bits(32) address);
```

### aarch32/functions/memory/Hint_PreloadInstr

```
Hint_PreloadInstr(bits(32) address);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/functions/memory/MemA

```
// MemA[] - non-assignment form
// ============================

bits(8*size) MemA[bits(32) address, integer size]
    acctype = AccType_ATOMIC;
    return Mem_with_type[address, size, acctype];

// MemA[] - assignment form
// ========================

MemA[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ATOMIC;
    Mem_with_type[address, size, acctype] = value;
    return;
```

### aarch32/functions/memory/MemO

```
// MemO[] - non-assignment form
// ============================

bits(8*size) MemO[bits(32) address, integer size]
    acctype = AccType_ORDERED;
    return Mem_with_type[address, size, acctype];

// MemO[] - assignment form
// ========================

MemO[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ORDERED;
    Mem_with_type[address, size, acctype] = value;
    return;
```

### aarch32/functions/memory/MemU

```
// MemU[] - non-assignment form
// ============================

bits(8*size) MemU[bits(32) address, integer size]
    acctype = AccType_NORMAL;
    return Mem_with_type[address, size, acctype];

// MemU[] - assignment form
// ========================

MemU[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_NORMAL;
    Mem_with_type[address, size, acctype] = value;
    return;
```

### aarch32/functions/memory/MemU_unpriv

```
// MemU_unpriv[] - non-assignment form
// ===================================

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType_UNPRIV;
    return Mem_with_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// ===============================

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    acctype = AccType_UNPRIV;
    Mem_with_type[address, size, acctype] = value;
    return;
```

### aarch32/functions/memory/Mem_with_type

```
// Mem_with_type[] - non-assignment (read) form
// ============================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    integer i;
    boolean iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned];

    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem_with_type[] - assignment (write) form
// =========================================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
    integer i;
    boolean iswrite = TRUE;

    if BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        AArch32.MemSingle[address, size, acctype, aligned] = value;
    return;
```

### aarch32/functions/registers/AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// ===============================

AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN;   // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
        if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

    return;
```

### aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```
// AArch32.ResetSIMDFPRegisters()
// ==============================

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

    return;
```

### aarch32/functions/registers/AArch32.ResetSpecialRegisters

```
// AArch32.ResetSpecialRegisters()
// ===============================

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq = bits(32) UNKNOWN;
    SPSR_irq = bits(32) UNKNOWN;
    SPSR_svc = bits(32) UNKNOWN;
    SPSR_abt = bits(32) UNKNOWN;
    SPSR_und = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

    return;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/functions/registers/AArch32.ResetSystemRegisters

```
AArch32.ResetSystemRegisters(boolean cold_reset);
```

### aarch32/functions/registers/ALUExceptionReturn

```
// ALUExceptionReturn()
// ====================

ALUExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;          // UNDEFINED or NOP
    else
        AArch32.ExceptionReturn(address, SPSR[]);
```

### aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address);
    else
        BranchWritePC(address);
```

### aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address)
    if address<0> == '1' then
        SelectInstrSet(InstrSet_T32);
        address<0> = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the processor takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address<1> == '1' && ConstrainUnpredictableBool() then
            address<1> = '0';
    BranchTo(address, BranchType_UNKNOWN);
```

### aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        address<1:0> = '00';
    else
        address<0> = '0';
    BranchTo(address, BranchType_UNKNOWN);
```

### aarch32/functions/registers/D

```
// D[] - non-assignment form
// =========================
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    return _V[n DIV 2]<base+63:base>;

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    _V[n DIV 2]<base+63:base> = value;
    return;
```

### aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];
```

### aarch32/functions/registers/LR

```
// LR - assignment form
// ====================

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// ========================

bits(32) LR
    return R[14];
```

### aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address);
```

### aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// ==============

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    case n of  // Select  index by mode:     usr fiq irq svc abt und hyp
        when 8     result = RBankSelect(mode,  8, 24,  8,  8,  8,  8,  8);
        when 9     result = RBankSelect(mode,  9, 25,  9,  9,  9,  9,  9);
        when 10    result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11    result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12    result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13    result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14    result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise  result = n;

    return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/functions/registers/Monitor_mode_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

### aarch32/functions/registers/PC

```
// PC - non-assignment form
// ========================

bits(32) PC
    return R[15];               // This includes the offset from AArch32 state
```

### aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// ==============

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before ARMv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

### aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =========================

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return _V[n];

// Q[] - assignment form
// =====================

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    _V[n] = value;
    return;
```

### aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// ===========================

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

### aarch32/functions/registers/R

```
// R[] - assignment form
// =====================

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =========================

bits(32) R[integer n]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];
```

### aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                    integer svc, integer abt, integer und, integer hyp)

    case mode of
        when M32_User    result = usr;  // User mode
        when M32_FIQ     result = fiq;  // FIQ mode
        when M32_IRQ     result = irq;  // IRQ mode
        when M32_Svc     result = svc;  // Supervisor mode
        when M32_Abort   result = abt;  // Abort mode
        when M32_Hyp     result = hyp;  // Hyp mode
        when M32_Undef   result = und;  // Undefined mode
        when M32_System  result = usr;  // System mode uses User mode registers
        otherwise        Unreachable(); // Monitor mode

    return result;
```

### aarch32/functions/registers/Rmode

```
// Rmode[] - non-assignment form
// =============================

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =========================

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if !HighestELUsingAArch32() && ConstrainUnpredictableBool() then
            _R[LookUpRIndex(n, mode)] = ZeroExtend(value);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;

    return;
```

### aarch32/functions/registers/S

```
// S[] - non-assignment form
// =========================

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    return _V[n DIV 4]<base+31:base>;

// S[] - assignment form
// =====================

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    _V[n DIV 4]<base+31:base> = value;
    return;
```

### aarch32/functions/registers/SP

```
// SP - assignment form
// ====================

SP = bits(32) value
    R[13] = value;
    return;

// SP - non-assignment form
// ========================

bits(32) SP
    return R[13];
```

### aarch32/functions/registers/_Dclone

```
array bits(64) _Dclone[0..31];
```

### aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =========================

AArch32.ExceptionReturn(bits(32) new_pc, bits(32) spsr)

    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    EventRegisterSet();

    // Align PC[1:0] according to the target instruction set state
    if spsr<5> == '1' then                // T32
        new_pc = Align(new_pc, 2);
    else                                  // A32
        new_pc = Align(new_pc, 4);

    BranchTo(new_pc, BranchType_UNKNOWN);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/functions/system/AArch32.ITAdvance

```
// AArch32.ITAdvance()
// ===================

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

### aarch32/functions/system/AArch32.WriteMode

```
// AArch32.WriteMode()
// ===================
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    if !valid then
        PSTATE.IL = '1';
    else
        PSTATE.M   = mode;
        PSTATE.EL  = el;
        PSTATE.nRW = '1';
        PSTATE.SP  = if mode IN {M32_User,M32_System} then '0' else '1';
    return;
```

### aarch32/functions/system/BadMode

```
// BadMode()
// =========

boolean BadMode(bits(5) mode)
    case mode of
        when M32_User    result = FALSE;
        when M32_FIQ     result = FALSE;
        when M32_IRQ     result = FALSE;
        when M32_Svc     result = FALSE;
        when M32_Monitor result = !HaveEL(EL3);
        when M32_Abort   result = FALSE;
        when M32_Hyp     result = !HaveEL(EL2);
        when M32_Undef   result = FALSE;
        when M32_System  result = FALSE;
        otherwise     result = TRUE;
    return result;
```

### aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

    case SYSm of
        when '000xx', '00100'                       // R8_usr to R12_usr
            if mode != M32_FIQ then UNPREDICTABLE;
        when '00101'                                // SP_usr
            if mode == M32_System then UNPREDICTABLE;
        when '00110'                                // LR_usr
            if mode IN {M32_Hyp,M32_System} then UNPREDICTABLE;
        when '010xx', '0110x', '01110'              // R8_fiq to R12_fiq, SP_fiq, LR_fiq
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            if mode == M32_FIQ then UNPREDICTABLE;
        when '1000x'                                    // LR_irq, SP_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '1001x'                                    // LR_svc, SP_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '1010x'                                    // LR_abt, SP_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '1011x'                                    // LR_und, SP_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '1110x'                                    // LR_mon, SP_mon
            if !HaveEL(EL3) || !IsSecure() || mode == M32_Monitor then UNPREDICTABLE;
        when '11110'                                    // ELR_hyp, only from Monitor or Hyp mode
            if !HaveEL(EL2) || !(mode IN {M32_Monitor,M32_Hyp}) then UNPREDICTABLE;
        when '11111'                                    // SP_hyp, only from Monitor mode
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

## aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// ==================
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;              // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        // Bits <23:20> are RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>;                    // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteMode sets PSTATE.IL to 1 if 'value<4:0>' is not a legal mode value
            AArch32.WriteMode(value<4:0>);

    return;
```

## aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

## aarch32/functions/system/CurrentCond

```
bits(4) AArch32.CurrentCond();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/functions/system/GenerateIntegerZeroDivide

```
GenerateIntegerZeroDivide();
```

### aarch32/functions/system/InITBlock

```
// InITBlock()
// ===========

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

### aarch32/functions/system/IntegerZeroDivideTrappingEnabled

```
boolean IntegerZeroDivideTrappingEnabled();
```

### aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

### aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// ==================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    new_spsr = SPSR[];

    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0],J bits

    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>;  // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits

    SPSR[] = new_spsr;                   // UNPREDICTABLE if User or System mode

    return;
```

### aarch32/functions/system/SPSRaccessValid

```
// SPSRaccessValid()
// =================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110'                                                    // SPSR_fiq
            if mode == M32_FIQ   then UNPREDICTABLE;
        when '10000'                                                    // SPSR_irq
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            if mode == M32_IRQ   then UNPREDICTABLE;
        when '10010'                                                    // SPSR_svc
            if mode == M32_Svc   then UNPREDICTABLE;
        when '10100'                                                    // SPSR_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110'                                                    // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100'                                                    // SPSR_mon
            if !HaveEL(EL3) || mode == M32_Monitor || !IsSecure() then UNPREDICTABLE;
        when '11110'                                                    // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

### aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';

    return;
```

### aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

### aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```

### aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

## J1.2.4 aarch32/translation

This section includes the following pseudocode functions:

- *aarch32/translation/attrs/AArch32.DefaultTEXDecode* on page J1-5325.
- *aarch32/translation/attrs/AArch32.InstructionDevice* on page J1-5326.
- *aarch32/translation/attrs/AArch32.RemappedTEXDecode* on page J1-5326.
- *aarch32/translation/attrs/AArch32.S1AttrDecode* on page J1-5327.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

- *aarch32/translation/attrs/AArch32.TranslateAddressS1Off* on page J1-5328.
- *aarch32/translation/checks/AArch32.CheckDomain* on page J1-5329.
- *aarch32/translation/checks/AArch32.CheckPermission* on page J1-5329.
- *aarch32/translation/checks/AArch32.CheckS2Permission* on page J1-5330.
- *aarch32/translation/debug/AArch32.CheckBreakpoint* on page J1-5330.
- *aarch32/translation/debug/AArch32.CheckDebug* on page J1-5331.
- *aarch32/translation/debug/AArch32.CheckVectorCatch* on page J1-5331.
- *aarch32/translation/debug/AArch32.CheckWatchpoint* on page J1-5332.
- *aarch32/translation/faults/AArch32.AccessFlagFault* on page J1-5332.
- *aarch32/translation/faults/AArch32.AddressSizeFault* on page J1-5332.
- *aarch32/translation/faults/AArch32.AlignmentFault* on page J1-5333.
- *aarch32/translation/faults/AArch32.AsynchExternalAbort* on page J1-5333.
- *aarch32/translation/faults/AArch32.DebugFault* on page J1-5333.
- *aarch32/translation/faults/AArch32.DomainFault* on page J1-5333.
- *aarch32/translation/faults/AArch32.NoFault* on page J1-5334.
- *aarch32/translation/faults/AArch32.PermissionFault* on page J1-5334.
- *aarch32/translation/faults/AArch32.TranslationFault* on page J1-5334.
- *aarch32/translation/translation/AArch32.FirstStageTranslate* on page J1-5334.
- *aarch32/translation/translation/AArch32.FullTranslate* on page J1-5335.
- *aarch32/translation/translation/AArch32.SecondStageTranslate* on page J1-5336.
- *aarch32/translation/translation/AArch32.SecondStageWalk* on page J1-5336.
- *aarch32/translation/translation/AArch32.TranslateAddress* on page J1-5337.
- *aarch32/translation/walk/AArch32.TranslationTableWalkLD* on page J1-5337.
- *aarch32/translation/walk/AArch32.TranslationTableWalkSD* on page J1-5341.
- *aarch32/translation/walk/RemapRegsHaveResetValues* on page J1-5343.


## aarch32/translation/attrs/AArch32.DefaultTEXDecode

```
// AArch32.DefaultTEXDecode()
// ==========================

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes mematTrs;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits();
        TEX = texcb<4:2>;  C = texcb<1>;  B = texcb<0>;

    case TEX:C:B of
        when '00000'
            // Device-nGnRnE
            mematTrs.type = MemType_Device;
            mematTrs.device = DeviceType_nGnRnE;
        when '00001', '01000'
            // Device-nGnRE
            mematTrs.type = MemType_Device;
            mematTrs.device = DeviceType_nGnRE;
        when '00010', '00011', '00100'
            // Write-back or Write-through Read allocate, or Non-cacheable
            mematTrs.type = MemType_Normal;
            mematTrs.inner = ShortConvertAttrsHints(C:B, acctype);
            mematTrs.outer = ShortConvertAttrsHints(C:B, acctype);
            mematTrs.shareable = (S == '1');
        when '00110'
            mematTrs = MemoryAttributes IMPLEMENTATION_DEFINED;
        when '00111'
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            // Non-cacheable, or Write-back Write allocate
            mematTrs.type = MemType_Normal;
            mematTrs.inner = ShortConvertAttrsHints('01', acctype);
            mematTrs.outer = ShortConvertAttrsHints('01', acctype);
            mematTrs.shareable = (S == '1');
        when '1xxxx'
            // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
            mematTrs.type = MemType_Normal;
            mematTrs.inner = ShortConvertAttrsHints(C:B, acctype);
            mematTrs.outer = ShortConvertAttrsHints(TEX<1:0>, acctype);
            mematTrs.shareable = (S == '1');
        otherwise
            // Reserved, handled above
            Unreachable();

    // transient bits are not supported in this format
    mematTrs.inner.transient = FALSE;
    mematTrs.outer.transient = FALSE;

    // distinction between inner and outer shareable is not supported in this format
    mematTrs.outershareable = mematTrs.shareable;

    return MemAttrDefaults(mematTrs);
```

## aarch32/translation/attrs/AArch32.InstructionDevice

```
// AArch32.InstructionDevice()
// ===========================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch32.InstructionDevice(AddressDescriptor addrdesc, bits(32) vaddress,
                                            bits(40) ipaddress, integer level,  bits(4) domain,
                                            AccType acctype, boolean iswrite, boolean secondstage,
                                            boolean s2fs1walk)

    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, level, acctype, iswrite,
                                                 secondstage, s2fs1walk);
    else
        addrdesc.mematTrs.type = MemType_Normal;
        addrdesc.mematTrs.inner.attrs = MemAttr_NC;
        addrdesc.mematTrs.inner.hints = MemHint_No;
        addrdesc.mematTrs.outer = addrdesc.mematTrs.inner;
        addrdesc.mematTrs = MemAttrDefaults(addrdesc.mematTrs);

    return addrdesc;
```

## aarch32/translation/attrs/AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// ===========================

MemoryAttributes AArch32.RemappedTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes mematTrs;

    region = UInt(TEX<0>:C:B);          // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        mematTrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        base = 2 * region;
        attrfield = PRRR<base+1:base>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if attrfield == '11' then      // Reserved, maps to allocated value
        (-, attrfield) = ConstrainUnpredictableBits();

    case attrfield of
        when '00'                  // Device-nGnRnE
            mematTrs.type = MemType_Device;
            mematTrs.device = DeviceType_nGnRnE;
        when '01'                  // Device-nGnRE
            mematTrs.type = MemType_Device;
            mematTrs.device = DeviceType_nGnRE;
        when '10'
            mematTrs.type = MemType_Normal;
            mematTrs.inner = ShortConvertAttrsHints(NMRR<base+1:base>, acctype);
            mematTrs.outer = ShortConvertAttrsHints(NMRR<base+17:base+16>, acctype);
            s_bit = if S == '0' then PRRR.NS0 else PRRR.NS1;
            mematTrs.shareable = (s_bit == '1');
            mematTrs.outershareable = (s_bit == '1' && PRRR<region+24> == '0');
        when '11'
            Unreachable();

// transient bits are not supported in this format
mematTrs.inner.transient = FALSE;
mematTrs.outer.transient = FALSE;

return MemAttrDefaults(mematTrs);
```

## aarch32/translation/attrs/AArch32.S1AttrDecode

```
// AArch32.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch32.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes mematTrs;

    if PSTATE.EL == EL2 then
        mair = HMAIR1:HMAIR0;
    else
        mair = MAIR1:MAIR0;
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();

    if attrfield<7:4> == '0000' then           // Device
        mematTrs.type = MemType_Device;
        case attrfield<3:0> of
            when '0000'  mematTrs.device = DeviceType_nGnRnE;
            when '0100'  mematTrs.device = DeviceType_nGnRE;
            when '1000'  mematTrs.device = DeviceType_nGRE;
            when '1100'  mematTrs.device = DeviceType_GRE;
            otherwise    Unreachable();        // Reserved, handled above

    elsif attrfield<3:0> != '0000'  then       // Normal
        mematTrs.type = MemType_Normal;
        mematTrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        mematTrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        mematTrs.shareable = SH<1> == '1';
        mematTrs.outershareable = SH == '10';

    else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        Unreachable();                          // Reserved, handled above

    return MemAttrDefaults(mematTrs);
```

### aarch32/translation/attrs/AArch32.TranslateAddressS1Off

```
// AArch32.TranslateAddressS1Off()
// ===============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch32.TranslateAddressS1Off(bits(32) vaddress, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
        default_cacheable = (dc == '1');
    else
        default_cacheable = FALSE;

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.mematTrs.type = MemType_Normal;
        result.addrdesc.mematTrs.inner.attrs = MemAttr_WB;      // Write-back
        result.addrdesc.mematTrs.inner.hints = MemHint_RWA;
        result.addrdesc.mematTrs.shareable = FALSE;
        result.addrdesc.mematTrs.outershareable = FALSE;
    elsif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.mematTrs.type = MemType_Device;
        result.addrdesc.mematTrs.device = DeviceType_nGnRnE;
        result.addrdesc.mematTrs.inner = MemAttrHints UNKNOWN;
    else
        // Instruction cacheability controlled by SCTLR/HSCTLR.I
        if PSTATE.EL == EL2 then
            cacheable = HSCTLR.I == '1';
        else
            cacheable = SCTLR.I == '1';
        result.addrdesc.mematTrs.type = MemType_Normal;
        if cacheable then
            result.addrdesc.mematTrs.inner.attrs = MemAttr_WT;
            result.addrdesc.mematTrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.mematTrs.inner.attrs = MemAttr_NC;
            result.addrdesc.mematTrs.inner.hints = MemHint_No;
        result.addrdesc.mematTrs.shareable = TRUE;
        result.addrdesc.mematTrs.outershareable = TRUE;

    result.addrdesc.mematTrs.outer = result.addrdesc.mematTrs.inner;

    result.addrdesc.mematTrs = MemAttrDefaults(result.addrdesc.mematTrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.physicaladdress = ZeroExtend(vaddress);
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    result.addrdesc.fault = AArch32.NoFault();

    return result;
```

### aarch32/translation/checks/AArch32.CheckDomain

```
// AArch32.CheckDomain()
// =====================

(boolean, FaultRecord) AArch32.CheckDomain(bits(4) domain, bits(32) vaddress, integer level,
                                           AccType acctype, boolean iswrite)

    index = 2 * UInt(domain);
    attrfield = DACR<index+1:index>;

    if attrfield == '10' then          // Reserved, maps to an allocated value
        // Reserved value maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits();

    if attrfield == '00' then
        fault = AArch32.DomainFault(domain, level, acctype, iswrite);
    else
        fault = AArch32.NoFault();

    permissioncheck = (attrfield == '01');

    return (permissioncheck, fault);
```

### aarch32/translation/checks/AArch32.CheckPermission

```
// AArch32.CheckPermission()
// =========================
// Function used for permission checking from AArch32 stage 1 translations

FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level,
                                    bits(4) domain, bit NS, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());

    if PSTATE.EL != EL2 then
        wxn = SCTLR.WXN == '1';
        if TTBCR.EAE == '1' || SCTLR.AFE == '1' || perms.ap<0> == '1' then
            priv_r = TRUE;
            priv_w = perms.ap<2> == '0';
            user_r = perms.ap<1> == '1';
            user_w = perms.ap<2:1> == '01';
        else
            priv_r = perms.ap<2:1> != '00';
            priv_w = perms.ap<2:1> == '01';
            user_r = perms.ap<1> == '1';
            user_w = FALSE;
        uwxn = SCTLR.UWXN == '1';
        user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
        priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
                   (priv_w && wxn) || (user_w && uwxn));
        ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2
        wxn = HSCTLR.WXN == '1';
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' then
        secure_instr_fetch = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;
        if secure_instr_fetch == '1' then xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(40) UNKNOWN;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch32.NoFault();
```

## aarch32/translation/checks/AArch32.CheckS2Permission

```
// AArch32.CheckS2Permission()
// ===========================
// Function used for permission checking from AArch32 stage 2 translations

FaultRecord AArch32.CheckS2Permission(Permissions perms, bits(32) vaddress, bits(40) ipaddress,
                                      integer level, AccType acctype, boolean iswrite,
                                      boolean s2fs1walk)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && PSTATE.EL != EL2;

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    xn = !r || perms.xn == '1';

    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
    elsif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch32.NoFault();
```

## aarch32/translation/debug/AArch32.CheckBreakpoint

```
// AArch32.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

    match = FALSE;
    mismatch = FALSE;

    for i = 0 to UInt(DBGDIDR.BRPs)
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_Breakpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

## aarch32/translation/debug/AArch32.CheckDebug

```
// AArch32.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch32.NoFault();

    d_side = (acctype != AccType_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool();

    if !d_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(vaddress, size);

    if fault.type == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch32.CheckBreakpoint(vaddress, size);

    if fault.type == Fault_None && !d_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(vaddress, size);

    return fault;
```

## aarch32/translation/debug/AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// ==========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// Vector Catch can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = AArch32.VCRMatch(vaddress);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
        match = ConstrainUnpredictableBool();

    if match && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_VectorCatch;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

### aarch32/translation/debug/AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = PSTATE.EL != EL0 && !(PSTATE.EL == EL1 && acctype == AccType_UNPRIV);

    for i = 0 to UInt(DBGDIDR.WRPs)
        match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elsif match && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

### aarch32/translation/faults/AArch32.AccessFlagFault

```
// AArch32.AccessFlagFault()
// =========================

FaultRecord AArch32.AccessFlagFault(bits(40) ipaddress, bits(4) domain, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_AccessFlag, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

### aarch32/translation/faults/AArch32.AddressSizeFault

```
// AArch32.AddressSizeFault()
// ==========================

FaultRecord AArch32.AddressSizeFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_AddressSize, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### aarch32/translation/faults/AArch32.AlignmentFault

```
// AArch32.AlignmentFault()
// ========================

FaultRecord AArch32.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch32.CreateFaultRecord(Fault_Alignment, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

### aarch32/translation/faults/AArch32.AsynchExternalAbort

```
// AArch32.AsynchExternalAbort()
// =============================
// Wrapper function for asynchronous external aborts

FaultRecord AArch32.AsynchExternalAbort(boolean parity, bit extflag)

    type = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(type, ipaddress,  domain, level, acctype, iswrite, extflag,
                                     debugmoe, secondstage, s2fs1walk);
```

### aarch32/translation/faults/AArch32.DebugFault

```
// AArch32.DebugFault()
// ====================

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_Debug, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

### aarch32/translation/faults/AArch32.DomainFault

```
// AArch32.DomainFault()
// =====================

FaultRecord AArch32.DomainFault(bits(4) domain, integer level, AccType acctype, boolean iswrite)

    ipaddress = bits(40) UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    secondstage = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_Domain, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

### aarch32/translation/faults/AArch32.NoFault

```
// AArch32.NoFault()
// =================

FaultRecord AArch32.NoFault()

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_None, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

### aarch32/translation/faults/AArch32.PermissionFault

```
// AArch32.PermissionFault()
// =========================

FaultRecord AArch32.PermissionFault(bits(40) ipaddress, bits(4) domain, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_Permission, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

### aarch32/translation/faults/AArch32.TranslationFault

```
// AArch32.TranslationFault()
// ==========================

FaultRecord AArch32.TranslationFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_Translation, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, secondstage, s2fs1walk);
```

### aarch32/translation/translation/AArch32.FirstStageTranslate

```
// AArch32.FirstStageTranslate()
// =============================
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.FirstStageTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                              boolean wasaligned, integer size)

    if PSTATE.EL == EL2 then
        s1_enabled = HSCTLR.M == '1';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    elsif HaveEL(EL2) && !IsSecure() then
        tge = (if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE);
        dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
        s1_enabled = tge == '0' && dc == '0' && SCTLR.M == '1';
    else
        dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
        s1_enabled = dc == '0' && SCTLR.M == '1';

    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                          // First stage enabled
        use_long_descriptor_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
        if use_long_descriptor_format then
            S1 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                                s2fs1walk, size);
            permissioncheck = TRUE;  domaincheck = FALSE;
        else
            S1 = AArch32.TranslationTableWalkSD(vaddress, acctype, iswrite, size);
            permissioncheck = TRUE;  domaincheck = TRUE;
    else
        S1 = AArch32.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;  domaincheck = FALSE;

    // Check for unaligned data accesses to Device memory
    if (!wasaligned && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
        acctype != AccType_IFETCH) then
        S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

    if !IsFault(S1.addrdesc) && domaincheck then
        (permissioncheck, abort) = AArch32.CheckDomain(S1.domain, vaddress, S1.level, acctype,
                                                       iswrite);
        S1.addrdesc.fault = abort;

    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch32.CheckPermission(S1.perms, vaddress, S1.level,
                                                    S1.domain, S1.addrdesc.paddress.NS,
                                                    acctype, iswrite);

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
        acctype == AccType_IFETCH) then
        S1.addrdesc = AArch32.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                                S1.domain, acctype, iswrite,
                                                secondstage, s2fs1walk);

    return S1.addrdesc;
```

## aarch32/translation/translation/AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch32.FullTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch32.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !IsFault(S1) && HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        s2fs1walk = FALSE;
        result = AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                                            size);
    else
        result = S1;

    return result;
```

### aarch32/translation/translation/AArch32.SecondStageTranslate

```
// AArch32.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.SecondStageTranslate(AddressDescriptor S1, bits(32) vaddress,
                                               AccType acctype, boolean iswrite, boolean wasaligned,
                                               boolean s2fs1walk, integer size)
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert IsZero(S1.paddress.physicaladdress<47:40>);

    if !ELUsingAArch32(EL2) then
        return AArch64.SecondStageTranslate(S1, ZeroExtend(vaddress, 64), acctype, iswrite,
                                            wasaligned, s2fs1walk, size);

    s2_enabled = HCR.VM == '1' || HCR.DC == '1';
    secondstage = TRUE;

    if s2_enabled then                          // Second stage enabled
        ipaddress = S1.paddress.physicaladdress<39:0>;
        S2 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                            s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if (!wasaligned && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype != AccType_IFETCH) then
            S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch32.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                          acctype, iswrite, s2fs1walk);

        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype == AccType_IFETCH) then
            domain = bits(4) UNKNOWN;
            S2.addrdesc = AArch32.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    domain, acctype, iswrite,
                                                    secondstage, s2fs1walk);

        // Check for protected table walk
        if (s2fs1walk && !IsFault(S2.addrdesc) && HCR.PTW == '1' &&
            S2.addrdesc.memattrs.type == MemType_Device) then
            domain = bits(4) UNKNOWN;
            S2.addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, S2.level, acctype,
                                                        iswrite, secondstage, s2fs1walk);
        result = CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

    return result;
```

### aarch32/translation/translation/AArch32.SecondStageWalk

```
// AArch32.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
AddressDescriptor AArch32.SecondStageWalk(AddressDescriptor S1, bits(32) vaddress, AccType acctype,
                                          boolean iswrite, integer size)
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                        size);
```

## aarch32/translation/translation/AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        return AArch64.TranslateAddress(ZeroExtend(vaddress, 64), acctype, iswrite, wasaligned,
                                        size);

    result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
```

## aarch32/translation/walk/AArch32.TranslationTableWalkLD

```
// AArch32.TranslationTableWalkLD()
// ================================
// Returns a result of a translation table walk using the Long-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkLD(bits(40) ipaddress, bits(32) vaddress,
                                         AccType acctype, boolean iswrite, boolean secondstage,
                                         boolean s2fs1walk, integer size)
    if !secondstage then
        assert ELUsingAArch32(S1TranslationRegime());
    else
        assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && PSTATE.EL != EL2;

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(40) inputaddr;         // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2
    domain = bits(4) UNKNOWN;

    descaddr.memattrs.type = MemType_Normal;

    // Fixed parameters for the page table walk:
    //  grainsize = Log2(Size of Table)         - Size of Table is 4KB in AArch32
    //  stride = Log2(Address per Level)        - Bits of address consumed at each level
    constant integer grainsize = 12;                    // Log2(4KB page size)
    constant integer stride = grainsize - 3;            // Log2(page size / 8 bytes)

    // Derived parameters for the page table walk:
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
//  inputsize = Log2(Size of Input Address) - Input Address size in bits
//  level = Level to start walk from
// This means that the number of levels after start level = 3-level

if !secondstage then
    // First stage translation
    inputaddr = ZeroExtend(vaddress);
    if PSTATE.EL == EL2 then
        inputsize = 32 - UInt(HTCR.T0SZ);
        basefound = inputsize == 32 || IsZero(inputaddr<31:inputsize>);
        disabled = FALSE;
        baseregister = HTTBR;
        descaddr.memattrs = WalkAttrDecode(HTCR.SH0, HTCR.ORGN0, HTCR.IRGN0);
        reversedescriptors = HSCTLR.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;

    else
        basefound = FALSE;
        disabled = FALSE;
        t0size = UInt(TTBCR.T0SZ);
        if t0size == 0 || IsZero(inputaddr<31:(32-t0size)>) then
            inputsize = 32 - t0size;
            basefound = TRUE;
            disabled = TTBCR.EPD0 == '1';
            baseregister = TTBR0;
            descaddr.memattrs = WalkAttrDecode(TTBCR.SH0, TTBCR.ORGN0, TTBCR.IRGN0);
        t1size = UInt(TTBCR.T1SZ);
        if (t1size == 0 && !basefound) || (t1size > 0 && IsOnes(inputaddr<31:(32-t1size)>)) then
            inputsize = 32 - t1size;
            basefound = TRUE;
            disabled = TTBCR.EPD1 == '1';
            baseregister = TTBR1;
            descaddr.memattrs = WalkAttrDecode(TTBCR.SH1, TTBCR.ORGN1, TTBCR.IRGN1);
        reversedescriptors = SCTLR.EE == '1';
        lookupsecure = IsSecure();
        singlepriv = FALSE;

    // The starting level is the number of strides needed to consume the input address
    level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));

else
    // Second stage translation
    inputaddr = ipaddress;
    inputsize = 32 - SInt(VTCR.T0SZ);
    // VTCR.S must match VTCR.T0SZ[3]
    if VTCR.S != VTCR.T0SZ<3> then
        (-, inputsize) = ConstrainUnpredictableInteger(32-7, 32+8);
    basefound = inputsize == 40 || IsZero(inputaddr<39:inputsize>);
    disabled = FALSE;
    baseregister = VTTBR;
    descaddr.memattrs = WalkAttrDecode(VTCR.IRGN0, VTCR.ORGN0, VTCR.SH0);
    reversedescriptors = HSCTLR.EE == '1';
    lookupsecure = FALSE;
    singlepriv = TRUE;

    startlevel = UInt(VTCR.SL0);
    level = 2 - startlevel;
    if level <= 0 then basefound = FALSE;

    // Number of entries in the starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    startsizecheck = inputsize - ((3 - level)*stride + grainsize); // Log2(Num of entries)

    // Check for starting level table with fewer than 2 entries or longer than 16 pages.
    // Lower bound check is:  startsizecheck < Log2(2 entries)
    //   That is, VTCR.SL0 == '00' and SInt(VTCR.T0SZ) > 1, Size of Input Address < 2^31 bytes
    // Upper bound check is:  startsizecheck > Log2(pagesize/8*16)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    //   That is, VTCR.SL0 == '01' and SInt(VTCR.T0SZ) < -2, Size of Input Address > 2^34 bytes
    if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

if !basefound || disabled then
    level = 1;           // AArch64 reports this as a level 0 fault
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

if !IsZero(baseregister<47:40>) then
    level = 0;
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

// Bottom bound of the Base address is:
//     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
//     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
baseaddress = baseregister<39:baselowerbound>:Zeros(baselowerbound);

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

addrselecttop = inputsize - 1;

repeat
    addrselectbottom = (3-level)*stride + grainsize;

    bits(40) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
    descaddr.paddress.physicaladdress = ZeroExtend(baseaddress OR index);
    descaddr.paddress.NS = ns_table;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if !HaveEL(EL2) || secondstage || IsSecure() || PSTATE.EL == EL2 then
        descaddr2 = descaddr;
    else
        descaddr2 = AArch32.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
        // Check for a fault on the stage 2 walk
        if IsFault(descaddr2) then
            result.addrdesc.fault = descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    descaddr2.vaddress = ZeroExtend(vaddress);

    desc = _Mem[descaddr2, 8, AccType_PTW];
    if reversedescriptors then desc = BigEndianReverse(desc);

    if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
         // Fault (00), Reserved (10), or Block (01) at level 3
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Valid Block, Page, or Table entry
    if desc<1:0> == '01' || level == 3 then                 // Block (01) or Page (11)
        blocktranslate = TRUE;
    else                                                    // Table (11)
        if !IsZero(desc<47:40>) then
            result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        baseaddress = desc<39:grainsize>:Zeros(grainsize);

        if !secondstage then
            // Unpack the upper and lower table attributes
            ns_table    = ns_table    OR desc<63>;
            ap_table<1> = ap_table<1> OR desc<62>;       // read-only
            xn_table    = xn_table    OR desc<60>;
            // pxn_table and ap_table[0] apply only in EL1&0 translation regimes
            if !singlepriv then
                ap_table<0> = ap_table<0> OR desc<61>;   // privileged
                pxn_table   = pxn_table   OR desc<59>;

        level = level + 1;
        addrselecttop = addrselectbottom - 1;
        blocktranslate = FALSE;
until blocktranslate;

// Check the output address is inside the supported range
if !IsZero(desc<47:40>) then
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype,
                                                     iswrite, secondstage, s2fs1walk);
    return result;

// Check the access flag
if desc<10> == '0' then
    result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);
    return result;

// Unpack the descriptor into address and upper and lower block attributes
outputaddress = desc<39:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
xn = desc<54>;
pxn = desc<53>;
contiguousbit = desc<52>;
nG = desc<11>;
sh = desc<9:8>;
ap = desc<7:6>:'1';
memattr = desc<5:2>;                                    // AttrIndx and NS bit in stage 1

result.domain = bits(4) UNKNOWN;                        // Domains not used
result.level = level;
result.blocksize = 2^((3-level)*stride + grainsize);

// Stage 1 translation regimes also inherit attributes from the tables
if !secondstage then
    result.perms.xn      = xn OR xn_table;
    result.perms.ap<2>   = ap<2> OR ap_table<1>;        // Force read-only

    // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
    if !singlepriv then
        result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
        result.perms.pxn   = pxn OR pxn_table;
        // Pages from Non-secure tables are marked non-global in Secure EL1&0
        if IsSecure() then
            result.nG = nG OR ns_table;
        else
            result.nG = nG;
    else
        result.perms.ap<1> = '1';
        result.perms.pxn   = '0';
        result.nG          = '0';
    result.perms.ap<0>   = '1';
    result.addrdesc.memattrs = AArch32.S1AttrDecode(sh, memattr<2:0>, acctype);
    result.addrdesc.paddress.NS = memattr<3> OR ns_table;
else
    result.perms.ap<2:1> = ap<2:1>;
    result.perms.ap<0>   = '1';
    result.perms.xn      = xn;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result.perms.pxn     = '0';
        result.nG            = '0';
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
        result.addrdesc.paddress.NS = '1';

    result.addrdesc.paddress.physicaladdress = ZeroExtend(outputaddress);
    result.addrdesc.fault = AArch32.NoFault();
    result.contiguous = contiguousbit == '1';

    return result;
```

## aarch32/translation/walk/AArch32.TranslationTableWalkSD

```
// AArch32.TranslationTableWalkSD()
// ================================
// Returns a result of a translation table walk using the Short-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkSD(bits(32) vaddress, AccType acctype, boolean iswrite,
                                         integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    // This is only called when address translation is enabled
    TLBRecord         result;
    AddressDescriptor l1descaddr;
    AddressDescriptor l2descaddr;
    bits(40)      outputaddress;

    // Variables for Abort functions
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    // Default setting of the domain
    domain = bits(4) UNKNOWN;

    // Determine correct Translation Table Base Register to use.
    bits(64) ttbr;
    n = UInt(TTBCR.N);
    if n == 0 || IsZero(vaddress<31:(32-n)>) then
        ttbr = TTBR0;
        disabled = (TTBCR.PD0 == '1');
    else
        ttbr = TTBR1;
        disabled = (TTBCR.PD1 == '1');
        n = 0;  // TTBR1 translation always works like N=0 TTBR0 translation

    // Check this Translation Table Base Register is not disabled.
    if disabled then
        level = 1;
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    // Obtain descriptor from initial lookup.
    l1descaddr.paddress.physicaladdress = ZeroExtend(ttbr<31:14-n>:vaddress<31-n:20>:'00');
    l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
    IRGN = ttbr<0>:ttbr<6>;             // TTBR.IRGN
    RGN = ttbr<4:3>;                    // TTBR.RGN
    SH = ttbr<1>:ttbr<5>;               // TTBR.S:TTBR.NOS
    l1descaddr.memattrs = WalkAttrDecode(SH, RGN, IRGN);

    if !HaveEL(EL2) || IsSecure()  then
        // if only 1 stage of translation
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    l1descaddr2 = l1descaddr;
else
    l1descaddr2 = AArch32.SecondStageWalk(l1descaddr, vaddress, acctype, iswrite, 4);
    // Check for a fault on the stage 2 walk
    if IsFault(l1descaddr2) then
        result.addrdesc.fault = l1descaddr2.fault;
        return result;

// Update virtual address for abort functions
l1descaddr2.vaddress = ZeroExtend(vaddress);

l1desc = _Mem[l1descaddr2, 4, AccType_PTW];
if SCTLR.EE == '1' then l1desc = BigEndianReverse(l1desc);

// Process descriptor from initial lookup.
case l1desc<1:0> of
    when '00'                                               // Fault, Reserved
        level = 1;
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    when '01'                                               // Large page or Small page
        domain = l1desc<8:5>;
        level = 2;
        pxn = l1desc<2>;
        NS = l1desc<3>;

        // Obtain descriptor from level 2 lookup.
        l2descaddr.paddress.physicaladdress = ZeroExtend(l1desc<31:10>:vaddress<19:12>:'00');
        l2descaddr.paddress.NS = if IsSecure() then '0' else '1';
        l2descaddr.memattrs = l1descaddr.memattrs;

        if !HaveEL(EL2) || IsSecure()  then
         // if only 1 stage of translation
            l2descaddr2 = l2descaddr;
        else
            l2descaddr2 = AArch32.SecondStageWalk(l2descaddr, vaddress, acctype, iswrite, 4);
            // Check for a fault on the stage 2 walk
            if IsFault(l2descaddr2) then
                result.addrdesc.fault = l2descaddr2.fault;
                return result;

        // Update virtual address for abort functions
        l2descaddr2.vaddress = ZeroExtend(vaddress);

        l2desc = _Mem[l2descaddr2, 4, AccType_PTW];
        if SCTLR.EE == '1' then l2desc = BigEndianReverse(l2desc);

        // Process descriptor from level 2 lookup.
        if l2desc<1:0> == '00' then
            result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        nG = l2desc<11>;
        S = l2desc<10>;
        ap = l2desc<9,5:4>;

        if SCTLR.AFE == '1' && l2desc<4> == '0' then
            // Hardware access to the Access Flag is not supported in ARMv8
            result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                            iswrite, secondstage, s2fs1walk);
            return result;

        if l2desc<1> == '0' then                            // Large page
            xn = l2desc<15>;
            tex = l2desc<14:12>;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                c = l2desc<3>;
                b = l2desc<2>;
                blocksize = 64;
                outputaddress = ZeroExtend(l2desc<31:16>:vaddress<15:0>);
            else                                                    // Small page
                tex = l2desc<8:6>;
                c = l2desc<3>;
                b = l2desc<2>;
                xn = l2desc<0>;
                blocksize = 4;
                outputaddress = ZeroExtend(l2desc<31:12>:vaddress<11:0>);

        when '1x'                                                   // Section or Supersection
            NS = l1desc<19>;
            nG = l1desc<17>;
            S = l1desc<16>;
            ap = l1desc<15,11:10>;
            tex = l1desc<14:12>;
            xn = l1desc<4>;
            c = l1desc<3>;
            b = l1desc<2>;
            pxn = l1desc<0>;
            level = 1;

            if SCTLR.AFE == '1' && l1desc<10> == '0' then
                // Hardware management of the Access Flag is not supported in ARMv8
                result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                                iswrite, secondstage, s2fs1walk);
                return result;

            if l1desc<18> == '0' then                               // Section
                domain = l1desc<8:5>;
                blocksize = 1024;
                outputaddress = ZeroExtend(l1desc<31:20>:vaddress<19:0>);
            else                                                    // Supersection
                domain = '0000';
                blocksize = 16384;
                outputaddress = l1desc<8:5>:l1desc<23:20>:l1desc<31:24>:vaddress<23:0>;

    // Decode the TEX, C, B and S bits to produce the TLBRecord's memory attributes
    if SCTLR.TRE == '0' then
        if RemapRegsHaveResetValues() then
            result.addrdesc.memattrs = AArch32.DefaultTEXDecode(tex, c, b, S, acctype);
        else
            result.addrdesc.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        result.addrdesc.memattrs = AArch32.RemappedTEXDecode(tex, c, b, S, acctype);

    // Set the rest of the TLBRecord, try to add it to the TLB, and return it.
    result.perms.ap = ap;
    result.perms.xn = xn;
    result.perms.pxn = pxn;
    result.nG = nG;
    result.domain = domain;
    result.level = level;
    result.blocksize = blocksize;
    result.addrdesc.paddress.physicaladdress = ZeroExtend(outputaddress);
    result.addrdesc.paddress.NS = if IsSecure() then NS else '1';
    result.addrdesc.fault = AArch32.NoFault();

    return result;
```

### aarch32/translation/walk/RemapRegsHaveResetValues

```
boolean RemapRegsHaveResetValues();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## J1.3 Shared pseudocode

This section holds the pseudocode that is common to execution in AArch64 state and in AArch32 state. Functions listed in this section are identified only by a `FunctionName`, without an `AArch64.` or `AArch32.` prefix. This section is organised by functional groups, with the functional groups being indicated by hierarchical path names, for example `shared/debug/DebugTarget`.

The top-level sections of the shared pseudocode hierarchy are:

- *shared/debug*.
- *shared/exceptions* on page J1-5360.
- *shared/functions* on page J1-5362.
- *shared/translation* on page J1-5413.

### J1.3.1 shared/debug

This section includes the following pseudocode functions:

- *shared/debug/ClearStickyErrors/ClearStickyErrors* on page J1-5345.
- *shared/debug/DebugTarget/DebugTarget* on page J1-5345.
- *shared/debug/DebugTarget/DebugTargetFrom* on page J1-5345.
- *shared/debug/DoubleLockStatus/DoubleLockStatus* on page J1-5346.
- *shared/debug/FindWatchpoint/FindWatchpoint* on page J1-5346.
- *shared/debug/authentication/AllowExternalDebugAccess* on page J1-5346.
- *shared/debug/authentication/AllowExternalPMUAccess* on page J1-5346.
- *shared/debug/authentication/Debug_authentication* on page J1-5347.
- *shared/debug/authentication/ExternalInvasiveDebugEnabled* on page J1-5347.
- *shared/debug/authentication/ExternalNoninvasiveDebugEnabled* on page J1-5347.
- *shared/debug/authentication/ExternalSecureInvasiveDebugEnabled* on page J1-5347.
- *shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled* on page J1-5347.
- *shared/debug/cti/CTI_SetEventLevel* on page J1-5348.
- *shared/debug/cti/CTI_SignalEvent* on page J1-5348.
- *shared/debug/cti/CrossTrigger* on page J1-5348.
- *shared/debug/dccanditr/CheckForDCCInterrupts* on page J1-5348.
- *shared/debug/dccanditr/DBGDTRRX_EL0* on page J1-5348.
- *shared/debug/dccanditr/DBGDTRTX_EL0* on page J1-5349.
- *shared/debug/dccanditr/DBGDTR_EL0* on page J1-5350.
- *shared/debug/dccanditr/DTR* on page J1-5351.
- *shared/debug/dccanditr/EDITR* on page J1-5351.
- *shared/debug/halting/DCPSInstruction* on page J1-5351.
- *shared/debug/halting/DRPSInstruction* on page J1-5352.
- *shared/debug/halting/DebugHalt* on page J1-5352.
- *shared/debug/halting/DisableITRAndResumeInstructionPrefetch* on page J1-5353.
- *shared/debug/halting/ExecuteA64* on page J1-5353.
- *shared/debug/halting/ExecuteT32* on page J1-5353.
- *shared/debug/halting/ExitDebugState* on page J1-5353.
- *shared/debug/halting/Halt* on page J1-5353.
- *shared/debug/halting/HaltOnBreakpointOrWatchpoint* on page J1-5354.
- *shared/debug/halting/Halted* on page J1-5354.
- *shared/debug/halting/HaltingAllowed* on page J1-5354.
- *shared/debug/halting/Restarting* on page J1-5354.
- *shared/debug/halting/StopInstructionPrefetchAndEnableITR* on page J1-5355.
- *shared/debug/halting/UpdateEDSCRFields* on page J1-5355.
- *shared/debug/haltingevents/CheckExceptionCatch* on page J1-5355.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/debug/haltingevents/CheckHaltingStep* on page J1-5355.
- *shared/debug/haltingevents/CheckOSUnlockCatch* on page J1-5356.
- *shared/debug/haltingevents/CheckPendingOSUnlockCatch* on page J1-5356.
- *shared/debug/haltingevents/CheckPendingResetCatch* on page J1-5356.
- *shared/debug/haltingevents/CheckResetCatch* on page J1-5356.
- *shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters* on page J1-5356.
- *shared/debug/haltingevents/ExternalDebugRequest* on page J1-5356.
- *shared/debug/haltingevents/HaltingStep_DidNotStep* on page J1-5357.
- *shared/debug/haltingevents/HaltingStep_SteppedEX* on page J1-5357.
- *shared/debug/haltingevents/RunHaltingStep* on page J1-5357.
- *shared/debug/interrupts/InterruptID* on page J1-5357.
- *shared/debug/interrupts/SetInterruptRequestLevel* on page J1-5357.
- *shared/debug/samplebasedprofiling/CreatePCSample* on page J1-5357.
- *shared/debug/samplebasedprofiling/EDPCSRlo* on page J1-5358.
- *shared/debug/samplebasedprofiling/PCSample* on page J1-5358.
- *shared/debug/softwarestep/CheckSoftwareStep* on page J1-5358.
- *shared/debug/softwarestep/DebugExceptionReturnSS* on page J1-5359.
- *shared/debug/softwarestep/SSAdvance* on page J1-5359.
- *shared/debug/softwarestep/SoftwareStep_DidNotStep* on page J1-5359.
- *shared/debug/softwarestep/SoftwareStep_SteppedEX* on page J1-5360.


### shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';            // Clear TX underrun flag
    EDSCR.RXO = '0';            // Clear RX overrun flag
    if Halted() then            // in Debug state
        EDSCR.ITO = '0';        // Clear ITR overrun flag
    EDSCR.ERR = '0';            // Clear cumulative error flag
    return;
```

### shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =============

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

### shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =================
// Returns the debug exception target Exception level

bits(2) DebugTargetFrom(boolean secure)

    if HaveEL(EL2) && !secure then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

### shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the value of EDPRSR.DLK.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()

    if ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

### shared/debug/FindWatchpoint/FindWatchpoint

```
// FindWatchpoint()
// ================

integer FindWatchpoint()
    address = FAR[];
    base = Align(address, ZVAGranuleSize());
    limit = base + ZVAGranuleSize();
    repeat
        for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
            if WatchpointByteMatch(i, address) then     // Candidate found
                return i;
        address = address + 1;
        if address == limit then address = base;        // Wrap round, as this must be a DC ZVA
    while address != FAR[];
    return -1;                                          // No candidate found (should not happen)
```

### shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns the status of EDPRSR.EDAD.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if ExternalInvasiveDebugEnabled() then
        if ExternalSecureInvasiveDebugEnabled() then
            return TRUE;
        elsif HaveEL(EL3) then
            return (if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD) == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

### shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// ========================
// Returns the status of EDPRSR.EPMAD.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean AllowExternalPMUAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if ExternalNoninvasiveDebugEnabled() then
        if ExternalSecureNoninvasiveDebugEnabled() then
            return TRUE;
        elsif HaveEL(EL3) then
            return (if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD) == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

### shared/debug/authentication/Debug_authentication

```
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
```

### shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================

boolean ExternalInvasiveDebugEnabled()
    // In the recommended interface, ExternalInvasiveDebugEnabled returns the state of the DBGEN
    // signal.
    return DBGEN == HIGH;
```

### shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =================================

boolean ExternalNoninvasiveDebugEnabled()
    // In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN
    // OR NIDEN) signal.
    return ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

### shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================

boolean ExternalSecureInvasiveDebugEnabled()
    // In the recommended interface, ExternalSecureInvasiveDebugEnabled returns the state of the
    // (DBGEN AND SPIDEN) signal.
    // CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

### shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================

boolean ExternalSecureNoninvasiveDebugEnabled()
    // In the recommended interface, ExternalSecureNoninvasiveDebugEnabled returns the state of the
    // (DBGEN OR NIDEN) AND (SPIDEN OR SPNIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/debug/cti/CTI_SetEventLevel

```
CTI_SetEventLevel(CrossTriggerIn id, signal level);
```

### shared/debug/cti/CTI_SignalEvent

```
CTI_SignalEvent(CrossTriggerIn id);
```

### shared/debug/cti/CrossTrigger

```
enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,  CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,  CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,     CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,         CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,  CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,  CrossTriggerIn_TraceExtOut3};
```

### shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
               (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
```

### shared/debug/dccanditr/DBGDTRRX_EL0

```
// DBGDTRRX_EL0[] (external write)
// ===============================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xD5330501<31:0>);               // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401<31:0>);               // A64 "STR W1,[X0],#4"
            X[1] = bits(64) UNKNOWN;
        else
            ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
            R[1] = bits(32) UNKNOWN;

        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(32) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
            assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_EL0[] (external read)
// ==============================

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;
```

## shared/debug/dccanditr/DBGDTRTX_EL0

```
// DBGDTRTX_EL0[] (external read)
// ==============================
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then           // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN

    EDSCR.TXfull = '0';

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"

        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(32) UNKNOWN;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';

        if !UsingAArch32() then
            X[1] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// ===============================

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write
    DTRTX = value;
    return;
```

## shared/debug/dccanditr/DBGDTR_EL0

```
// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ===================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

### shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x088.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if !Halted() then return;                           // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';              // Overrun condition: block write
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```

### shared/debug/halting/DCPSInstruction

```
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
            elsif HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then UndefinedFault();
            else handle_el = EL1;

        when EL2
            if !HaveEL(EL2) then UndefinedFault();
            elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
            elsif IsSecure() then UndefinedFault();
            else handle_el = EL2;

        when EL3
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            if EDSCR.SDD == '1' || !HaveEL(EL3) then UndefinedFault();
            handle_el = EL3;

    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32();                  // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1  AArch32.WriteMode(M32_Svc);
            when EL2  AArch32.WriteMode(M32_Hyp);
            when EL3  AArch32.WriteMode(M32_Monitor);
        if handle_el == EL2 then
            ELR_hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
        else
            LR = bits(32) UNKNOWN;
        SPSR[] = bits(32) UNKNOWN;
        PSTATE.E = SCTLR[].EE;
    else                                        // Targeting AArch64
        if UsingAArch32() then AArch64.MaybeZeroRegisterUppers();
        ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(32) UNKNOWN;  ESR[] = bits(32) UNKNOWN;
        PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;

    DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    return;
```

### shared/debug/halting/DRPSInstruction

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    SetPSTATEFromPSR(SPSR[]);

    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
    else
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;

    UpdateEDSCRFields();                            // Update EDSCR processor state flags.

    return;
```

### shared/debug/halting/DebugHalt

```
constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```
DisableITRAndResumeInstructionPrefetch();
```

### shared/debug/halting/ExecuteA64

```
ExecuteA64(bits(32) instr);
```

### shared/debug/halting/ExecuteT32

```
ExecuteT32(bits(16) hw1, bits(16) hw2);
```

### shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// ================

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the processor is restarting, debuggers must use EDPRSR.SDR
    // to detect that the processor has restarted.
    EDSCR.STATUS = '000001';                            // Signal restarting
    EDESR<2:0> = '000';                                 // Clear any pending Halting debug events

    new_pc = DLR_EL0;
    spsr = DSPSR;

    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr);                             // Can update privileged bits, even at EL0.

    if UsingAArch32() then
        if ConstrainUnpredictableBool() then new_pc<0> = '0';
        BranchTo(new_pc<31:0>, BranchType_UNKNOWN);     // AArch32 branch
    else
        // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
        if spsr<4> == '1' && ConstrainUnpredictableBool() then
            new_pc<63:32> = Zeros();
        BranchTo(new_pc, BranchType_DBGEXIT);           // A type of branch that is never predicted

    (EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');         // Atomically signal restarted
    UpdateEDSCRFields();                                // Stop signalling processor state.
    DisableITRAndResumeInstructionPrefetch();

    return;
```

### shared/debug/halting/Halt

```
// Halt()
// ======

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    DLR_EL0 = ThisInstrAddr();
    DSPSR_EL0 = GetPSRFromPSTATE();
    DSPSR_EL0.SS = PSTATE.SS;                   // Always save PSTATE.SS

    EDSCR.ITE = '1';  EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0';                        // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = (if ExternalSecureInvasiveDebugEnabled() then '0' else '1');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        assert EDSCR.SDD == '1';                // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if
    // UNKNOWN. PSTATE.{N,Z,C,V,Q,GE} are also not observable, but since these are not changed on
    // exception entry, this function also leaves them unchanged. PSTATE.{E,M,nRW,EL,SP} are
    // unchanged. PSTATE.IL is set to 0.
    if UsingAArch32() then
        PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1'; // PSTATE.J is RES0
    else
        PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                      // Signal entered Debug state
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    return;
```

### shared/debug/halting/HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

### shared/debug/halting/Halted

```
// Halted()
// ========

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});                     // Halted
```

### shared/debug/halting/HaltingAllowed

```
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elsif IsSecure() then
        return ExternalSecureInvasiveDebugEnabled();
    else
        return ExternalInvasiveDebugEnabled();
```

### shared/debug/halting/Restarting

```
// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';                                    // Restarting
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/debug/halting/StopInstructionPrefetchAndEnableITR

```
StopInstructionPrefetchAndEnableITR();
```

### shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// ===================
// Update EDSCR processor state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;
        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        EDSCR.NS = if IsSecure() then '0' else '1';
        EDSCR.RW<1> = (if ELUsingAArch32(EL1) then '0' else '1');
        if PSTATE.EL != EL0 then
            EDSCR.RW<0> = EDSCR.RW<1>;
        else
            EDSCR.RW<0> = (if UsingAArch32() then '0' else '1');
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0') then
            EDSCR.RW<2> = EDSCR.RW<1>;
        else
            EDSCR.RW<2> = (if ELUsingAArch32(EL2) then '0' else '1');
        if !HaveEL(EL3) then
            EDSCR.RW<3> = EDSCR.RW<2>;
        else
            EDSCR.RW<3> = (if ELUsingAArch32(EL3) then '0' else '1');

    return;
```

### shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch()
    // Called after taking an exception, that is, such that IsSecure() and PSTATE.EL are correct
    // for the exception target.
    base = if IsSecure() then 0 else 4;
    if HaltingAllowed() && EDECCR<UInt(PSTATE.EL) + base> == '1' then
        Halt(DebugHalt_ExceptionCatch);
```

### shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if EDECR.OSUCE == '1' && !Halted() then EDESR.OSUC = '1';
```

### shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);
```

### shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);
```

### shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if EDECR.RCE == '1' then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

### shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()

    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

### shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt(DebugHalt_EDBGRQ);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/debug/haltingevents/HaltingStep_DidNotStep

```
boolean HaltingStep_DidNotStep();
```

### shared/debug/haltingevents/HaltingStep_SteppedEX

```
boolean HaltingStep_SteppedEX();
```

### shared/debug/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// ================

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    // if "exception_generated" == TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exeception.
    // "reset" = TRUE if exiting reset state into the highest EL.
    if reset then assert !Halted();             // Cannot come out of reset halted

    active = EDECR.SS == '1' && !Halted();

    if active && reset then                     // Coming out of reset with EDECR.SS set.
        EDESR.SS = '1';
    elsif active && HaltingAllowed() then
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

### shared/debug/interrupts/InterruptID

```
enumeration InterruptID {InterruptID_PMUIRQ, InterruptID_COMMIRQ, InterruptID_CTIIRQ,
                         InterruptID_COMMRX, InterruptID_COMMTX};
```

### shared/debug/interrupts/SetInterruptRequestLevel

```
SetInterruptRequestLevel(InterruptID id, signal level);
```

### shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// ================

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.
    enabled = (if IsSecure() then ExternalSecureNoninvasiveDebugEnabled()
                             else ExternalNoninvasiveDebugEnabled());

    pc_sample.valid = enabled && !Halted();
    pc_sample.pc = ThisInstrAddr();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ns = if IsSecure() then '0' else '1';
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1;
    if HaveEL(EL2) && !IsSecure() then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        pc_sample.vmid = if ELUsingAArch32(EL2) then VTTBR.VMID else VTTBR_EL2.VMID;

    return;
```

### shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// ================

bits(32) EDPCSRlo[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || EDLSR.SLK == '0';        // Software locked: no side-effects

    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            EDVIDSR.VMID = (if HaveEL(EL2) && pc_sample.ns == '1' && pc_sample.el IN {EL1,EL0}
                            then pc_sample.vmid else Zeros(8));
            EDVIDSR.NS = pc_sample.ns;
            EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
            EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
            // The conditions for setting HV are not specified if PCSRhi is zero.
            // An example implementation may be "pc_sample.rw".
            EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
    else
        sample = Ones(32);
        if update then
            EDPCSRhi = bits(32) UNKNOWN;
            EDCIDSR = bits(32) UNKNOWN;
            EDVIDSR = (bits(4) UNKNOWN):Zeros(20):(bits(8) UNKNOWN);

    return sample;
```

### shared/debug/samplebasedprofiling/PCSample

```
type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    bit ns,
    bits(32) contextidr,
    bits(8) vmid
)

PCSample pc_sample;
```

### shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    if !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() then
        if MDSCR_EL1.SS == '1' && PSTATE.SS == '0' then AArch64.SoftwareStepException();
```

### shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(32) spsr)
    assert Halted() || Restarting() ||  PSTATE.EL != EL0;

    SS_bit = '0';

    if MDSCR_EL1.SS == '1' then
        if Restarting() then
            enabled_at_source = FALSE;
        elsif UsingAArch32() then
            enabled_at_source = AArch32.GenerateDebugExceptions();
        else
            enabled_at_source = AArch64.GenerateDebugExceptions();

        if IllegalExceptionReturn(spsr) then
            dest = PSTATE.EL;
        else
            (valid, dest) = ELFromSPSR(spsr);  assert valid;

        secure = IsSecureBelowEL3() || dest == EL3;

        if ELUsingAArch32(dest) then
            enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
        else
            mask = spsr<9>;
            enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);

        ELd = DebugTargetFrom(secure);
        if !ELUsingAArch32(ELd) && !enabled_at_source && enabled_at_dest then
            SS_bit = spsr<21>;

    return SS_bit;
```

### shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

### shared/debug/softwarestep/SoftwareStep_DidNotStep

```
boolean SoftwareStep_DidNotStep();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/debug/softwarestep/SoftwareStep_SteppedEX

```
boolean SoftwareStep_SteppedEX();
```

## J1.3.2 shared/exceptions

This section includes the following pseudocode functions:

- *shared/exceptions/exceptions/Exception*.
- *shared/exceptions/exceptions/ExceptionRecord*.
- *shared/exceptions/exceptions/ExceptionSyndrome*.
- *shared/exceptions/traps/ReservedValue* on page J1-5361.
- *shared/exceptions/traps/UnallocatedEncoding* on page J1-5361.

### shared/exceptions/exceptions/Exception

```
enumeration Exception {Exception_Uncategorized,       // Uncategorized or unknown reason
                       Exception_WFxTrap,             // Trapped WFI or WFE instruction
                       Exception_CP15RTTrap,          // Trapped AArch32 MCR or MRC access to CP15
                       Exception_CP15RRTTrap,         // Trapped AArch32 MCRR or MRRC access to CP15
                       Exception_CP14RTTrap,          // Trapped AArch32 MCR or MRC access to CP14
                       Exception_CP14DTTrap,          // Trapped AArch32 LDC or STC access to CP14
                       Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
                       Exception_FPIDTrap,            // Trapped access to SIMD or FP ID register
                       // Trapped BXJ instruction not supported in ARMv8
                       Exception_CP14RRTTrap,         // Trapped MRRC access to CP14 from AArch32
                       Exception_IllegalState,        // Illegal Execution state
                       Exception_SupervisorCall,      // Supervisor Call
                       Exception_HypervisorCall,      // Hypervisor Call
                       Exception_MonitorCall,         // Monitor Call or Trapped SMC instruction
                       Exception_SystemRegisterTrap,  // Trapped MRS or MSR system register access
                       Exception_InstructionAbort,    // Instruction Abort or Prefetch Abort
                       Exception_PCAlignment,         // Misaligned PC
                       Exception_DataAbort,           // Data Abort
                       Exception_SPAlignment,         // Misaligned SP
                       Exception_FPTrappedException,  // IEEE trapped FP exception
                       Exception_SError,              // SError interrupt or Asynchronous Abort
                       Exception_Breakpoint,          // (Hardware) Breakpoint
                       Exception_SoftwareStep,        // Software Step
                       Exception_Watchpoint,          // Watchpoint
                       Exception_SoftwareBreakpoint,  // Software Breakpoint Instruction
                       Exception_VectorCatch,         // AArch32 Vector Catch
                       Exception_IRQ,                 // IRQ interrupt
                       Exception_FIQ};                // FIQ interrupt
```

### shared/exceptions/exceptions/ExceptionRecord

```
type ExceptionRecord is (Exception type,        // Exception class
                         bits(25) syndrome,     // Syndrome record
                         bits(64) vaddress,     // Virtual fault address
                         boolean ipavalid,      // Physical fault address is valid
                         bits(48) ipaddress)    // Physical fault address for second stage faults
```

### shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception type)

    ExceptionRecord r;

    r.type = type;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    r.syndrome = Zeros();
    if r.type IN {Exception_WFxTrap, Exception_CP15RTTrap, Exception_CP15RRTTrap,
                  Exception_CP14RTTrap, Exception_CP14DTTrap,
                  Exception_AdvSIMDFPAccessTrap, Exception_FPIDTrap,
                  Exception_CP14RRTTrap} then
        if UsingAArch32() then
            cond = AArch32.CurrentCond();
            if PSTATE.T == '0' then             // A32
                r.syndrome<24> = '1';
                // A conditional A32 instruction that is known to pass its condition code check
                // can be presented either with COND set to 0xE, the value for unconditional, or
                // the COND value held in the instruction.
                if ConditionHolds(cond) && ConstrainUnpredictableBool() then
                    r.syndrome<23:20> = '1110';
                else
                    r.syndrome<23:20> = cond;
            else                                // T32
                // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
                //  * CV set to 0 and COND is set to an UNKNOWN value
                //  * CV set to 1 and COND is set to the condition code for the condition that
                //    applied to the instruction.
                if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                    r.syndrome<24> = '1';
                    r.syndrome<23:20> = cond;
                else
                    r.syndrome<24> = '0';
                    r.syndrome<23:20> = bits(4) UNKNOWN;
        else
            r.syndrome<24> = '1';
            r.syndrome<23:20> = '1110';

    // Initialize all other fields
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.ipaddress = Zeros();

    return r;
```

## shared/exceptions/traps/ReservedValue

```
// ReservedValue()
// ===============

ReservedValue()

    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();
```

## shared/exceptions/traps/UnallocatedEncoding

```
// UnallocatedEncoding()
// =====================

UnallocatedEncoding()

    // If the unallocated encoding is an AArch32 CP10 or CP11 instruction, FPEXC.DEX must be written
    // to zero. This is omitted from this code.
    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### J1.3.3 shared/functions

This section includes the following pseudocode functions:


- *shared/functions/aborts/EncodeLDFSC* on page J1-5366.
- *shared/functions/aborts/FaultSyndrome* on page J1-5366.
- *shared/functions/aborts/IPAValid* on page J1-5367.
- *shared/functions/aborts/IsAsyncAbort* on page J1-5367.
- *shared/functions/aborts/IsDebugException* on page J1-5367.
- *shared/functions/aborts/IsExternalAbort* on page J1-5367.
- *shared/functions/aborts/IsFault* on page J1-5367.
- *shared/functions/aborts/IsSecondStage* on page J1-5368.
- *shared/functions/aborts/LSInstructionSyndrome* on page J1-5368.
- *shared/functions/common/ASR* on page J1-5368.
- *shared/functions/common/ASR_C* on page J1-5368.
- *shared/functions/common/Abs* on page J1-5368.
- *shared/functions/common/Align* on page J1-5368.
- *shared/functions/common/BitCount* on page J1-5369.
- *shared/functions/common/CountLeadingSignBits* on page J1-5369.
- *shared/functions/common/CountLeadingZeroBits* on page J1-5369.
- *shared/functions/common/Elem* on page J1-5369.
- *shared/functions/common/Extend* on page J1-5370.
- *shared/functions/common/HighestSetBit* on page J1-5370.
- *shared/functions/common/Int* on page J1-5370.
- *shared/functions/common/IsOnes* on page J1-5370.
- *shared/functions/common/IsZero* on page J1-5370.
- *shared/functions/common/IsZeroBit* on page J1-5370.
- *shared/functions/common/LSL* on page J1-5370.
- *shared/functions/common/LSL_C* on page J1-5371.
- *shared/functions/common/LSR* on page J1-5371.
- *shared/functions/common/LSR_C* on page J1-5371.
- *shared/functions/common/LowestSetBit* on page J1-5371.
- *shared/functions/common/Max* on page J1-5371.
- *shared/functions/common/Min* on page J1-5372.
- *shared/functions/common/NOT* on page J1-5372.
- *shared/functions/common/Ones* on page J1-5372.
- *shared/functions/common/ROR* on page J1-5372.
- *shared/functions/common/ROR_C* on page J1-5372.
- *shared/functions/common/Replicate* on page J1-5373.
- *shared/functions/common/RoundDown* on page J1-5373.
- *shared/functions/common/RoundTowardsZero* on page J1-5373.
- *shared/functions/common/RoundUp* on page J1-5373.
- *shared/functions/common/SInt* on page J1-5373.
- *shared/functions/common/SignExtend* on page J1-5373.
- *shared/functions/common/UInt* on page J1-5373.
- *shared/functions/common/ZeroExtend* on page J1-5374.
- *shared/functions/common/Zeros* on page J1-5374.
- *shared/functions/crc/BitReverse* on page J1-5374.
- *shared/functions/crc/HaveCRCExt* on page J1-5374.
- *shared/functions/crc/Poly32Mod2* on page J1-5374.
- *shared/functions/crypto/AESInvMixColumns* on page J1-5375.
- *shared/functions/crypto/AESInvShiftRows* on page J1-5375.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/crypto/AESInvSubBytes* on page J1-5375.
- *shared/functions/crypto/AESMixColumns* on page J1-5375.
- *shared/functions/crypto/AESShiftRows* on page J1-5375.
- *shared/functions/crypto/AESSubBytes* on page J1-5375.
- *shared/functions/crypto/HaveCryptoExt* on page J1-5375.
- *shared/functions/crypto/ROL* on page J1-5375.
- *shared/functions/crypto/SHA256hash* on page J1-5375.
- *shared/functions/crypto/SHAchoose* on page J1-5376.
- *shared/functions/crypto/SHAhashSIGMA0* on page J1-5376.
- *shared/functions/crypto/SHAhashSIGMA1* on page J1-5376.
- *shared/functions/crypto/SHAmajority* on page J1-5376.
- *shared/functions/crypto/SHAparity* on page J1-5376.
- *shared/functions/exclusive/ClearExclusiveByAddress* on page J1-5376.
- *shared/functions/exclusive/ClearExclusiveLocal* on page J1-5376.
- *shared/functions/exclusive/ClearExclusiveMonitors* on page J1-5376.
- *shared/functions/exclusive/ExclusiveMonitorsStatus* on page J1-5376.
- *shared/functions/exclusive/IsExclusiveGlobal* on page J1-5377.
- *shared/functions/exclusive/IsExclusiveLocal* on page J1-5377.
- *shared/functions/exclusive/MarkExclusiveGlobal* on page J1-5377.
- *shared/functions/exclusive/MarkExclusiveLocal* on page J1-5377.
- *shared/functions/exclusive/ProcessorID* on page J1-5377.
- *shared/functions/float/fixedtofp/FixedToFP* on page J1-5377.
- *shared/functions/float/fpabs/FPAbs* on page J1-5377.
- *shared/functions/float/fpadd/FPAdd* on page J1-5377.
- *shared/functions/float/fpcompare/FPCompare* on page J1-5378.
- *shared/functions/float/fpcompareeq/FPCompareEQ* on page J1-5378.
- *shared/functions/float/fpcomparege/FPCompareGE* on page J1-5379.
- *shared/functions/float/fpcomparegt/FPCompareGT* on page J1-5379.
- *shared/functions/float/fpconvert/FPConvert* on page J1-5379.
- *shared/functions/float/fpconvertnan/FPConvertNaN* on page J1-5380.
- *shared/functions/float/fpcrtype/FPCRType* on page J1-5380.
- *shared/functions/float/fpdecoderm/FPDecodeRM* on page J1-5380.
- *shared/functions/float/fpdecoderounding/FPDecodeRounding* on page J1-5381.
- *shared/functions/float/fpdefaultnan/FPDefaultNaN* on page J1-5381.
- *shared/functions/float/fpdiv/FPDiv* on page J1-5381.
- *shared/functions/float/fpexc/FPExc* on page J1-5381.
- *shared/functions/float/fpinfinity/FPInfinity* on page J1-5381.
- *shared/functions/float/fpmax/FPMax* on page J1-5382.
- *shared/functions/float/fpmaxnormal/FPMaxNormal* on page J1-5382.
- *shared/functions/float/fpmaxnum/FPMaxNum* on page J1-5382.
- *shared/functions/float/fpmin/FPMin* on page J1-5383.
- *shared/functions/float/fpminnum/FPMinNum* on page J1-5383.
- *shared/functions/float/fpmul/FPMul* on page J1-5383.
- *shared/functions/float/fpmuladd/FPMulAdd* on page J1-5384.
- *shared/functions/float/fpmulx/FPMulX* on page J1-5385.
- *shared/functions/float/fpneg/FPNeg* on page J1-5385.
- *shared/functions/float/fponepointfive/FPOnePointFive* on page J1-5385.
- *shared/functions/float/fpprocessexception/FPProcessException* on page J1-5385.
- *shared/functions/float/fpprocessnan/FPProcessNaN* on page J1-5386.
- *shared/functions/float/fpprocessnans/FPProcessNaNs* on page J1-5386.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/float/fpprocessnans3/FPProcessNaNs3* on page J1-5386.
- *shared/functions/float/fprecipestimate/FPRecipEstimate* on page J1-5387.
- *shared/functions/float/fprecpx/FPRecpX* on page J1-5388.
- *shared/functions/float/fpround/FPRound* on page J1-5389.
- *shared/functions/float/fprounding/FPRounding* on page J1-5390.
- *shared/functions/float/fproundingmode/FPRoundingMode* on page J1-5390.
- *shared/functions/float/fproundint/FPRoundInt* on page J1-5391.
- *shared/functions/float/fprsqrtestimate/FPRSqrtEstimate* on page J1-5391.
- *shared/functions/float/fpsqrt/FPSqrt* on page J1-5392.
- *shared/functions/float/fpsub/FPSub* on page J1-5393.
- *shared/functions/float/fpthree/FPThree* on page J1-5393.
- *shared/functions/float/fptofixed/FPToFixed* on page J1-5393.
- *shared/functions/float/fptwo/FPTwo* on page J1-5394.
- *shared/functions/float/fptype/FPType* on page J1-5394.
- *shared/functions/float/fpunpack/FPUnpack* on page J1-5394.
- *shared/functions/float/fpzero/FPZero* on page J1-5396.
- *shared/functions/float/vfpexpandimm/VFPExpandImm* on page J1-5396.
- *shared/functions/integer/AddWithCarry* on page J1-5396.
- *shared/functions/memory/AccType* on page J1-5396.
- *shared/functions/memory/AddrTop* on page J1-5397.
- *shared/functions/memory/AddressDescriptor* on page J1-5397.
- *shared/functions/memory/Allocation* on page J1-5397.
- *shared/functions/memory/BigEndian* on page J1-5397.
- *shared/functions/memory/BigEndianReverse* on page J1-5397.
- *shared/functions/memory/Cacheability* on page J1-5398.
- *shared/functions/memory/DataMemoryBarrier* on page J1-5398.
- *shared/functions/memory/DataSynchronizationBarrier* on page J1-5398.
- *shared/functions/memory/DeviceType* on page J1-5398.
- *shared/functions/memory/Fault* on page J1-5398.
- *shared/functions/memory/FaultRecord* on page J1-5398.
- *shared/functions/memory/FullAddress* on page J1-5398.
- *shared/functions/memory/Hint_Prefetch* on page J1-5398.
- *shared/functions/memory/MBReqDomain* on page J1-5399.
- *shared/functions/memory/MBReqTypes* on page J1-5399.
- *shared/functions/memory/MemAttrHints* on page J1-5399.
- *shared/functions/memory/MemType* on page J1-5399.
- *shared/functions/memory/MemoryAttributes* on page J1-5399.
- *shared/functions/memory/Permissions* on page J1-5399.
- *shared/functions/memory/PrefetchHint* on page J1-5399.
- *shared/functions/memory/TLBRecord* on page J1-5399.
- *shared/functions/memory/_Mem* on page J1-5400.
- *shared/functions/registers/BranchTo* on page J1-5400.
- *shared/functions/registers/BranchType* on page J1-5400.
- *shared/functions/registers/Hint_Branch* on page J1-5400.
- *shared/functions/registers/NextInstrAddr* on page J1-5400.
- *shared/functions/registers/ResetExternalDebugRegisters* on page J1-5400.
- *shared/functions/registers/ThisInstrAddr* on page J1-5400.
- *shared/functions/registers/_PC* on page J1-5401.
- *shared/functions/registers/_R* on page J1-5401.
- *shared/functions/registers/_V* on page J1-5401.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/sysregisters/SPSR* on page J1-5401.
- *shared/functions/system/ArchVersion* on page J1-5402.
- *shared/functions/system/ClearEventRegister* on page J1-5402.
- *shared/functions/system/ConditionHolds* on page J1-5402.
- *shared/functions/system/CurrentInstrSet* on page J1-5402.
- *shared/functions/system/CurrentPL* on page J1-5402.
- *shared/functions/system/EL0* on page J1-5402.
- *shared/functions/system/ELFromM32* on page J1-5403.
- *shared/functions/system/ELFromSPSR* on page J1-5403.
- *shared/functions/system/ELStateUsingAArch32* on page J1-5403.
- *shared/functions/system/ELStateUsingAArch32K* on page J1-5404.
- *shared/functions/system/ELUsingAArch32* on page J1-5404.
- *shared/functions/system/ELUsingAArch32K* on page J1-5404.
- *shared/functions/system/EndOfInstruction* on page J1-5404.
- *shared/functions/system/EventRegisterSet* on page J1-5404.
- *shared/functions/system/EventRegistered* on page J1-5405.
- *shared/functions/system/GetPSRFromPSTATE* on page J1-5405.
- *shared/functions/system/HasArchVersion* on page J1-5405.
- *shared/functions/system/HaveAArch32EL* on page J1-5405.
- *shared/functions/system/HaveAnyAArch32* on page J1-5405.
- *shared/functions/system/HaveEL* on page J1-5406.
- *shared/functions/system/HighestEL* on page J1-5406.
- *shared/functions/system/HighestELUsingAArch32* on page J1-5406.
- *shared/functions/system/Hint_Debug* on page J1-5406.
- *shared/functions/system/Hint_Yield* on page J1-5406.
- *shared/functions/system/IllegalExceptionReturn* on page J1-5406.
- *shared/functions/system/InstrSet* on page J1-5407.
- *shared/functions/system/InstructionSynchronizationBarrier* on page J1-5407.
- *shared/functions/system/InterruptPending* on page J1-5407.
- *shared/functions/system/IsSecure* on page J1-5407.
- *shared/functions/system/IsSecureBelowEL3* on page J1-5407.
- *shared/functions/system/Mode_Bits* on page J1-5408.
- *shared/functions/system/PLOfEL* on page J1-5408.
- *shared/functions/system/PSTATE* on page J1-5408.
- *shared/functions/system/PrivilegeLevel* on page J1-5408.
- *shared/functions/system/ProcState* on page J1-5408.
- *shared/functions/system/RestoredITBits* on page J1-5409.
- *shared/functions/system/SCRType* on page J1-5409.
- *shared/functions/system/SCR_GEN* on page J1-5409.
- *shared/functions/system/SendEvent* on page J1-5409.
- *shared/functions/system/SetPSTATEFromPSR* on page J1-5409.
- *shared/functions/system/SynchronizeContext* on page J1-5410.
- *shared/functions/system/ThisInstr* on page J1-5410.
- *shared/functions/system/ThisInstrLength* on page J1-5410.
- *shared/functions/system/Unreachable* on page J1-5410.
- *shared/functions/system/UsingAArch32* on page J1-5410.
- *shared/functions/system/WaitForEvent* on page J1-5410.
- *shared/functions/system/WaitForInterrupt* on page J1-5411.
- *shared/functions/unpredictable/ConstrainUnpredictable* on page J1-5411.
- *shared/functions/unpredictable/ConstrainUnpredictableBits* on page J1-5411.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/functions/unpredictable/ConstrainUnpredictableBool* on page J1-5411.
- *shared/functions/unpredictable/ConstrainUnpredictableInteger* on page J1-5411.
- *shared/functions/unpredictable/Constraint* on page J1-5411.
- *shared/functions/vector/AdvSIMDExpandImm* on page J1-5411.
- *shared/functions/vector/PolynomialMult* on page J1-5412.
- *shared/functions/vector/SatQ* on page J1-5412.
- *shared/functions/vector/SignedSatQ* on page J1-5412.
- *shared/functions/vector/UnsignedRSqrtEstimate* on page J1-5412.
- *shared/functions/vector/UnsignedRecipEstimate* on page J1-5413.
- *shared/functions/vector/UnsignedSatQ* on page J1-5413.

## shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault type, integer level)

    bits(6) result;
    case type of
        when Fault_AddressSize         result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag          result = '0010':level<1:0>; assert level IN {1,2,3};
        when Fault_Permission          result = '0011':level<1:0>; assert level IN {1,2,3};
        when Fault_Translation         result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal        result = '010000';
        when Fault_SyncExternalOnWalk  result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity          result = '011000';
        when Fault_SyncParityOnWalk    result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity         result = '011001';
        when Fault_AsyncExternal       result = '010001';
        when Fault_Alignment           result = '100001';
        when Fault_Debug               result = '100010';
        when Fault_TLBConflict         result = '110000';
        when Fault_Lockdown            result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive           result = '110101';  // IMPLEMENTATION DEFINED
        otherwise                      Unreachable();

    return result;
```

## shared/functions/aborts/FaultSyndrome

```
// FaultSyndrome()
// ===============
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode or an Exception Level using AArch64.

bits(25) FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;

    bits(25) iss = Zeros();
    if d_side then
        if IsSecondStage(fault) && !fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.type, fault.level);

    return iss;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/aborts/IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.type != Fault_None;

    if fault.s2fs1walk then
        return fault.type IN {Fault_AccessFlag, Fault_Permission, Fault_Translation,
                              Fault_AddressSize};
    elsif fault.secondstage then
        return fault.type IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
    else
        return FALSE;
```

### shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.type);
```

### shared/functions/aborts/IsDebugException

```
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.type != Fault_None;
    return fault.type == Fault_Debug;
```

### shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =================

boolean IsExternalAbort(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_SyncExternal, Fault_SyncParity, Fault_AsyncExternal, Fault_AsyncParity,
                     Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk});

// IsExternalAbort()
// =================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.type);
```

### shared/functions/aborts/IsFault

```
// IsFault()
// =========
// Return true if a fault is associated with an address descriptor
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.type != Fault_None;
```

### shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.type != Fault_None;

    return fault.secondstage;
```

### shared/functions/aborts/LSInstructionSyndrome

```
bits(11) LSInstructionSyndrome();
```

### shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

### shared/functions/common/ASR_C

```
// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

### shared/functions/common/Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

### shared/functions/common/Align

```
// Align()
// =======

integer Align(integer x, integer y)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    return y * (x DIV y);

// Align()
// =======

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

### shared/functions/common/BitCount

```
// BitCount()
// ==========

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

### shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

### shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
    return N - 1 - HighestSetBit(x);
```

### shared/functions/common/Elem

```
// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector<e*size+size-1 : e*size>;

// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e]
    return Elem[vector, e, size];

// Elem[] - assignment form
// ========================

Elem[bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;

// Elem[] - assignment form
// ========================

Elem[bits(N) &vector, integer e] = bits(size) value
    Elem[vector, e, size] = value;
    return;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/common/Extend

```
// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

// Extend()
// ========

bits(N) Extend(bits(M) x, boolean unsigned)
    return Extend(x, N, unsigned);
```

### shared/functions/common/HighestSetBit

```
// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

### shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

### shared/functions/common/IsOnes

```
// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

### shared/functions/common/IsZero

```
// IsZero()
// ========

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

### shared/functions/common/IsZeroBit

```
// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

### shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

### shared/functions/common/LSL_C

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

### shared/functions/common/LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

### shared/functions/common/LSR_C

```
// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

### shared/functions/common/LowestSetBit

```
// LowestSetBit()
// ==============

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

### shared/functions/common/Max

```
// Max()
// =====

integer Max(integer a, integer b)
    return if a >= b then a else b;

// Max()
// =====
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
real Max(real a, real b)
    return if a >= b then a else b;
```

### shared/functions/common/Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```

### shared/functions/common/NOT

```
bits(N) NOT(bits(N) x);
```

### shared/functions/common/Ones

```
// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// ======

bits(N) Ones()
    return Ones(N);
```

### shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;
```

### shared/functions/common/ROR_C

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/common/Replicate

```
// Replicate()
// ===========

bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);
```

### shared/functions/common/RoundDown

```
integer RoundDown(real x);
```

### shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

### shared/functions/common/RoundUp

```
integer RoundUp(real x);
```

### shared/functions/common/SInt

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

### shared/functions/common/SignExtend

```
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;

// SignExtend()
// ============

bits(N) SignExtend(bits(M) x)
    return SignExtend(x, N);
```

### shared/functions/common/UInt

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

### shared/functions/common/ZeroExtend

```
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;

// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

### shared/functions/common/Zeros

```
// Zeros()
// =======

bits(N) Zeros(integer N)
    return Replicate('0',N);

// Zeros()
// =======

bits(N) Zeros()
    return Zeros(N);
```

### shared/functions/crc/BitReverse

```
// BitReverse()
// ============

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<N-i-1> = data<i>;
    return result;
```

### shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// ============

boolean HaveCRCExt()
    return boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

### shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
    assert N > 32;
    for i = N-1 downto 32
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR poly:Zeros(i-32);
    return data<31:0>;
```

### shared/functions/crypto/AESInvMixColumns

```
bits(128) AESInvMixColumns(bits (128) op);
```

### shared/functions/crypto/AESInvShiftRows

```
bits(128) AESInvShiftRows(bits(128) op);
```

### shared/functions/crypto/AESInvSubBytes

```
bits(128) AESInvSubBytes(bits(128) op);
```

### shared/functions/crypto/AESMixColumns

```
bits(128) AESMixColumns(bits (128) op);
```

### shared/functions/crypto/AESShiftRows

```
bits(128) AESShiftRows(bits(128) op);
```

### shared/functions/crypto/AESSubBytes

```
bits(128) AESSubBytes(bits(128) op);
```

### shared/functions/crypto/HaveCryptoExt

```
boolean HaveCryptoExt();
```

### shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

### shared/functions/crypto/SHA256hash

```
// SHA256hash()
// ============

bits(128) SHA256hash (bits (128) X, bits(128) Y, bits(128) W, boolean part1)
    bits(32) chs, maj, t;

    for e = 0 to 3
        chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
        maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
        t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
        X<127:96> = t + X<127:96>;
        Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
        <Y, X> = ROL(Y : X, 32);
    return (if part1 then X else Y);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/crypto/SHAchoose

```
// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);
```

### shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// ===============

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

### shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// ===============

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

### shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =============

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));
```

### shared/functions/crypto/SHAparity

```
// SHAparity()
// ===========

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

### shared/functions/exclusive/ClearExclusiveByAddress

```
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

### shared/functions/exclusive/ClearExclusiveLocal

```
ClearExclusiveLocal(integer processorid);
```

### shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// ========================

// Clear the local Exclusive Monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

### shared/functions/exclusive/ExclusiveMonitorsStatus

```
bit ExclusiveMonitorsStatus();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/exclusive/IsExclusiveGlobal

```
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

### shared/functions/exclusive/IsExclusiveLocal

```
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

### shared/functions/exclusive/MarkExclusiveGlobal

```
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

### shared/functions/exclusive/MarkExclusiveLocal

```
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

### shared/functions/exclusive/ProcessorID

```
integer ProcessorID();
```

### shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// ===========

// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
    assert N IN {32,64};
    assert M IN {32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0');
    else
        result = FPRound(real_operand, fpcr, rounding);

    return result;
```

### shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =======

bits(N) FPAbs(bits(N) op)
    assert N IN {32,64};
    return '0' : op<N-2:0>;
```

### shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

## shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = '0011';
        if type1==FPType_SNaN || type2==FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';
    return result;
```

## shared/functions/float/fpcompareeq/FPCompareEQ

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        if type1==FPType_SNaN || type2==FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/float/fpcomparege/FPCompareGE

```
// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
    return result;
```

### shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
    return result;
```

### shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding)
    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (type,sign,value) = FPUnpack(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if type == FPType_SNaN || type == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        if type == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif type == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result = FPInfinity(sign);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRound(value, fpcr, rounding);

    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType fpcr)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

## shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// ==============

// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```

## shared/functions/float/fpcrtype/FPCRType

```
type FPCRType;
```

## shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
    case rm of
        when '00' return FPRounding_TIEAWAY; // A
        when '01' return FPRounding_TIEEVEN; // N
        when '10' return FPRounding_POSINF;  // P
        when '11' return FPRounding_NEGINF;  // M
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// ==================

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF;  // P
        when '10' return FPRounding_NEGINF;  // M
        when '11' return FPRounding_ZERO;    // Z
```

### shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN()
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = '0';
    exp  = Ones(E);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

### shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);
    return result;
```

### shared/functions/float/fpexc/FPExc

```
enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                         FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

### shared/functions/float/fpinfinity/FPInfinity

```
// FPInfinity()
// ============
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(N) FPInfinity(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Ones(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

### shared/functions/float/fpmax/FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 > value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType_Infinity then
            result = FPInfinity(sign);
        elsif type == FPType_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign);
        else
            result = FPRound(value, fpcr);
    return result;
```

### shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```

### shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};

    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // treat a single quiet-NaN as -Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1');

    return FPMax(op1, op2, fpcr);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/float/fpmin/FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 < value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType_Infinity then
            result = FPInfinity(sign);
        elsif type == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            result = FPRound(value, fpcr);
    return result;
```

### shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};

    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // Treat a single quiet-NaN as +Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('0');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('0');

    return FPMin(op1, op2, fpcr);
```

### shared/functions/float/fpmul/FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

## shared/functions/float/fpmuladd/FPMulAdd

```
// FPMulAdd()
// ==========
//
// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);

    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity);  zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);

    return result;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    bits(N) result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

### shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)
    assert N IN {32,64};
    return NOT(op<N-1>) : op<N-2:0>;
```

### shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

### shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)
    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp     cumul = 0;
        when FPExc_DivideByZero  cumul = 1;
        when FPExc_Overflow      cumul = 2;
        when FPExc_Underflow     cumul = 3;
        when FPExc_Inexact       cumul = 4;
        when FPExc_InputDenorm   cumul = 7;
    enable = cumul + 8;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrapException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

### shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType type, bits(N) op, FPCRType fpcr)
    assert N IN {32,64};
    assert type IN {FPType_QNaN, FPType_SNaN};

    topfrac = if N == 32 then 22 else 51;
    result = op;
    if type == FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN();
    return result;
```

### shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                 bits(N) op1, bits(N) op2,
                                 FPCRType fpcr)
    assert N IN {32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

### shared/functions/float/fpprocessnans3/FPProcessNaNs3

```
// FPProcessNaNs3()
// ================
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRType fpcr)
    assert N IN {32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

## shared/functions/float/fprecipestimate/FPRecipEstimate

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {32, 64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPType_Infinity then
        result = FPZero(sign);
    elsif type == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (N == 32 && Abs(value) < 2.0^-128)
          || (N == 64 && Abs(value) < 2.0^-1024) then
        case FPRoundingMode(fpcr) of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    elsif fpcr.FZ == '1'
          && ((N == 32 && Abs(value) >= 2.0^126)
              || (N == 64 && Abs(value) >= 2.0^1022)) then
        // Result flushed to zero of correct sign
        result = FPZero(sign);
        FPProcessException(FPExc_Underflow, fpcr);
    else
        // Scale to a double-precision value in the range 0.5 <= x < 1.0, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction extended with zeros.

        if N == 32 then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            fraction = operand<22:0> : Zeros(29);
            exp = UInt(operand<30:23>);
        else // N == 64
            fraction = operand<51:0>;
            exp = UInt(operand<62:52>);

        if exp == 0 then
            if fraction<51> == 0 then
                exp = -1;
                fraction = fraction<49:0>:'00';
            else
                fraction = fraction<50:0>:'0';
        scaled = '0' : '01111111110' : fraction<51:44> : Zeros(44);

        if N == 32 then
            result_exp = 253 - exp;     // In range 253-254 = -1 to 253+1 = 254
        else // N == 64
            result_exp = 2045 - exp;    // In range 2045-2046 = -1 to 2045+1 = 2046

        // Call C function to get reciprocal estimate of scaled value.
        // Input is rounded down to a multiple of 1/512.
        estimate = recip_estimate(scaled);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Convert to scaled single-precision result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.

        fraction = estimate<51:0>;
        if result_exp == 0 then
            fraction = '1' : fraction<51:1>;
        elsif result_exp == -1 then
            fraction = '01' : fraction<51:2>;
            result_exp = 0;
        if N == 32 then
            result = sign : result_exp<N-25:0> : fraction<51:29>;
        else // N == 64
            result = sign : result_exp<N-54:0> : fraction<51:0>;

    return result;
```

## shared/functions/float/fprecpx/FPRecpX

```
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr)
    assert N IN {32,64};
    bits(N) result;
    integer esize = if N == 32 then 8 else 11;
    bits(esize)     exp;
    bits(esize)     max_exp;
    bits(N-esize-1) frac = Zeros();

    if N == 32 then
        exp = op<23+esize-1:23>;
    else
        exp = op<52+esize-1:52>;
    max_exp = Ones(esize) - 1;

    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    else
        if IsZero(exp) then // Zero and denormals
            result = sign:max_exp:frac;
        else // Infinities and normals
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            result = sign:NOT(exp):frac;

    return result;
```

## shared/functions/float/fpround/FPRound

```
// FPRound()
// =========

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Deal with flush-to-zero.
    if fpcr.FZ == '1' && N != 16 && exponent < minimum_exp then
        // Flush-to-zero never generates a trapped exception
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            FPSR.UFC = '1';
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped.
    if biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
        FPProcessException(FPExc_Underflow, fpcr);

    // Round result according to rounding mode.
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up = (error != 0.0 && sign == '0');
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then      // Rounded up from denormalized to normalized
            biased_exp = 1;
        if int_mant == 2^(F+1) then  // Rounded up to next exponent
            biased_exp = biased_exp + 1;  int_mant = int_mant DIV 2;

    // Handle rounding to odd aka Von Neumann rounding
    if error != 0.0 && rounding == FPRounding_ODD then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
            FPProcessException(FPExc_Overflow, fpcr);
            error = 1.0;  // Ensure that an Inexact exception occurs
        else
            result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;
    else                                    // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0;  // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;

    // Deal with Inexact exception.
    if error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
```

## shared/functions/float/fprounding/FPRounding

```
enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
                         FPRounding_NEGINF,  FPRounding_ZERO,
                         FPRounding_TIEAWAY, FPRounding_ODD};
```

## shared/functions/float/fproundingmode/FPRoundingMode

```
// FPRoundingMode()
// ================

// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
    return FPDecodeRounding(fpcr.RMode);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/float/froundint/FPRoundInt

```
// FPRoundInt()
// ============

// Round OP to nearest integral floating point value using rounding mode ROUNDING.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to OP.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};

    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPType_Infinity then
        result = FPInfinity(sign);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    else
        // extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact
        if real_result == 0.0 then
            result = FPZero(sign);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO);

        // Generate inexact exceptions
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

### shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {32, 64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPType_Zero then
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif type == FPType_Infinity then
        result = FPZero('0');
    else
        // Scale to a double-precision value in the range 0.25 <= x < 1.0, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        if N == 32 then
            fraction = operand<22:0> : Zeros(29);
            exp = UInt(operand<30:23>);
        else // N == 64
            fraction = operand<51:0>;
            exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == 0 do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        if exp<0> == '0' then
            scaled = '0' : '01111111110' : fraction<51:44> : Zeros(44);
        else
            scaled = '0' : '01111111101' : fraction<51:44> : Zeros(44);

        if N == 32 then
            result_exp = (380 - exp) DIV 2;
        else // N == 64
            result_exp = (3068 - exp) DIV 2;

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_sqrt_estimate(scaled);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Convert to scaled single-precision result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.

        if N == 32 then
            result = '0' : result_exp<N-25:0> : estimate<51:29>;
        else // N == 64
            result = '0' : result_exp<N-54:0> : estimate<51:0>;
    return result;
```

## shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
    assert N IN {32,64};
    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    elsif type == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        result = FPRound(Sqrt(value), fpcr);
    return result;
```

### shared/functions/float/fpsub/FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

### shared/functions/float/fpthree/FPThree

```
// FPThree()
// =========

bits(N) FPThree(bit sign)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

### shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
    assert N IN {32,64};
    assert M IN {32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    (type,sign,value) = FPUnpack(op, fpcr);

    // If NaN, set cumulative flag or take exception
    if type == FPType_SNaN || type == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

### shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =======

bits(N) FPTwo(bit sign)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

### shared/functions/float/fptype/FPType

```
enumeration FPType      {FPType_Nonzero, FPType_Zero, FPType_Infinity,
                         FPType_QNaN, FPType_SNaN};
```

### shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// ==========
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)
    assert N IN {16,32,64};

    if N == 16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            // Produce zero if value is zero
            if IsZero(frac16) then
                type = FPType_Zero;  value = 0.0;
            else
                type = FPType_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 then

        sign   = fpval<31>;
        exp32  = fpval<30:23>;
        frac32 = fpval<22:0>;
        if IsZero(exp32) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac32) || fpcr.FZ == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac32) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpcr);
            else
                type = FPType_Nonzero;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64

        sign   = fpval<63>;
        exp64  = fpval<62:52>;
        frac64 = fpval<51:0>;
        if IsZero(exp64) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac64) || fpcr.FZ == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac64) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpcr);
            else
                type = FPType_Nonzero;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;
    return (type, sign, value);
```

### shared/functions/float/fpzero/FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Zeros(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

### shared/functions/float/vfpexpandimm/VFPExpandImm

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8)
    assert N IN {32,64};
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    return sign : exp : frac;
```

### shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

### shared/functions/memory/AccType

```
enumeration AccType {AccType_NORMAL, AccType_VEC,        // Normal loads and stores
                     AccType_STREAM, AccType_VECSTREAM,  // Streaming loads and stores
                     AccType_ATOMIC,                     // Atomic loads and stores
                     AccType_ORDERED,                    // Load-Acquire and Store-Release
                     AccType_UNPRIV,                     // Load and store unprivileged
                     AccType_IFETCH,                     // Instruction fetch
                     AccType_PTW,                        // Page table walk
                     // Other operations
                     AccType_DC,                         // Data cache maintenance
                     AccType_IC,                         // Instruction cache maintenance
                     AccType_AT};                        // Address translation
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/memory/AddrTop

```
// AddrTop()
// =========

integer AddrTop(bits(64) address)
    // Return the MSB number of a virtual address in the current stage 1 translation
    // regime. If EL1 is using AArch64 then addresses from EL0 using AArch32
    // are zero-extended to 64 bits.
    if UsingAArch32() && !(PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) then
        // AArch32 translation regime.
        return 31;
    else
        // AArch64 translation regime.
        case PSTATE.EL of
            when EL0, EL1
                tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            when EL2
                tbi = TCR_EL2.TBI;
            when EL3
                tbi = TCR_EL3.TBI;
        return (if tbi == '1' then 55 else 63);
```

### shared/functions/memory/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord      fault,      // fault.type indicates whether the address is valid
    MemoryAttributes mematttrs,
    FullAddress      paddress,
    bits(64)         vaddress
)
```

### shared/functions/memory/Allocation

```
constant bits(2) MemHint_No = '00';     // No allocate
constant bits(2) MemHint_WA = '01';     // Write-allocate, Read-no-allocate
constant bits(2) MemHint_RA = '10';     // Read-allocate, Write-no-allocate
constant bits(2) MemHint_RWA = '11';    // Read-allocate and Write-allocate
```

### shared/functions/memory/BigEndian

```
// BigEndian()
// ===========

boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR_EL1.E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

### shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// ==================

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```

### shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

### shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);
```

### shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

### shared/functions/memory/Fault

```
enumeration Fault {Fault_None,
                   Fault_AccessFlag,
                   Fault_Alignment,
                   Fault_Background,
                   Fault_Domain,
                   Fault_Permission,
                   Fault_Translation,
                   Fault_AddressSize,
                   Fault_SyncExternal,
                   Fault_SyncExternalOnWalk,
                   Fault_SyncParity,
                   Fault_SyncParityOnWalk,
                   Fault_AsyncParity,
                   Fault_AsyncExternal,
                   Fault_Debug,
                   Fault_TLBConflict,
                   Fault_Lockdown,
                   Fault_Exclusive,
                   Fault_ICacheMaint};
```

### shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault    type,         // Fault Status
                     AccType  acctype,      // Type of access that faulted
                     bits(48) ipaddress,    // Intermediate physical address
                     boolean  s2fs1walk,    // Is on a Stage 1 page table walk
                     boolean  write,        // TRUE for a write, FALSE for a read
                     integer  level,        // For translation, access flag and permission faults
                     bit      extflag,      // IMPLEMENTATION DEFINED syndrome for external aborts
                     boolean  secondstage,  // Is a Stage 2 abort
                     bits(4)  domain,       // Domain number, AArch32 only
                     bits(4)  debugmoe)     // Debug method of entry, from AArch32 only
```

### shared/functions/memory/FullAddress

```
type FullAddress is (
    bits(48) physicaladdress,
    bit      NS                 // '0' = Secure, '1' = Non-secure
)
```

### shared/functions/memory/Hint_Prefetch

```
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/memory/MBReqDomain

```
enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

### shared/functions/memory/MBReqTypes

```
enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

### shared/functions/memory/MemAttrHints

```
type MemAttrHints is (
    bits(2) attrs,  // The possible encodings for each attributes field are as below
    bits(2) hints,  // The possible encodings for the hints are below
    boolean transient
)
```

### shared/functions/memory/MemType

```
enumeration MemType {MemType_Normal, MemType_Device};
```

### shared/functions/memory/MemoryAttributes

```
type MemoryAttributes is (
    MemType      type,

    DeviceType   device,     // For Device memory types
    MemAttrHints inner,      // Inner hints and attributes
    MemAttrHints outer,      // Outer hints and attributes

    boolean      shareable,
    boolean      outershareable
)
```

### shared/functions/memory/Permissions

```
type Permissions is (
    bits(3) ap,   // Access permission bits
    bit     xn,   // Execute-never bit
    bit     pxn   // Privileged execute-never bit
)
```

### shared/functions/memory/PrefetchHint

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

### shared/functions/memory/TLBRecord

```
type TLBRecord is (
    Permissions       perms,
    bit               nG,             // '0' = Global, '1' = not Global
    bits(4)           domain,         // AArch32 only
    boolean           contiguous,     // Contiguous bit from page table
    integer           level,          // In AArch32 Short-descriptort format, indicates Section/Page
    integer           blocksize,      // Describes size of memory translated in KBytes
    AddressDescriptor addrdesc
)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/memory/_Mem

```
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccType acctype];

_Mem[AddressDescriptor desc, integer size, AccType acctype] = bits(8*size) value;
```

### shared/functions/registers/BranchTo

```
// BranchTo()
// ==========

// Set program counter to a new address, with a branch reason hint
// for possible use by hardware fetching the next instruction.

BranchTo(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        // Remove the tag bits from a tagged target
        case PSTATE.EL of
            when EL0, EL1
                if target<55> == '1' && TCR_EL1.TBI1 == '1' then
                    target<63:56> = '11111111';
                if target<55> == '0' && TCR_EL1.TBI0 == '1' then
                    target<63:56> = '00000000';
            when EL2
                if TCR_EL2.TBI == '1' then
                    target<63:56> = '00000000';
            when EL3
                if TCR_EL3.TBI == '1' then
                    target<63:56> = '00000000';
        _PC = target<63:0>;
    return;
```

### shared/functions/registers/BranchType

```
enumeration BranchType {BranchType_CALL, BranchType_ERET, BranchType_DBGEXIT,
                        BranchType_RET, BranchType_JMP, BranchType_EXCEPTION,
                        BranchType_UNKNOWN};
```

### shared/functions/registers/Hint_Branch

```
Hint_Branch(BranchType hint);
```

### shared/functions/registers/NextInstrAddr

```
bits(N) NextInstrAddr();
```

### shared/functions/registers/ResetExternalDebugRegisters

```
ResetExternalDebugRegisters(boolean cold_reset);
```

### shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
bits(N) ThisInstrAddr()
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

### shared/functions/registers/_PC

```
bits(64) _PC;
```

### shared/functions/registers/_R

```
array bits(64) _R[0..30];
```

### shared/functions/registers/_V

```
array bits(128) _V[0..31];
```

### shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// ============================

bits(32) SPSR[]
    bits(32) result;
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ      result = SPSR_fiq;
            when M32_IRQ      result = SPSR_irq;
            when M32_Svc      result = SPSR_svc;
            when M32_Monitor  result = SPSR_mon;
            when M32_Abort    result = SPSR_abt;
            when M32_Hyp      result = SPSR_hyp;
            when M32_Undef    result = SPSR_und;
            otherwise         Unreachable();
    else
        case PSTATE.EL of
            when EL1          result = SPSR_EL1;
            when EL2          result = SPSR_EL2;
            when EL3          result = SPSR_EL3;
            otherwise         Unreachable();

    return result;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(32) value
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ      SPSR_fiq = value;
            when M32_IRQ      SPSR_irq = value;
            when M32_Svc      SPSR_svc = value;
            when M32_Monitor  SPSR_mon = value;
            when M32_Abort    SPSR_abt = value;
            when M32_Hyp      SPSR_hyp = value;
            when M32_Undef    SPSR_und = value;
            otherwise         Unreachable();
    else
        case PSTATE.EL of
            when EL1          SPSR_EL1 = value;
            when EL2          SPSR_EL2 = value;
            when EL3          SPSR_EL3 = value;
            otherwise         Unreachable();

    return;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/system/ArchVersion

```
enumeration ArchVersion {
    ARMv8p0,
};
```

### shared/functions/system/ClearEventRegister

```
ClearEventRegister();
```

### shared/functions/system/ConditionHolds

```
// ConditionHolds()
// ================

// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1');                          // EQ or NE
        when '001' result = (PSTATE.C == '1');                          // CS or CC
        when '010' result = (PSTATE.N == '1');                          // MI or PL
        when '011' result = (PSTATE.V == '1');                          // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');       // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V);                     // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');  // GT or LE
        when '111' result = TRUE;                                       // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

### shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()

    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;
```

### shared/functions/system/CurrentPL

```
// CurrentPL()
// ===========

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);
```

### shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/system/ELFromM32

```
// ELFromM32()
// ===========

// Convert an AArch32 mode encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'mode<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'mode'.

(boolean,bits(2)) ELFromM32(bits(5) mode)
    bits(2) el;
    boolean valid = TRUE;
    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
            valid = !HaveEL(EL3) || SCR_GEN[].NS == '1';
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            el = if HaveEL(EL3) && HighestELUsingAArch32() && SCR.NS == '0' then EL3 else EL1;
        when M32_User
            el = EL0;
        otherwise
            valid = FALSE;                              // Passed an illegal mode value
    if valid then valid = HaveAArch32EL(el);
    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

### shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(32) spsr)
    if spsr<4> == '0' then                              // AArch64 state
        el = spsr<3:2>;
        if HighestELUsingAArch32() then                 // No AArch64 support
            valid = FALSE;
        elsif !HaveEL(el) then                          // Exception level not implemented
            valid = FALSE;
        elsif spsr<1> == '1' then                       // M[1] must be 0
            valid = FALSE;
        elsif el == EL0 && spsr<0> == '1' then          // for EL0, M[0] must be 0
            valid = FALSE;
        elsif el == EL2 && HaveEL(EL3) && SCR_EL3.NS == '0' then
            valid = FALSE;                              // EL2 only valid in Non-secure state
        else
            valid = TRUE;
    elsif !HaveAnyAArch32() then                        // AArch32 not supported
        valid = FALSE;
    else                                                // AArch32 state
        (valid, el) = ELFromM32(spsr<4:0>);
    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
```

### shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====================
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

### shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// ======================

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    // Returns (known, aarch32):
    //   'known'   is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
    //             using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
    //   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
    boolean aarch32;
    known = TRUE;
    if !HaveAArch32EL(el) then
        aarch32 = FALSE;                            // All levels are using AArch64
    elsif HighestELUsingAArch32() then
        aarch32 = TRUE;                             // All levels are using AArch32
    else
        aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0';
        aarch32_at_el1 = aarch32_below_el3 || (HaveEL(EL2) && !secure && HCR_EL2.RW == '0');

        if el == EL0 && !aarch32_at_el1 then        // Only know if EL0 using AArch32 from PSTATE
            if PSTATE.EL == EL0 then
                aarch32 = PSTATE.nRW == '1';        // EL0 controlled by PSTATE
            else
                known = FALSE;                      // EL0 state is UNKNOWN
        else
            aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});
    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);
```

### shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

### shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

### shared/functions/system/EndOfInstruction

```
EndOfInstruction();
```

### shared/functions/system/EventRegisterSet

```
EventRegisterSet();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/system/EventRegistered

```
boolean EventRegistered();
```

### shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(32) GetPSRFromPSTATE()
    bits(32) spsr = Zeros();
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    spsr<21>    = PSTATE.SS;
    spsr<20>    = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        spsr<27>    = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        spsr<19:16> = PSTATE.GE;
        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9>     = PSTATE.E;
        spsr<8:6>   = PSTATE.<A,I,F>;                   // No PSTATE.D in AArch32 state
        spsr<5>     = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW;               // bit [4] is the discriminator
        spsr<4:0>   = PSTATE.M;
    else // AArch64 state
        spsr<9:6>   = PSTATE.<D,A,I,F>;
        spsr<4>     = PSTATE.nRW;
        spsr<3:2>   = PSTATE.EL;
        spsr<0>     = PSTATE.SP;
    return spsr;
```

### shared/functions/system/HasArchVersion

```
// HasArchVersion()
// ================

// Return TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
```

### shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// ===============

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32
    if !HaveEL(el) then
        return FALSE;
    elsif !HaveAnyAArch32() then
        return FALSE;                                   // No exception level can use AArch32
    elsif HighestELUsingAArch32() then
        return TRUE;                                    // All exception levels must use AArch32
    elsif el == EL0 then
        return TRUE;                                    // EL0 must support using AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

### shared/functions/system/HaveAnyAArch32

```
// HaveAnyAArch32()
// ================
// Return TRUE if AArch32 state is supported at any Exception level
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
boolean HaveAnyAArch32()
    return boolean IMPLEMENTATION_DEFINED;
```

### shared/functions/system/HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    if el IN {EL1,EL0} then
        return TRUE;                            // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;
```

### shared/functions/system/HighestEL

```
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

### shared/functions/system/HighestELUsingAArch32

```
// HighestELUsingAArch32()
// =======================
// Return TRUE if configured to boot into AArch32 operation

boolean HighestELUsingAArch32()
    if !HaveAnyAArch32() then return FALSE;
    return boolean IMPLEMENTATION_DEFINED;       // e.g. CFG32SIGNAL == HIGH
```

### shared/functions/system/Hint_Debug

```
Hint_Debug(bits(4) option);
```

### shared/functions/system/Hint_Yield

```
Hint_Yield();
```

### shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(32) spsr)

    // Check for return:
    //   * To an unimplemented Exeception level.
    //   * To EL2 in Secure state.
    //   * To EL0 using AArch64 state, with SPSR.M[0]==1.
    //   * To AArch64 state with SPSR.M[1]==1.
    //   * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for return:
    //   * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //     Execution state used in the Exception level being returned to, as determined by
    //     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    //   * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    //   * To AArch64 state from AArch32 state (should be caught by above)

    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for return to EL1 in Non-secure state when HCR_EL2.TGE is set
    if target == EL1 && !IsSecureBelowEL3() && HCR_EL2.TGE == '1' then return TRUE;

    return FALSE;
```

### shared/functions/system/InstrSet

```
enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

### shared/functions/system/InstructionSynchronizationBarrier

```
InstructionSynchronizationBarrier();
```

### shared/functions/system/InterruptPending

```
boolean InterruptPending();
```

### shared/functions/system/IsSecure

```
// IsSecure()
// ==========

boolean IsSecure()
    // Return TRUE if current Exception level is in Secure state.
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elsif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
```

### shared/functions/system/IsSecureBelowEL3

```
// IsSecureBelowEL3()
// ==================

// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    if HaveEL(EL3) then
        return SCR_GEN[].NS == '0';
    elsif HaveEL(EL2) then
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure;
        return boolean IMPLEMENTATION_DEFINED;
```

### shared/functions/system/Mode_Bits

```
constant bits(5) M32_User    = '10000';
constant bits(5) M32_FIQ     = '10001';
constant bits(5) M32_IRQ     = '10010';
constant bits(5) M32_Svc     = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort   = '10111';
constant bits(5) M32_Hyp     = '11010';
constant bits(5) M32_Undef   = '11011';
constant bits(5) M32_System  = '11111';
```

### shared/functions/system/PLOfEL

```
// PLOfEL()
// ========

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3  return if HighestELUsingAArch32() then PL1 else PL3;
        when EL2  return PL2;
        when EL1  return PL1;
        when EL0  return PL0;
```

### shared/functions/system/PSTATE

```
ProcState PSTATE;
```

### shared/functions/system/PrivilegeLevel

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

### shared/functions/system/ProcState

```
type ProcState is (
    bits (1) N,        // Negative condition flag
    bits (1) Z,        // Zero condition flag
    bits (1) C,        // Carry condition flag
    bits (1) V,        // oVerflow condition flag
    bits (1) D,        // Debug mask bit                     [AArch64 only]
    bits (1) A,        // Asynchronous abort mask bit
    bits (1) I,        // IRQ mask bit
    bits (1) F,        // FIQ mask bit
    bits (1) SS,       // Software step bit
    bits (1) IL,       // Illegal Execution state bit
    bits (2) EL,       // Exception Level
    bits (1) nRW,      // not Register Width: 0=64, 1=32
    bits (1) SP,       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,        // Cumulative saturation flag         [AArch32 only]
    bits (4) GE,       // Greater than or Equal flags        [AArch32 only]
    bits (8) IT,       // If-then bits, RES0 in CPSR         [AArch32 only]
    bits (1) J,        // J bit, RES0                        [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,        // T32 bit, RES0 in CPSR              [AArch32 only]
    bits (1) E,        // Endianness bit                     [AArch32 only]
    bits (5) M         // Mode field                         [AArch32 only]
)
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/system/RestoredITBits

```
// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(32) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool() then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

### shared/functions/system/SCRType

```
type SCRType;
```

### shared/functions/system/SCR_GEN

```
// SCR_GEN[]
// =========

SCRType SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(32) r;
    if HighestELUsingAArch32() then
        r = SCR;
    else
        r = SCR_EL3;
    return r;
```

### shared/functions/system/SendEvent

```
SendEvent();
```

### shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(32) spsr)

    SynchronizeContext();

    PSTATE.SS = DebugExceptionReturnSS(spsr);

    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
    else
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then                  // AArch32 state
            AArch32.WriteMode(spsr<4:0>);       // Sets PSTATE.EL correctly
        else                                    // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL  = spsr<3:2>;
            PSTATE.SP  = spsr<0>;

    // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
    // the T bit is set to zero or copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool() then spsr<5> = '0';

    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if PSTATE.nRW == '1' then                   // AArch32 state
        PSTATE.Q         = spsr<27>;
        PSTATE.IT        = RestoredITBits(spsr);
        PSTATE.GE        = spsr<19:16>;
        PSTATE.E         = spsr<9>;
        PSTATE.<A,I,F>   = spsr<8:6>;           // No PSTATE.D in AArch32 state
        PSTATE.T         = spsr<5>;             // PSTATE.J is RES0
    else                                        // AArch64 state
        PSTATE.<D,A,I,F> = spsr<9:6>;           // No PSTATE.<Q,IT,GE,E,T> in AArch64 state

    return;
```

### shared/functions/system/SynchronizeContext

```
SynchronizeContext();
```

### shared/functions/system/ThisInstr

```
bits(32) ThisInstr();
```

### shared/functions/system/ThisInstrLength

```
integer ThisInstrLength();
```

### shared/functions/system/Unreachable

```
Unreachable()
    assert FALSE;
```

### shared/functions/system/UsingAArch32

```
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAnyAArch32() then assert !aarch32;
    if HighestELUsingAArch32() then assert aarch32;
    return aarch32;
```

### shared/functions/system/WaitForEvent

```
WaitForEvent();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/functions/system/WaitForInterrupt

```
WaitForInterrupt();
```

### shared/functions/unpredictable/ConstrainUnpredictable

```
Constraint ConstrainUnpredictable();
```

### shared/functions/unpredictable/ConstrainUnpredictableBits

```
(Constraint,bits(width)) ConstrainUnpredictableBits();
```

### shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

boolean ConstrainUnpredictableBool()

    c = ConstrainUnpredictable();
    assert c IN {Constraint_TRUE, Constraint_FALSE};
    return (c == Constraint_TRUE);
```

### shared/functions/unpredictable/ConstrainUnpredictableInteger

```
(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high);
```

### shared/functions/unpredictable/Constraint

```
enumeration Constraint    {// General:
                           Constraint_NONE, Constraint_UNKNOWN,
                           Constraint_UNDEF, Constraint_NOP,
                           Constraint_TRUE, Constraint_FALSE,
                           Constraint_DISABLED,
                           // Load-store:
                           Constraint_WBSUPPRESS, Constraint_FAULT,
                           // IPA too large
                           Constraint_FORCE, Constraint_FORCENOSLCHECK};
```

### shared/functions/vector/AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);

    return imm64;
```

### shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

### shared/functions/vector/SatQ

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```

### shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

### shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =======================

bits(32) UnsignedRSqrtEstimate(bits(32) operand)

    if operand<31:30> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(32);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
    else
        // Generate double-precision value = operand * 2^(-32). This has zero sign bit, with:
        //     exponent = 1022 or 1021 = double-precision representation of 2^(-1) or 2^(-2)
        //     fraction taken from operand, excluding its most significant one or two bits.
        if operand<31> == '1' then
            dp_operand = '0 01111111110' : operand<30:0> : Zeros(21);
        else  // operand<31:30> == '01'
            dp_operand = '0 01111111101' : operand<29:0> : Zeros(22);

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_sqrt_estimate(dp_operand);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Multiply by 2^31 and convert to an unsigned integer - this just involves
        // concatenating the implicit units bit with the top 31 fraction bits.
        result = '1' : estimate<51:21>;

    return result;
```

### shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

bits(32) UnsignedRecipEstimate(bits(32) operand)

    if operand<31> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(32);
    else
        // Generate double-precision value = operand * 2^(-32). This has zero sign bit, with:
        //     exponent = 1022 = double-precision representation of 2^(-1)
        //     fraction taken from operand, excluding its most significant bit.
        dp_operand = '0 01111111110' : operand<30:0> : Zeros(21);

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_estimate(dp_operand);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Multiply by 2^31 and convert to an unsigned integer - this just involves
        // concatenating the implicit units bit with the top 31 fraction bits.
        result = '1' : estimate<51:21>;

    return result;
```

### shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

## J1.3.4 shared/translation

This section includes the following pseudocode functions:


- *shared/translation/attrs/CombineS1S2AttrHints* on page J1-5414.
- *shared/translation/attrs/CombineS1S2Desc* on page J1-5414.
- *shared/translation/attrs/CombineS1S2Device* on page J1-5415.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *shared/translation/attrs/LongConvertAttrsHints* on page J1-5415.
- *shared/translation/attrs/MemAttrDefaults* on page J1-5415.
- *shared/translation/attrs/S1CacheDisabled* on page J1-5416.
- *shared/translation/attrs/S2AttrDecode* on page J1-5416.
- *shared/translation/attrs/S2CacheDisabled* on page J1-5417.
- *shared/translation/attrs/S2ConvertAttrsHints* on page J1-5417.
- *shared/translation/attrs/ShortConvertAttrsHints* on page J1-5417.
- *shared/translation/attrs/WalkAttrDecode* on page J1-5418.
- *shared/translation/translation/PAMax* on page J1-5418.
- *shared/translation/translation/S1TranslationRegime* on page J1-5418.

## shared/translation/attrs/CombineS1S2AttrHints

```
// CombineS1S2AttrHints()
// ======================

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc)

    MemAttrHints result;

    if s2desc.attrs == '01' || s1desc.attrs == '01' then
        result.attrs = bits(2) UNKNOWN;   // Reserved
    elsif s2desc.attrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
        result.attrs = MemAttr_NC;        // Non-cacheable
    elsif s2desc.attrs == MemAttr_WT || s1desc.attrs == MemAttr_WT then
        result.attrs = MemAttr_WT;        // Write-through
    else
        result.attrs = MemAttr_WB;        // Write-back

    result.hints = s1desc.hints;
    result.transient = s1desc.transient;

    return result;
```

## shared/translation/attrs/CombineS1S2Desc

```
// CombineS1S2Desc()
// =================
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    AddressDescriptor result;

    result.paddress = s2desc.paddress;

    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    elsif s2desc.mematttrs.type == MemType_Device || s1desc.mematttrs.type == MemType_Device then
        result.mematttrs.type = MemType_Device;
        if s1desc.mematttrs.type == MemType_Normal then
            result.mematttrs.device = s2desc.mematttrs.device;
        elsif s2desc.mematttrs.type == MemType_Normal then
            result.mematttrs.device = s1desc.mematttrs.device;
        else                    // Both Device
            result.mematttrs.device = CombineS1S2Device(s1desc.mematttrs.device,
                                                        s2desc.mematttrs.device);
    else                        // Both Normal
        result.mematttrs.type = MemType_Normal;
        result.mematttrs.device = DeviceType UNKNOWN;
        result.mematttrs.inner = CombineS1S2AttrHints(s1desc.mematttrs.inner, s2desc.mematttrs.inner);
        result.mematttrs.outer = CombineS1S2AttrHints(s1desc.mematttrs.outer, s2desc.mematttrs.outer);
        result.mematttrs.shareable = (s1desc.mematttrs.shareable || s2desc.mematttrs.shareable);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
        result.memattrs.outershareable = (s1desc.memattrs.outershareable ||
                                          s2desc.memattrs.outershareable);

    result.memattrs = MemAttrDefaults(result.memattrs);

    return result;
```

### shared/translation/attrs/CombineS1S2Device

```
// CombineS1S2Device()
// ===================
// Combines device types from stage 1 and stage 2

DeviceType CombineS1S2Device(DeviceType s1device, DeviceType s2device)

    if s2device == DeviceType_nGnRnE || s1device == DeviceType_nGnRnE then
        result = DeviceType_nGnRnE;
    elsif s2device == DeviceType_nGnRE || s1device == DeviceType_nGnRE then
        result = DeviceType_nGnRE;
    elsif s2device == DeviceType_nGRE || s1device == DeviceType_nGRE then
        result = DeviceType_nGRE;
    else
        result = DeviceType_GRE;

    return result;
```

### shared/translation/attrs/LongConvertAttrsHints

```
// LongConvertAttrsHints()
// =======================
// Convert the long attribute fields for Normal memory as used in the MAIR fields
// to orthogonal attributes and hints

MemAttrHints LongConvertAttrsHints(bits(4) attrfield, AccType acctype)
    assert !IsZero(attrfield);

    MemAttrHints result;

    if S1CacheDisabled(acctype) then             // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        if attrfield<3:2> == '00' then           // Write-through transient
            result.attrs = MemAttr_WT;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        elsif attrfield<3:0> == '0100' then      // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
            result.transient = FALSE;
        elsif attrfield<3:2> == '01' then        // Write-back transient
            result.attrs = attrfield<1:0>;
            result.hints = MemAttr_WB;
            result.transient = TRUE;
        else                                     // Write-through/Write-back non-transient
            result.attrs = attrfield<3:2>;
            result.hints = attrfield<1:0>;
            result.transient = FALSE;

    return result;
```

### shared/translation/attrs/MemAttrDefaults

```
// MemAttrDefaults()
// =================
// Supply default values for memory attributes, including overriding the shareability attributes
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
// for Device and Non-cacheable memory types.

MemoryAttributes MemAttrDefaults(MemoryAttributes mematTrs)

    if mematTrs.type == MemType_Device then
        mematTrs.inner = MemAttrHints UNKNOWN;
        mematTrs.outer = MemAttrHints UNKNOWN;
        mematTrs.shareable = TRUE;
        mematTrs.outershareable = TRUE;
    else
        mematTrs.device = DeviceType UNKNOWN;
        if mematTrs.inner.attrs == MemAttr_NC && mematTrs.outer.attrs == MemAttr_NC then
            mematTrs.shareable = TRUE;
            mematTrs.outershareable = TRUE;

    return mematTrs;
```

## shared/translation/attrs/S1CacheDisabled

```
// S1CacheDisabled()
// =================

boolean S1CacheDisabled(AccType acctype)
    if ELUsingAArch32(S1TranslationRegime()) then
        if PSTATE.EL == EL2 then
            enable = if acctype == AccType_IFETCH then HSCTLR.I else HSCTLR.C;
        else
            enable = if acctype == AccType_IFETCH then SCTLR.I else SCTLR.C;
    else
        enable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;

    return enable == '0';
```

## shared/translation/attrs/S2AttrDecode

```
// S2AttrDecode()
// ==============
// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)

    MemoryAttributes mematTrs;

    if attr<3:2> == '00' then                    // Device
        mematTrs.type = MemType_Device;
        case attr<1:0> of
            when '00'  mematTrs.device = DeviceType_nGnRnE;
            when '01'  mematTrs.device = DeviceType_nGnRE;
            when '10'  mematTrs.device = DeviceType_nGRE;
            when '11'  mematTrs.device = DeviceType_GRE;

    elsif attr<1:0> != '00' then                // Normal
        mematTrs.type = MemType_Normal;
        mematTrs.outer = S2ConvertAttrsHints(attr<3:2>, acctype);
        mematTrs.inner = S2ConvertAttrsHints(attr<1:0>, acctype);
        mematTrs.shareable = SH<1> == '1';
        mematTrs.outershareable = SH == '10';

    else
        mematTrs = MemoryAttributes UNKNOWN;    // Reserved

    return MemAttrDefaults(mematTrs);
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### shared/translation/attrs/S2CacheDisabled

```
// S2CacheDisabled()
// =================

boolean S2CacheDisabled(AccType acctype)
    if ELUsingAArch32(EL2) then
        disable = if acctype == AccType_IFETCH then HCR2.ID else HCR2.CD;
    else
        disable = if acctype == AccType_IFETCH then HCR_EL2.ID else HCR_EL2.CD;

    return disable == '1';
```

### shared/translation/attrs/S2ConvertAttrsHints

```
// S2ConvertAttrsHints()
// =====================
// Converts the attribute fields for Normal memory as used in stage 2
// descriptors to orthogonal attributes and hints

MemAttrHints S2ConvertAttrsHints(bits(2) attr, AccType acctype)
    assert !IsZero(attr);

    MemAttrHints result;

    if S2CacheDisabled(acctype) then   // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case attr of
            when '01'                  // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '10'                  // Write-through
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RWA;
            when '11'                  // Write-back
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;

    result.transient = FALSE;

    return result;
```

### shared/translation/attrs/ShortConvertAttrsHints

```
// ShortConvertAttrsHints()
// ========================
// Converts the short attribute fields for Normal memory as used in the TTBR and
// TEX fields to orthogonal attributes and hints

MemAttrHints ShortConvertAttrsHints(bits(2) RGN, AccType acctype)

    MemAttrHints result;

    if S1CacheDisabled(acctype) then     // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case RGN of
            when '00'                  // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '01'                  // Write-back, Read and Write allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
            when '10'                   // Write-through, Read allocate
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RA;
            when '11'                   // Write-back, Read allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RA;

    result.transient = FALSE;

    return result;
```

### shared/translation/attrs/WalkAttrDecode

```
// WalkAttrDecode()
// ================

MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN)

    MemoryAttributes mematTrs;

    AccType acctype = AccType_NORMAL;

    mematTrs.type = MemType_Normal;
    mematTrs.inner = ShortConvertAttrsHints(IRGN, acctype);
    mematTrs.outer = ShortConvertAttrsHints(ORGN, acctype);
    mematTrs.shareable = SH<1> == '1';
    mematTrs.outershareable = SH == '10';

    return MemAttrDefaults(mematTrs);
```

### shared/translation/translation/PAMax

```
// PAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the physical address
// size for this processor, as log2().

integer PAMax()

    case ID_AA64MMFR0_EL1.PARange of
        when '0000'  pa_size = 32;
        when '0001'  pa_size = 36;
        when '0010'  pa_size = 40;
        when '0011'  pa_size = 42;
        when '0100'  pa_size = 44;
        when '0101'  pa_size = 48;
        otherwise    Unreachable();

    return pa_size;
```

### shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    if PSTATE.EL != EL0 then
        return PSTATE.EL;
    elsif IsSecure() && HaveEL(EL3) && ELUsingAArch32(EL3) then
        return EL3;
    else
        return EL1;
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Part K

## Appendixes

# Appendix K1
# Architectural Constraints on UNPREDICTABLE behaviors

This chapter describes the architectural constraints on UNPREDICTABLE behaviors in the ARMv8 architecture. It contains the following sections:


- *AArch32 CONSTRAINED UNPREDICTABLE behaviors* on page K1-5422.
- *Constraints on AArch64 state UNPREDICTABLE behaviors* on page K1-5501.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors

ARMv8 defines architecturally-required constraints on many behaviors that are UNPREDICTABLE in ARMv7. The following sections define those constraints:


- *Overview of the constraints on ARMv7 UNPREDICTABLE behaviors* on page K1-5423.
- *Using R13* on page K1-5423.
- *Using R15* on page K1-5423.
- *Branching into an IT block* on page K1-5424.
- *Branching to an unaligned PC* on page K1-5424.
- *Loads and Stores to unaligned locations* on page K1-5424.
- *CONSTRAINED UNPREDICTABLE instructions in an IT block* on page K1-5425.
- *Unallocated CP14 and CP15 instructions* on page K1-5426.
- *SBZ or SBO fields in instructions* on page K1-5426.
- *Immediate constants in T32 data processing instructions* on page K1-5427.
- *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.
- *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5427.
- *CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization* on page K1-5427
- *Translation Table Base Address alignment* on page K1-5428.
- *Handling of conceptual coprocessors CP10 and CP11* on page K1-5428.
- *Performance Monitors Extension* on page K1-5428.
- *Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED* on page K1-5430.
- *Out of range virtual address* on page K1-5430.
- *Instruction fetches from Device memory* on page K1-5431.
- *Multi-access instructions that load the PC from Device memory* on page K1-5431.
- *Programming CSSELR.Level for a cache level that is not implemented* on page K1-5431.
- *Crossing a page boundary with different memory types or Shareability attributes* on page K1-5431.
- *Crossing a 4KB boundary with a Device access* on page K1-5432.
- *CONSTRAINED UNPREDICTABLE behavior for memory hints, Advanced SIMD instructions, and miscellaneous A32 instructions* on page K1-5432.
- *Out of range values of the Set/Way/Index fields in cache maintenance instructions* on page K1-5432.
- *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instructions in the base instruction set* on page K1-5433.
- *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 system instructions in the base instruction set* on page K1-5478.
- *CONSTRAINED UNPREDICTABLE behavior, A32 and T32 Advanced SIMD and floating-point instructions* on page K1-5485.
- *CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR* on page K1-5496.



Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- *CONSTRAINED UNPREDICTABLE behavior in Debug state* on page K1-5496.
- *CONSTRAINED UNPREDICTABLE behavior of EL2 features* on page K1-5497.

### K1.1.1 Overview of the constraints on ARMv7 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch32 state, where previous versions of the architecture define behavior as UNPREDICTABLE, the ARMv8-A architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

**Note**

Software designed to be compatible with the ARMv8-A architecture must not rely on these CONSTRAINED UNPREDICTABLE cases.

### K1.1.2 Using R13

In prior versions of the architecture, the use of R13 as a named register specifier was described as UNPREDICTABLE in the pseudocode. In the ARMv8-A architecture, the use of R13 as a named register specifier is not UNPREDICTABLE, unless this is specifically stated, and R13 can be used in the regular form. Bits[1:0] of R13 are not treated as RES0, but can hold any values programmed into them.

### K1.1.3 Using R15

All uses of R15 as a named register specifier for a source register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual must do one of the following:

- Cause the instruction to be treated as an UNDEFINED instruction.
- Cause the instruction to execute as a `NOP`.
- Read the PC with the standard offset that applies for the current instruction set.
- Read the PC with the standard offset that applies for the current instruction set with alignment to a word boundary.
- Read 0.
- Read an UNKNOWN value.

All uses of R15 as a named register specifier for a destination register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a `NOP`.
- Ignore the write.
- Branch to an UNKNOWN location in either A32 or T32 state.

The choice between these behaviors might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

Instructions that are CONSTRAINED UNPREDICTABLE when the base register is R15 and the instruction specifies a writeback of the base register, are treated as having R15 as both a source register and a destination register.

For instructions that have two destination registers, for example `LDRD`, `MRRC`, and many of the multiply instructions, if Rt, Rt2, RdLo, or RdHi is R15, then the other destination register of the pair is UNKNOWN, if the CONSTRAINED UNPREDICTABLE behavior for the write to R15 is either to ignore the write or to branch to an UNKNOWN location.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For instructions that affect any or all of PSTATE.{N, Z, C, V}, PSTATE.Q, and PSTATE.GE when the register specifier is not R15, any flags affected by an instruction that is CONSTRAINED UNPREDICTABLE when the register specifier is R15 become UNKNOWN,

In addition, for MRC instructions for CP14 and CP15 that use R15 as the destination register descriptor, and thereby target APSR_nzcv where these are described as being CONSTRAINED UNPREDICTABLE, PSTATE.{N, Z, C, V} becomes UNKNOWN.

### K1.1.4 Branching into an IT block

Branching into an IT block leads to CONSTRAINED UNPREDICTABLE behavior. Execution starts from the address determined by the branch, but each instruction in the IT block is:

- Executed as if it were not in an IT block. This means that it is executed unconditionally.
- Executed as if it had passed its Condition code check within an IT block.
- Executed as a NOP. That is, it behaves as if it had failed the Condition code check.

### K1.1.5 Branching to an unaligned PC

In A32 state, when branching to an address that is not word aligned and is defined to be CONSTRAINED UNPREDICTABLE, one of the following behaviors must occur:

- The unaligned location is forced to be aligned.
- The unaligned address generates an exception on the first instruction using the unaligned PC value.

  If that instruction is executed at EL0 and either of the following applies, the exception is taken to EL2:
  - EL2 is using AArch32 and the value of HCR.TGE is 1
  - EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

  If the instruction is executed at EL0 when the applicable TGE bit is 0 the exception is taken to EL1.

  If the instruction is executed at an Exception level that is higher than EL0 the exception is taken to the Exception level at which the instruction was executed.

  In all cases, the exception is generated only if the first instruction using the unaligned PC value is architecturally executed.

If the exception that results from a branch to an unaligned PC value:

- Is taken to an Exception level that is using AArch64, it is reported as a Misaligned PC exception, see *Exception from an Illegal Execution state, Misaligned PC, or Misaligned stack pointer* on page D1-1546.
- Is taken to an Exception level that is using AArch32, it is reported as a Prefetch Abort exception, see *Prefetch Abort exception reporting a Misaligned PC exception* on page G1-3819.

**Note**

Because bit[0] is used for interworking, it is impossible to specify a branch to A32 state when the bottom bit of the target address is 1. Therefore the bottom bit of IFAR, HIFAR, or FAR_ELx is 0 for all these cases.

### K1.1.6 Loads and Stores to unaligned locations

Some unaligned loads and stores in the ARMv7 architecture are described as UNPREDICTABLE. These are defined in the ARMv8-A architecture to do one of the following:

- Take an alignment fault.
- Perform the specified load or store to the unaligned memory location.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K1.1.7 CONSTRAINED UNPREDICTABLE instructions in an IT block

A number of instructions in the architecture are described as being CONSTRAINED UNPREDICTABLE either:

- Anywhere within an IT block.
- As an instruction within an IT block, other than the last instruction within an IT block.

Unless otherwise stated in this manual, when these instructions are committed for execution, one of the following occurs:

- An UNDEFINED exception results.
- The instructions are executed as if they had passed the Condition code check.
- The instructions execute as NOPs. This means that they behave as if they had failed the Condition code check.

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

Many instructions that are CONSTRAINED UNPREDICTABLE in an IT block are branch instructions or other non-sequential instructions that change the PC. Where these instructions are not treated as UNDEFINED within an IT block, the remaining iterations of the PSTATE.IT state machine must be treated in one of the following ways:

- PSTATE.IT is cleared to 0.
- PSTATE.IT advances for either a sequential or a nonsequential change of the PC in the same way as it does for instructions that are not CONSTRAINED UNPREDICTABLE that cause a sequential change of the PC.

**Note**

This does not apply to an instruction that is the last instruction in an IT block.

The instructions addressed by the updated PC must do one of the following:

- Execute as if they had passed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as NOPs. That is, they behave as if they had failed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as if they were unconditional, or, if the instructions are part of another IT block, in accordance with the behavior described in *Branching into an IT block* on page K1-5424.

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

For exception returns or Debug state exits that cause PSTATE.IT to be set to a reserved value in T32 state or that return to A32 state with a nonzero value in PSTATE.IT, the PSTATE.IT bits are forced to '00000000'.

The reserved values are:

```
PSTATE.IT[7:4] != '0000' && PSTATE.IT[3:0] = '0000'
PSTATE.IT[2:0] != '000' when SCTLR/SCTLR_EL_1.ITD == '1'
```

Exception returns or Debug state exits that set PSTATE.IT to a non-reserved value in T32 state can occur when the flow of execution returns to a point:

- Outside an IT block, but with the PSTATE.IT bits set to a value other than '00000000'.
- Inside an IT block, but with a different value of the PSTATE.IT bits than if the IT block had been executed without an exception return or Debug state exit.

In this case the instructions at the target of the exception return or Debug state exit must do one of the following:

- Execute as if they passed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as NOPs. That is, they behave as if they failed the Condition code check for the remaining iterations of the PSTATE.IT state machine.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- Execute as if they were unconditional, or as if the instruction were part of another IT block, in accordance with the behavior in *Branching into an IT block* on page K1-5424.

The remaining iterations of the PSTATE.IT state machine must behave in one of the following ways:

- The PSTATE.IT state machine advances as if it were in an IT block.
- The PSTATE.IT bits are ignored.
- The PSTATE.IT bits are forced to `'00000000'`.

### K1.1.8 Unallocated CP14 and CP15 instructions

In ARMv8-A, accesses to unallocated CP14 and CP15 register encodings are UNDEFINED.

This includes:

- Reads using encodings that are defined as WO.
- Writes using encodings that are defined as RO.
- `MCR` or `MRC` accesses to using a set of {`CRn`, `opc1`, `CRm`, `opc2`} values that the ARMv7 architecture defined as UNPREDICTABLE.
- CP14 or CP15 accesses that the ARMv7 architecture defined as UNPREDICTABLE when particular functionality was not implemented, when an implementation does not include the Exception level that provides that functionality in ARMv8.

### K1.1.9 SBZ or SBO fields in instructions

|Many of the A32 and T32 instructions have (0) or (1) in the instruction decode to indicate *should-be-zero*, SBZ, or *should-be-one*, SBO. Except for the specific cases called out in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instructions in the base instruction set* on page K1-5433, if the instruction bit pattern of an instruction is executed with these fields not having the *should be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.

The exceptions to this rule are:

- *LDM/LDMIA/LDMFD (T32)* on page K1-5437.
- *LDMDB/LDMEA* on page K1-5439.
- *LDR (literal)* on page K1-5440.
- *LDRB (literal)* on page K1-5442.
- *LDRH (literal)* on page K1-5447.
- *LDRSB (literal)* on page K1-5448.
- *LDRSH (literal)* on page K1-5450.
- *LDRD (immediate)* on page K1-5443.
- *LDRD (register)* on page K1-5444.
- *LDRD (literal)* on page K1-5444.
- *POP (T32)* on page K1-5452.
- *POP (A32)* on page K1-5452.
- *PUSH* on page K1-5453.
- *SDIV* on page K1-5456.
- *UDIV* on page K1-5477.
- *STM (STMIA, STMEA)* on page K1-5460.
- *STMDB (STMFD)* on page K1-5461.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K1.1.10 Immediate constants in T32 data processing instructions

The immediate constants in the T32 data processing instructions describe immediate values generated by hw2[7:0] == 0000000 as being UNPREDICTABLE when

- Hw1[10] == 0 and hw2[14:12] == 001.
- Hw1[10] == 0 and hw2[14:12] == 010.
- Hw1[10] == 0 and hw2[14:12] == 011.

For the ARM architecture, these encodings produce the value `0b0000000`.

### K1.1.11 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated, all unallocated or reserved values of fields with allocated values within the AArch32 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.
- The encoding causes the field to have no functional effect.

**Note**

These constraints are identical to those for the equivalent AArch64 definitions, as given in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5501.

### K1.1.12 CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values

The ARM architecture allows copies of control values or data values to be cached in a cache or TLB. This can lead to CONSTRAINED UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

**Note**

This rules applies where inadequate invalidation of the TLB might cause multiple hits within the TLB. In this situation, a failure to invalidate the TLB by code running at a given Privilege level must not make access to regions of memory with permissions or attributes that could not be accessed at that Privilege level possible.

Alternatively, an implementation might generate a Data Abort exception when detecting multiple hits within a TLB, using the TLB Conflict fault code.

The choice between these behaviors might, in some implementations, vary for each use of a control or data value.

### K1.1.13 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The ARM architecture requires that changes to System registers must be synchronized before they take effect. This can lead to CONSTRAINED UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the processor is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

### K1.1.14 Translation Table Base Address alignment

A misaligned Translation Table Base Address can occur if:

- The VMSAv8-32 Short-descriptor translation table format is enabled and TTBR0[13-N:7], which is defined to be RES0, contains one or more nonzero values.
- The VMSAv8-32 Long-descriptor translation table format is enabled, and TTBR0[x-1:3], TTBR1[x-1:3], HTTBR[x-1:3], or VTTBR[x-1:3], which are defined to be RES0, contain one or more nonzero values.

In the event of a misaligned Translation Table Base Address, one of the following behaviors must occur:

- The field that is defined to be RES0 is treated as if all bits were zero:
  - The value that is read back might be the value written or it might be zero.
- The calculation of an address for a translation table walk using that register might be corrupted in those bits that are nonzero.

### K1.1.15 Handling of conceptual coprocessors CP10 and CP11

The floating-point instruction set covers both of the conceptual coprocessors CP10 and CP11. This means that CPACR, HCPTR, and NSACR all have control bits whose naming reflects the {CP10, CP11} coprocessors. If these control bits differ, then the behavior is the same as if the control bit for CP11 were equal to the control bit for CP10 in all respects, other than the value read back by a direct read of the CP11 control bit. After a register write that writes different values to the CP10 and CP11 control bits, a direct read of the register might return an UNKNOWN value for the CP11 control bit.

**Note**

This means that, when different values are written to the CP10 and CP11 control bits in a single register, the architecture permits but does not require that a read of that register returns the value written to the CP11 control bit.

#### CONSTRAINED UNPREDICTABLE CPACR and NSACR settings

If CPACR.cp<n> contains the encoding `'10'`, then one of the following behaviors must occur:

- The encoding maps onto any of the allocated values, but otherwise does not cause UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.

**Note**

In ARMv7, CPACR had a D32DIS bit, and NSACR had an NSD32DIS bit. There is no CPACR.D32DIS or NSACR.NSD32DIS in ARMv8-A, and the corresponding bits in the two registers are RES0.

### K1.1.16 Performance Monitors Extension

The following subsections describe CONSTRAINED UNPREDICTABLE behaviors in AArch32 use of the Performance Monitors Extension:

- *CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR* on page K1-5429.
- *CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n>* on page K1-5429.
- *CONSTRAINED UNPREDICTABLE behavior caused by HDCR.HPMN* on page K1-5430.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR

The following cases can cause CONSTRAINED UNPREDICTABLE behavior:

- If PMSELR.SEL is not equal to 31, and PMSELR.SEL is greater than or equal to PMCR.N, and the PE is executing in Secure state or in Non-secure Hyp mode.
- If PMSELR.SEL is not 31, and PMSELR.SEL is greater than or equal to HDCR.HPMN, and the PE is executing in a Non-secure mode other than Hyp mode.

In these UNPREDICTABLE cases, one of the following behaviors must occur:

- Accesses to PMXEVTYPER or PMXEVCNTR from that mode are UNDEFINED.
- Accesses to PMXEVTYPER or PMXEVCNTR from that mode behave as RAZ/WI.
- Accesses to PMXEVTYPER or PMXEVCNTR from that mode execute as NOPs.
- Accesses to PMXEVTYPER or PMXEVCNTR from that mode behave as if PMSELR.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- Accesses to PMXEVTYPER or PMXEVCNTR behave as if PMSELR.SEL is 31.
- In Non-secure state, for an access to PMXEVTYPER or PMXEVCNTR from PL1 or a permitted access from PL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2. Accesses from PL0 are permitted when:
  - EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  - EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

If PMSELR.SEL is equal to 31, then one of the following behaviors must occur:

- Accesses to PMXEVCNTR are UNDEFINED.
- Accesses to PMXEVCNTR behave as RAZ/WI.
- Accesses to PMXEVCNTR execute as NOPs.
- Accesses to PMXEVCNTR behave as if PMSELR.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- In Non-secure state, for an access to PMXEVCNTR from PL1 or a permitted access from PL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2. Accesses from PL0 are permitted when:
  - EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  - EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

**Note**

In an implementation that includes EL2, in Non-secure state at PL0 and PL1, HDCR.HPMN, or MDCR_EL2.HPMN, identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters.

### CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n>

If <n> is greater than the number of counters available in the current Exception level and state, reads and writes of PMEVCNTR<n> and PMEVTYPER<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- In Non-secure state, for an access to PMEVCNTR<n> or PMEVTYPER<n> from PL1 or a permitted access from PL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2. Accesses from PL0 are permitted when:
  - EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  - EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

**Note**

In an implementation that includes EL2, in Non-secure state at PL0 and PL1, HDCR.HPMN, or MDCR_EL2.HPMN, identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of implemented counters.

#### CONSTRAINED UNPREDICTABLE behavior caused by HDCR.HPMN

If HDCR.HPMN is set to 0 or to a value greater than PMCR.N, then the CONSTRAINED UNPREDICTABLE behavior is:

- The value returned by a direct read of HDCR.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if HDCR.HPMN is set to an UNKNOWN non-zero value less than PMCR.N.
  - All counters are reserved for EL2 use, meaning no counters are accessible from Non-secure EL1 and Non-secure EL0.

### K1.1.17 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, this generates an exception:

- If this exception is taken to an Exception level that is using AArch64 then ESR_ELx is UNKNOWN.
- If this exception is taken to EL2 and EL2 is using AArch32, then the HSR is UNKNOWN.

**Note**

The value written to ESR or HSR must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but resulted from a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

### K1.1.18 Out of range virtual address

If the PE executes an instruction for which the instruction address, size, and alignment mean it contains the bytes `0xFFFF FFFF` and `0x0000 0000`, then the bytes that wrap around and appear to be from `0x0000 0000` onwards come from an UNKNOWN address.

If the PE executes a load or store instruction for which the computed address, total access size, and alignment mean it accesses bytes `0xFFFF FFFF` and `0x0000 0000`, then the bytes that wrap around and appear to be from `0x0000 0000` onwards come from an UNKNOWN address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K1.1.19 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location when address translation is enabled.

If a branch causes the program counter to point to a location in memory with the Device attribute that is not marked as execute-never for the current Exception level for instruction fetches, then an implementation must perform one of the following behaviors:

- It treats the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- It generates a Permission fault.

### K1.1.20 Multi-access instructions that load the PC from Device memory

Multi-access instructions that load the PC from Device memory when address translation is enabled are UNPREDICTABLE in AArch32 state. In the ARMv8-A architecture in AArch32, an implementation must perform one of the following behaviors:

- It loads the PC from the memory location as if the memory location had the Normal Non-cacheable attribute.
- It generates a permission fault.

### K1.1.21 Programming CSSELR.Level for a cache level that is not implemented

If CSSELR.Level is programmed to a cache level that is not implemented, then a read of CSSELR returns an UNKNOWN value in CSSELR.Level.

If the CSSELR.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR an implementation must perform one of the following behaviors:

- The CCSIDR read executes as a `NOP`.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.

### K1.1.22 Crossing a page boundary with different memory types or Shareability attributes

A memory access from a load or store instruction that crosses a page boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- All memory accesses generated by the instruction use the memory type and Shareability attributes associated with the first address accessed by the instruction.
- All memory accesses generated by the instruction use the memory type and Shareability attributes associated with the last address accessed by the instruction.
- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an alignment fault caused by the memory type.

  For the Non-secure PL1&0 translation regime:

  — If the stage 1 translation causes the mismatch then the resulting exception is taken to PL1.

  — If the stage 2 translation causes the mismatch then the resulting exception is taken to PL2.

  — If both stages of translation cause the mismatch then the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a `NOP`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K1.1.23 Crossing a 4KB boundary with a Device access

A memory access from a load or store instruction to Device memory that crosses a 4KB boundary results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses, except that there is no guarantee of ordering between memory accesses.
- The instruction generates an Alignment fault caused by the memory type.

  For the Non-secure PL1&0 translation regime:

  — If the stage 1 translation causes the boundary to be crossed then the resulting exception is taken to PL1.

  — If the stage 2 translation causes the boundary to be crossed then the resulting exception is taken to PL2.

  — If both stages of translation cause the boundary to be crossed then the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a `NOP`.

**Note**

The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.

### K1.1.24 CONSTRAINED UNPREDICTABLE behavior for memory hints, Advanced SIMD instructions, and miscellaneous A32 instructions

A number of memory hints, Advanced SIMD instructions, and miscellaneous A32 instructions can result in CONSTRAINED UNPREDICTABLE behavior.

If an instructions listed in *Unconditional instructions* on page F4-2541 is CONSTRAINED UNPREDICTABLE but does not have an encoding of `Op1= 101x001`, `Op2 = -`, and `Rn = 1111`, then an implementation must treat these encodings in one of the following ways:

- The encoding is UNDEFINED.
- The instruction executes as a `NOP`.

When `Op1= 101x001`, `Op2 = -`, and `Rn = 1111`, *PLD (literal)* on page F6-2868 describes the behavior. This encoding is subject to the rules outlined in *SBZ or SBO fields in instructions* on page K1-5426.

### K1.1.25 Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions DCCISW, DCCSW, and DCISW, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:

  — No cache lines.

  — A single arbitrary cache line.

  — Multiple arbitrary cache lines.

**Note**

This CONSTRAINED UNPREDICTABLE behavior applies, also, to the A64 cache maintenance by set/way instructions DC CISW, DC CSW, and DC ISW.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## K1.1.26 CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instructions in the base instruction set

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A32 and T32 instructions listed in *Alphabetical list of T32 and A32 base instruction set instructions* on page F6-2576.

**Note**

- The pseudocode used in this section to describe cases that can result in CONSTRAINED UNPREDICTABLE behavior does not necessarily match the encoding specific pseudocode for a specific instruction.
- If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

### BFC

For a description of these instructions and the encodings, see *BFC* on page F6-2626.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The output value in the registers is UNKNOWN.

### BFI

For a description of these instructions and the encodings, see *BFI* on page F6-2628.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The output value in the registers is UNKNOWN.

### BKPT

For a description of this instruction and the encoding, see *BKPT* on page F6-2637.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `cond != '1110' && cond != '1111'`, then one of the following behaviors occurs:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction executes unconditionally.
  - The instruction executes conditionally.

For the treatment of the case where `cond == '1111'` see *CONSTRAINED UNPREDICTABLE behavior for memory hints, Advanced SIMD instructions, and miscellaneous A32 instructions* on page K1-5432.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## CLZ

For a description of this instruction and the encoding, see *CLZ* on page F6-2650.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T1 encoding:

- If `!consistent(Rm)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The register specified by hw1[3:0] is used as the source register.
  - The register specified by hw2[3:0] is used a the source register.
  - The value in the destination register is UNKNOWN.

## CMP (register)

For a description of this instruction and the encoding, see *CMP (register)* on page F6-2658.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T2 encoding:

- If `n < 8 && m < 8`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the comparison between Rn and Rm.
  - The condition flags become UNKNOWN.

## CRC32, CRC32C

For a description of this instruction and the encoding, see *CRC32* on page F6-2665 and *CRC32C* on page F6-2667.

### *CONSTRAINED UNPREDICTABLE behavior*

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction behaves as if `sz == '10'`.

For the A1 encoding:

- If `cond != '1110'`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction is executed unconditionally.
- The instruction is executed conditionally.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## HLT

For a description of this instruction and the encoding, see *HLT* on page F6-2685.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `cond != '1110' && cond != '1111'`, then one of the following behaviors occurs:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction is executed unconditionally.
  - The instruction is executed conditionally.

For the treatment of the case where `cond == '1111'` see *CONSTRAINED UNPREDICTABLE behavior for memory hints, Advanced SIMD instructions, and miscellaneous A32 instructions* on page K1-5432.

## IT

For a description of this instruction and the encoding, see *IT* on page F6-2690.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T1 encoding:

- If `firstcond == '1111'||(firstcond == '1110' && BitCount(mask)!= 1)` then one of the following behaviors occurs:
  - The instruction is undefined.
  - The instruction executes as a `NOP`.
  - `firstcond == '1111'` is treated the same as `firstcond` == '1110', meaning the always condition, and the PSTATE.IT state machine is progressed in the same way as for any other `cond_base` value.

## LDAEX

For a description of this instruction and the encoding, see *LDAEX* on page F6-2694.

### *CONSTRAINED UNPREDICTABLE behavior*

If the Load-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.

## LDAEXB

For a description of this instruction and the encoding, see *LDAEXB* on page F6-2696.

### *CONSTRAINED UNPREDICTABLE behavior*

If the Load-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.

## LDAEXD

For a description of this instruction and the encoding, see *LDAEXD* on page F6-2698.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If the Load-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the T1 encoding:

- If `t == t2`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

For the A1 encoding:

- If `Rt<0>=='1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The behavior is the same as if the `Rt<0>=='0'`.
  - The registers accessed are both specified by Rt. This means that Rt and Rt+1 are the same, and both have bit[0] == 1.
  - The registers accessed are specified by Rt and Rt+1.

**Note**

If `t2 == 15`, then the requirements described in *Using R15* on page K1-5423 apply.

## LDAEXH

For a description of this instruction and the encoding, see *LDAEXH* on page F6-2700.

### *CONSTRAINED UNPREDICTABLE behavior*

If the Load-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.

## LDC/LDC2 (literal)

For a description of this instruction and the encoding, see *LDC, LDC2 (literal)* on page F6-2708.

### *CONSTRAINED UNPREDICTABLE behavior*

If `W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_ARM)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The load instruction operates without writeback.

  **Note**

  This is consistent with ignoring writes to the PC.

- The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, see *Using R15* on page K1-5423.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## LDM/LDMIA/LDMFD (A32)

For a description of this instruction and the encoding, see *LDM, LDMIA, LDMFD* on page F6-2711.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.
- If `wback && registers<n> == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode, and the content of the register being written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

## LDM/LDMIA/LDMFD (T32)

For a description of this instruction and the encoding, see *LDM, LDMIA, LDMFD* on page F6-2711.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T1 encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For the T2 encoding:

- If `BitCount(registers) < 2`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction loads a single register using the specified addressing modes.
  - The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.
- If hw2[13] is set to 1, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

    **Note**

    This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If `P == '1' && M == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.
- If `wback && registers<n> == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode, and the content of the register being written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

## LDMDA/LDMFA

For a description of this instruction and the encoding, see *LDMDA, LDMFA* on page F6-2718.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `LDM` with the same addressing mode, but targeting an unspecified set of registers. These registers might include R15.
- If `wback && registers<n> == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

## LDMIB/LDMED

For a description of this instruction and the encoding, see *LDMIB, LDMED* on page F6-2722.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `LDM` with the same addressing mode, but targeting an unspecified set of registers. These registers might include R15.
- If `wback && registers<n> == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## LDMDB/LDMEA

For a description of this instruction and the encoding, see *LDMDB, LDMEA* on page F6-2720.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `wback && registers<n> == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For the A1 encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED
  - The instruction executes as a `NOP`.
  - The instruction operates as an `LDM` with the same addressing mode, but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For the T1 encoding:

- If `BitCount(registers) < 2`, one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction loads a single register using the specified addressing modes.
  - The instruction operates as an `LDM` with the same addressing mode, but targeting an unspecified set of registers. These registers might include R15.
- If hw2[13] is set to 1, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

  **Note**

  This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.
- If `P == '1' && M == '1'`, then one of the following behaviors must occur:
  - The instruction is undefined.
  - The instruction executes as a `NOP`.
  - The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### LDR (immediate, A32)

For a description of this instruction and the encoding, see *LDR (immediate)* on page F6-2724.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

### LDR (immediate, T32)

For a description of this instruction and the encoding, see *LDR (immediate)* on page F6-2724.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T4 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

### LDR (literal)

For a description of this instruction and the encoding, see *LDR (literal)* on page F6-2728.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If bit[24] has the same value as bit[21]:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP`.
  - The instruction is treated as if bit[24] had value of 1 and bit[21] had a value of 0.
  - The instruction treats bit[24] as the P bit, and bit[21] as the writeback bit, and uses the same addressing mode as described in the immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    This is an exception to the principle in *SBZ or SBO fields in instructions* on page K1-5426.

- If bit[24] == 0 and bit[21] == 1, then the instruction can be handled according to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426 or the requirements described in *LDR (immediate, T32)*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### LDR (register, A32)

For a description of this instruction and the encoding, see *LDR (register)* on page F6-2730.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

### LDRB (immediate, A32)

For a description of this instruction and the encoding, see *LDRB (immediate)* on page F6-2733.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

### LDRB (immediate, T32)

For a description of this instruction and the encoding, see *LDRB (immediate)* on page F6-2733.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T3 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### LDRB (literal)

For a description of this instruction and the encoding, see *LDRB (literal)* on page F6-2736.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If bit[24] has the same value as bit[21]:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP`.
  - The instruction is treated as if bit[24] had value of 1 and bit[21] had a value of 0.
  - The instruction treats bit[24] as the P bit, and bit[21] as the writeback bit, and uses the same addressing mode as described in the immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    This is an exception to the principle in *SBZ or SBO fields in instructions* on page K1-5426.

- If bit[24] == 0 and bit[21] == 1, then the instruction can be handled according to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426 or the requirements described in *LDRB (immediate, T32)* on page K1-5441.

### LDRB (register)

For a description of this instruction and the encoding, see *LDRB (register)* on page F6-2738.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

### LDRBT

For a description of this instruction and the encoding, see *LDRBT* on page F6-2740.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 and A2 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

- When the Rn field specifies R15, then this instruction can be treated either as described in this section or as described in *LDRB (literal)* on page K1-5442.

## LDRD (immediate)

For a description of this instruction and the encoding, see *LDRD (immediate)* on page F6-2742.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For the T1 encoding:

- If `t == t2`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

For the A1 encoding:

- If `P=='0' && W=='1'` then one of the following behaviors must occurs:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction behaves as if the values of P and W were as follows:

    `P == '1' && W == '0'`

    `P == '1' && W == '1'`

    `P == '0' && W == '0'`

- If `Rt<0>=='1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The behavior is the same as if `Rt<0> =='0'`.
  - The register accessed are both specified by Rt. This means that Rt and Rt+1 are the same, and both have bit[0] == 1.
  - The registers accessed are specified by Rt and Rt+1.

    **Note**

    This does not apply if `Rt=='1111'`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## LDRD (literal)

For a description of this instruction and the encoding, see *LDRD (literal)* on page F6-2744.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If bit[24] and bit[21] do not have their *should be* values, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction is treated as if the bits had their *should be* values.
- If `Rt<0>=='1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The behavior is the same as if `Rt<0> =='0'`.
  - The registers accessed are both specified by Rt. This means that Rt and Rt+1 are the same, and both have bit[0] equal to 1.
  - The registers accessed are specified by Rt and Rt+1.

    **Note**

    This does not apply if `Rt=='1111'`.

For the T1 encoding:

- If `t == t2`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction performs all of the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.
- If `W == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction is treated as if `W == '0'`.
  - The instruction uses the P and W bits as described for *LDRD (immediate)* on page K1-5443.

## LDRD (register)

For a description of this instruction and the encoding, see *LDRD (register)* on page F6-2746.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If `P == '0' && W == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction behaves as if the values of P and W were as follows:

    `P == '1' && W == '0'`

    `P == '1' && W == '1'`

    `P == '0' && W == '0'`
- If `m == t || m ==t2`, then one of the following behaviors must occur;
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP`.
  - The instruction loads register Rm with an UNKNOWN value.
- If `Rt<0>=='1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The behavior is the same as if `Rt<0> == '0'`.
  - The registers accessed are both specified by Rt. This means that Rt and Rt+1 are the same, and in both cases bit[0] == 1.
  - The registers accessed are specified by Rt and Rt+1.

    **Note**

    This does not apply if `Rt=='1111'`.

## LDREX

For a description of this instruction and the encoding, see *LDREX* on page F6-2748.

### *CONSTRAINED UNPREDICTABLE behavior*

If the Load-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.

## LDREXB

For a description of this instruction and the encoding, see *LDREXB* on page F6-2750.

### *CONSTRAINED UNPREDICTABLE behavior*

If the Load-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## LDREXD

For a description of this instruction and the encoding, see *LDREXD* on page F6-2752.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If the Load-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.

For the T1 encoding:

- If `t == t2`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

For the A1 encoding:

- If `Rt<0>=='1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The behavior is the same as if the `Rt<0>=='0'`.
  - The registers accessed are both specified by Rt. This means that Rt and Rt+1 are the same, and both have bit[0] == 1.
  - The registers accessed are specified by Rt and Rt+1.

**Note**

If `t2 == 15`, then the requirements described in *Using R15* on page K1-5423 apply.

## LDREXH

For a description of this instruction and the encoding, see *LDREXH* on page F6-2754.

### *CONSTRAINED UNPREDICTABLE behavior*

If the Load-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.

## LDRH (immediate, A32)

For a description of this instruction and the encoding, see *LDRH (immediate)* on page F6-2756.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## LDRH (immediate, T32)

For a description of this instruction and the encoding, see *LDRH (immediate)* on page F6-2756.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T3 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

## LDRH (literal)

For a description of this instruction and the encoding, see *LDRH (literal)* on page F6-2759.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If bit[24] has the same value as bit[21]:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP`.
  - The instruction is treated as if bit[24] had value of 1 and bit[21] had a value of 0.
  - The instruction treats bit[24] as the P bit, and bit[21] as the writeback bit, and uses the same addressing mode as described in the immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    This is an exception to the principle in *SBZ or SBO fields in instructions* on page K1-5426.

- If bit[24] == 0 and bit[21] == 1, then the instruction can be handled according to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426 or the requirements described in *LDRH (immediate, T32)*.

## LDRH (register)

For a description of this instruction and the encoding, see *LDRH (register)* on page F6-2761.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### LDRHT

For a description of this instruction and the encoding, see *LDRHT* on page F6-2763.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 and A2 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

When the Rn field specifies R15, then this instruction can be treated either as described in this section or as described in *LDRH (literal)* on page K1-5447.

### LDRSB (immediate)

For a description of this instruction and the encoding, see *LDRSB (immediate)* on page F6-2765.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 and T2 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

### LDRSB (literal)

For a description of this instruction and the encoding, see *LDRSB (literal)* on page F6-2768.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If bit[24] has the same value as bit[21]:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP`.
  - The instruction is treated as if bit[24] had value of 1 and bit[21] had a value of 0.
  - The instruction treats bit[24] as the P bit, and bit[21] as the writeback bit, and uses the same addressing mode as described in the immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    This is an exception to the principle in *SBZ or SBO fields in instructions* on page K1-5426.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If bit[24] == 0 and bit[21] == 1, then the instruction can be handled according to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426 or the requirements described in *LDRSB (immediate)* on page K1-5448.

### LDRSB (register)

For a description of this instruction and the encoding, see *LDRSB (register)* on page F6-2770.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `wback && (n == t)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

### LDRSBT

For a description of this instruction and the encoding, see *LDRSBT* on page F6-2772.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 and A2 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

When the Rn field specifies R15, then this instruction can be treated either as described in this section or as described in *LDRSB (literal)* on page K1-5448.

### LDRSH (immediate)

For a description of this instruction and the encoding, see *LDRSH (immediate)* on page F6-2774.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T2 and A1 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## LDRSH (literal)

For a description of this instruction and the encoding, see *LDRSH (literal)* on page F6-2777.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If bit[24] has the same value as bit[21]:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP`.
  - The instruction is treated as if bit[24] had value of 1 and bit[21] had a value of 0.
  - The instruction treats bit[24] as the P bit, and bit[21] as the writeback bit, and uses the same addressing mode as described in the immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    This is an exception to the principle in *SBZ or SBO fields in instructions* on page K1-5426.

- If bit[24] == 0 and bit[21] == 1, then the instruction can be handled according to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426 or the requirements described in *LDRSH (immediate)* on page K1-5449.

## LDRSH (register)

For a description of this instruction and the encoding, see *LDRSH (register)* on page F6-2779.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

## LDRSHT

For a description of this instruction and the encoding, see *LDRSHT* on page F6-2781.

For the A1 and A2 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

When the Rn field specifies R15, then this instruction can be treated either as described in this section or as described in *LDRSB (literal)* on page K1-5448.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### LDRT

For a description of this instruction and the encoding, see *LDRT* on page F6-2783.

For the A1 and A2 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base register might be corrupted so that the instruction cannot be repeated.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

When the Rn field specifies R15, then this instruction can be treated either as described in this section or as described in *LDRSB (literal)* on page K1-5448.

### MOV (register, T32)

For a description of this instruction and the encoding, see *MOV, MOVS (register)* on page F6-2813.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T2 encoding:

- If `InITBlock()`, then one of the following behaviors must occur:
  - The instruction generates an UNDEFINED exception.
  - The instruction is executed as if it passed its condition code check.
  - The instruction executes as a `NOP`. That is, it behaves as if it failed its condition code check.
  - The instruction is treated as MOV Rd, Rm.

**Note**

This is an exception to the general behavior described in *CONSTRAINED UNPREDICTABLE instructions in an IT block* on page K1-5425.

### MRRC, MRRC2

For a description of this instruction and the encoding, see *MRRC, MRRC2* on page F6-2828.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `t == t2`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The register that is transferred takes an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### MSR (register)

For a description of this instruction and the encoding, see *MSR (register)* on page F6-2840.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `mask == '00'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.

### POP (A32)

For a description of this instruction and the encoding, see *POP* on page F6-2877.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If bit[13] is set to 1, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode but R13 is UNKNOWN.

For the A2 encoding:

- If `t == 13`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the load using the specified addressing mode but R13 is UNKNOWN.

  **Note**

  This is an exception to the requirements described in *Using R13* on page K1-5423.

### POP (T32)

For a description of this instruction and the encoding, see *POP* on page F6-2877.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T1 encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as a `POP` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the T2 encoding:

- If `BitCount(registers) < 2`, one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction loads a single register using the specified addressing modes.
  - The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These might include R15.
- If hw2 bit[13] is set to 1, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the loads using the specified addressing mode but R13 is UNKNOWN.

    **Note**

    This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.
- If `P == '1' && M == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

For the T3 encoding:

- If `t == 13`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the load using the specified addressing mode but R15 is UNKNOWN.

  **Note**

  This is an exception to the requirements described in *Using R13* on page K1-5423.

## PUSH

For a description of this instruction and the encoding, see *PUSH* on page F6-2883.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T1 encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as a `PUSH` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the T2 encoding:

- If `BitCount(registers) < 2`, one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction stores a single register using the specified addressing modes.
  - The instruction operates as a `PUSH` with the same addressing mode but targeting an unspecified set of registers. These might include R15.
- If hw2 bit[13] is set, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the stores using the specified addressing mode but R13 is UNKNOWN.

    **Note**

    This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.
- If hw2 bit[15] is set to 1, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the stores using the specified addressing mode, but R15 is UNKNOWN.

    **Note**

    This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.

## RBIT

For a description of this instruction and the encoding, see *RBIT* on page F6-2907.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T1 encoding:

- If `! Consistent(Rm)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The register specified by Rm in halfword 1 is used as the source register.
  - The register specified by Rm in halfword 2] is used as the source register.
  - The value in the destination register is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### REV

For a description of this instruction and the encoding, see *REV* on page F6-2909.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T2 encoding:

- If ! `Consistent(Rm)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The register specified by Rm in halfword 1 is used as the source register.
  - The register specified by Rm in halfword 2 is used as the source register.
  - The value in the destination register is UNKNOWN.

### REV16

For a description of this instruction and the encoding, see *REV16* on page F6-2911.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T2 encoding:

- If ! `Consistent(Rm)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The register specified by Rm in halfword 1 is used as the source register.
  - The register specified by Rm in halfword 2 is used as the source register.
  - The value in the destination register is UNKNOWN.

### REVSH

For a description of this instruction and the encoding, see *REVSH* on page F6-2913.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T2 encoding:

- If ! `Consistent(Rm)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The register specified by Rm in halfword 1 is used as the source register.
  - The register specified by Rm in halfword 2 is used as the source register.
  - The value in the destination register is UNKNOWN.

### SBFX

For a description of this instruction and the encoding, see *SBFX* on page F6-2956.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `msbit > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The output value in the register is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### SDIV

For a description of this instruction and the encoding, see *SDIV* on page F6-2958.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T1encoding:

- If hw2 bits[15:12] !='1111', then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction performs a divide with no side-effects on other registers.
  - The instruction performs a divide and the register specified by bits[15:12] becomes UNKNOWN.

  **Note**

  This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.

For the A1 encoding:

- If bits[15:12] !='1111', then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction performs a divide with no side-effects on other registers.
  - The instruction performs a divide and the register specified by bits[15:12] becomes UNKNOWN.

  **Note**

  This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.

### SMLAL

For a description of this instruction and the encoding, see *SMLAL, SMLALS* on page F6-2985.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If dHi == dLo, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The destination register takes an UNKNOWN value.

### SMLALBB, SMLALBT, SMLALTB, SMLALTT

For a description of this instruction and the encoding, see *SMLALBB, SMLALBT, SMLALTB, SMLALTT* on page F6-2987.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If dHi == dLo, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The destination register takes an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### SMLALD

For a description of this instruction and the encoding, see *SMLAD, SMLADX* on page F6-2983.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `dHi == dLo`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The destination register takes an UNKNOWN value.

### SMLSLD

For a description of this instruction and the encoding, see *SMLSD, SMLSDX* on page F6-2993.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `dHi == dLo`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The destination register takes an UNKNOWN value.

### SMULL

For a description of this instruction and the encoding, see *SMULL, SMULLS* on page F6-3007.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `dHi == dLo`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The destination register takes an UNKNOWN value.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### STC, STC2

For a description of this instruction and the encoding, see *STC, STC2* on page F6-3026.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `n == 15 && (wback || CurrentInstrSet() != InstrSet_A32)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction set uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

**Note**

Use of R15 by the T32 `STC` instruction is also covered by *Using R15* on page K1-5423.

### STLEX

For a description of this instruction and the encoding, see *STLEX* on page F6-3032.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction executes, but the value stored is UNKNOWN.
- If `d == n`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store to an UNKNOWN address.
- If the Store-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.
- If the instruction specifies that `t2==15`, then the requirements described in *Using R15* on page K1-5423 apply.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## STLEXB

For a description of this instruction and the encoding, see *STLEXB* on page F6-3034.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction executes, but the value stored is UNKNOWN.
- If `d == n`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store to an UNKNOWN address.
- If the Store-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.
- If the instruction specifies that `t2==15`, then the requirements described in *Using R15* on page K1-5423 apply.

## STLEXD

For a description of this instruction and the encoding, see *STLEXD* on page F6-3036.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction executes, but the value stored is UNKNOWN.
- If `d == n`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store to an UNKNOWN address.
- If the Store-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.
- If the instruction specifies that `t2==15`, then the requirements described in *Using R15* on page K1-5423 apply.

For the A1 encoding:

- If `Rt<0>=='1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The behavior is the same as if `Rt<0>=='0'`.
  - The register accessed are both specified by Rt. For example, Rt and Rt+1 are the same, and both have bit[0] == 1.
  - The registers accessed are specified by Rt and R+1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## STLEXH

For a description of this instruction and the encoding, see *STLEXH* on page F6-3038.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction executes, but the value stored is UNKNOWN.
- If `d == n`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store to an UNKNOWN address.
- If the Store-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.
- If the instruction specifies that `t2==15`, then the requirements described in *Using R15* on page K1-5423 apply.

## STM (STMIA, STMEA)

For a description of this instruction and the encoding, see *STM, STMIA, STMEA* on page F6-3041.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

For the T1 and A1 encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

For the T2 encoding:

- If `BitCount(registers) < 2`, then one of the following behaviors occurs:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction stores a single register using the specified addressing modes.
  - The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These might include R15.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If hw2 bit[13] is set, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction performs all the stores using the specified addressing mode but the value of R13 is UNKNOWN.

  **Note**

  This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.

- If hw2 bit[15] is set to 1, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

  **Note**

  This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.

- If `wback && registers<n> == '1'`, and that case is defined to be UNPREDICTABLE, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction performs all of the stores using the specified addressing mode and the value stored for the base register is UNKNOWN.

## STMDB (STMFD)

For a description of this instruction and the encoding, see *STMDB, STMFD* on page F6-3047.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

For the A1 encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The store instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

For the T1 encoding:

- If `BitCount(register) < 2`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction stores a single register using the specified addressing mode.
  - The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These might include R15.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If `wback && registers<n> == '1'`, and that case is defined to be UNPREDICTABLE, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the stores using the specified addressing mode and the value stored for the base register is UNKNOWN.
- If hw2 bit[13] is set to 1, then one of the following behaviors occurs:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

    **Note**

    This an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.
- If hw2 bit[15] is set to 1, then one of the following behaviors occurs:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

    **Note**

    This is an exception to the requirement described in *SBZ or SBO fields in instructions* on page K1-5426.

## STMDA (STMED)

For a description of this instruction and the encoding, see *STMDA, STMED* on page F6-3045.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.
- If the instruction uses R15 as a base register and specifies writeback, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction is performed without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.
  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

### STMIB (STMFA)

For a description of this instruction and the encoding, see *STMIB, STMFA* on page F6-3049.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.
- If the instruction uses R15 as a base register and specifies writeback, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction is performed without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.
- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

### STR (immediate, A32)

For a description of this instruction and the encoding, see *STR (immediate)* on page F6-3051.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1encoding:

- If `t == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

### STR (immediate, T32)

For a description of this instruction and the encoding, see *STR (register)* on page F6-3055.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T3 encoding:

- If `t == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

For the T4 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

## STR (register)

For a description of this instruction and the encoding, see *STR (register)* on page F6-3055.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T2 encoding:

- If `t == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

For the A1encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    Pre-indexed addressing and post-indexed addressing imply writeback.

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### STRB (immediate, A32)

For a description of this instruction and the encoding, see *STRB (immediate)* on page F6-3058.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `t == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    Pre-indexed addressing and post-indexed addressing imply writeback.
- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

### STRB (immediate), T32

For a description of this instruction and the encoding, see *STRB (immediate)* on page F6-3058.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the T3 encoding:

- If `t == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

## STRB (register)

For a description of this instruction and the encoding, see *STRB (register)* on page F6-3061.

### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 and T2 encoding:

- If `t == 15` is UNPREDICTABLE, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For the A1 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## STRBT

For a description of this instruction and the encoding, see *STRBT* on page F6-3063.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `t == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

For the A1 and A2 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.
  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

## STRD (immediate)

For a description of this instruction and the encoding, see *STRD (immediate)* on page F6-3065.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `t == 15` or `t2 == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

- If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store of the registers specified using the specified addressing mode but the value of the registers stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

  **Note**

  Pre-indexed addressing and post-indexed addressing imply writeback.

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

For the A1 encoding:

- If `Rt<0>=='1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The behavior is the same as if `Rt<0>=='0'`.
  - The registers accessed are both specified by Rt. For example, Rt and Rt+1 are the same, and both have bit[0] equal to 1.
  - The registers accessed are specified by Rt and Rt+1.

    **Note**

    This does not apply if `Rt=='1111'`.

- If `P == '0' && W == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP`.
  - The instruction behaves as if the values of P and W were one of:

    ```
    P == '1' && W == '0'
    P == '1' && W == '1'
    P == '0' && W == '0'
    ```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### STRD (register)

For a description of this instruction and the encoding, see *STRD (register)* on page F6-3067.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `t2 == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store of the registers specified using the specified addressing mode but the value of the registers stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

  **Note**

  Pre-indexed addressing and post-indexed addressing imply writeback.

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.
- If `Rt<0>=='1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The behavior is the same as if `Rt<0>=='0'`.
  - The registers accessed are both specified by Rt. For example, Rt and Rt+1 are the same, and both have bit[0] equal to 1.
  - The registers accessed are specified by Rt and Rt+1.

    **Note**

    This does not apply if `Rt=='1111'`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If `P == '0' && W == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP`.
  - The instruction behaves as if the values of P and W were one of:
    `P == '1' && W == '0'`
    `P == '1' && W == '1'`
    `P == '0' && W == '0'`

## STREX

For a description of this instruction and the encoding, see *STREX* on page F6-3069.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction executes, but the value stored is UNKNOWN.
- If `d == n`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store to an UNKNOWN address.
- If the Store-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.
- If the instruction specifies that `t2==15`, then the requirements described in *Using R15* on page K1-5423 apply.

## STREXB

For a description of this instruction and the encoding, see *STREXB* on page F6-3071.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction executes, but the value stored is UNKNOWN.
- If `d == n`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store to an UNKNOWN address.
- If the Store-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.
- If the instruction specifies that `t2==15`, then the requirements described in *Using R15* on page K1-5423 apply.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## STREXD

For a description of this instruction and the encoding, see *STREXD* on page F6-3073.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction executes, but the value stored is UNKNOWN.
- If `d == n`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store to an UNKNOWN address.
- If the Store-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.
- If the instruction specifies that `t2==15`, then the requirements described in *Using R15* on page K1-5423 apply.

For the A1 encoding:

- If `Rt<0>=='1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The behavior is the same as if `Rt<0>=='0'`.
  - The register accessed are both specified by Rt. For example, Rt and Rt+1 are the same, and both have bit[0] == 1.
  - The registers accessed are specified by Rt and R+1.

## STREXH

For a description of this instruction and the encoding, see *STREXH* on page F6-3075.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction executes, but the value stored is UNKNOWN.
- If `d == n`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store to an UNKNOWN address.
- If the Store-Exclusive is to any type of Device memory, then the instruction operates as if the access were to Normal memory.
- If the instruction specifies that `t2==15`, then the requirements described in *Using R15* on page K1-5423 apply.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### STRH (immediate, A32)

For a description of this instruction and the encoding, see *STRH (immediate)* on page F6-3077.

#### *CONSTRAINED UNPREDICTABLE behavior*

For the A1 encoding:

- If `t == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    Pre-indexed addressing and post-indexed addressing imply writeback.

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

### STRH (immediate, T32)

For a description of this instruction and the encoding, see *STRH (immediate)* on page F6-3077.

#### *CONSTRAINED UNPREDICTABLE behavior*

For T2 and T3 encodings:

- If `t == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

For the T3 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

## STRH (register)

For a description of this instruction and the encoding, see *STRH (register)* on page F6-3080.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T2 and A1 encoding:

- If `t == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

For the A1 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    Pre-indexed addressing and post-indexed addressing imply writeback.

### STRHT

For a description of this instruction and the encoding, see *STRHT* on page F6-3082.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `t == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

For the A1 and A2 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    Pre-indexed addressing and post-indexed addressing imply writeback.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## STRT

For a description of this instruction and the encoding, see *STRT* on page F6-3084.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T1 encoding:

- If `t == 15` is UNPREDICTABLE, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For the A1 and A2 encoding:

- If `wback && n == t`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs the store using the specified addressing mode but the value stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The store instruction operates without writeback.

    **Note**

    This is consistent with ignoring writes to the PC.

  - The instruction uses the addressing mode described in the equivalent immediate offset instruction. If this results in writeback to the PC, then the requirements described in *Using R15* on page K1-5423 apply.

    **Note**

    Pre-indexed addressing and post-indexed addressing imply writeback.

- The value used by stores using R15 as a base register follow the requirements described in *Using R15* on page K1-5423.

## UBFX

For a description of this instruction and the encoding, see *UBFX* on page F6-3134.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `msbit > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The output value in the register is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## UDIV

For a description of this instruction and the encoding, see *UDIV* on page F6-3138.

### *CONSTRAINED UNPREDICTABLE behavior*

For the T1encoding:

- If hw2 bits[15:12] !='1111', then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction performs a divide with no side-effects on other registers.
  - The instruction performs a divide and the register specified by bits[15:12] becomes UNKNOWN.

  **Note**

  This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.

For the A1 encoding:

- If bits[15:12] !='1111', then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction performs a divide with no side-effects on other registers.
  - The instruction performs a divide and the register specified by bits[15:12] becomes UNKNOWN.

  **Note**

  This is an exception to the requirements described in *SBZ or SBO fields in instructions* on page K1-5426.

## UMAAL

For a description of this instruction and the encoding, see *UMAAL* on page F6-3152.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If dHi == dLo, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The destination register takes an UNKNOWN value.

## UMLAL

For a description of this instruction and the encoding, see *UMLAL, UMLALS* on page F6-3154.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If dHi == dLo, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The destination register takes an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### UMULL

For a description of this instruction and the encoding, see *UMULL, UMULLS* on page F6-3156.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `dHi == dLo`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The destination register takes an UNKNOWN value.

## K1.1.27 CONSTRAINED UNPREDICTABLE behavior for A32 and T32 system instructions in the base instruction set

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A32 and T32 system instructions.

**Note**

If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

### CPS (A32)

For a description of this instruction and the encoding, see *CPS, CPSID, CPSIE* on page F6-2662.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes an illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

  **Note**

  - An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.
  - `CPS` executed from User mode acts as a `NOP`.

For the A1 encoding:

- If `(imod == '00 && M =='0') || imod == '01'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
- If `mode != '00000' && M == '0'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as if `M == '1'`.
  - The instruction operates as if `mode == '0000'`.
- If `(imod<1> == '1' && A:I:F == '000')`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction behaves as if `imod<1> == '0'`.
  - The instruction behaves as if `A:I:F` had an UNKNOWN nonzero value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If (imod<1> == '0' && A:I:F != 000) then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction behaves as if imod<1> == '1'.
  - The instruction behaves as if A:I:F == 000.

## CPS (T32)

For a description of this instruction and the encoding, see *CPS, CPSID, CPSIE* on page F6-2662.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

  **Note**

  - An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.
  - CPS executed from User mode acts as a NOP.

For the T1 encoding:

- If A:I:F == '000', then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.

For the T2 encoding:

- If imod == '01', then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
- If mode != '00000' && M == '0', then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction operates as if M =='1'.
  - The instruction behaves as if mode == '0000'.
- If (imod<1> == '1' && A:I:F == '000'), then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction behaves as if imod<1> == '0'.
  - The instruction behaves as if A:I:F had an UNKNOWN nonzero value.
- If (imod<1> == '0' && A:I:F != '000') then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The instruction behaves as if imod<1> == '1'.
  - The instruction behaves as if A:I:F == '000'.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## LDM (exception return)

For a description of this instruction and the encoding, see *LDM (exception return)* on page F6-2714.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
- If `wback && registers<n> == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction performs all the loads using the specified addressing mode and the content of the register being written back is UNKNOWN. In addition, if an exception occurs during the execution of this instruction, the base address might be corrupted so that the instruction cannot be repeated.
- If the instruction transfers an illegal mode encoding PSTATE.M, then this invokes the illegal exception return.

  **Note**

  An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

## LDM (User registers)

For a description of this instruction and the encoding, see *LDM (User registers)* on page F6-2716.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.
- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## MRS

For a description of this instruction and the encoding, see *MRS* on page F6-2830.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If this instruction is executed in User mode or in System mode and is accessing the SPSR, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.

## MSR (immediate)

For a description of this instruction and the encoding, see *MSR (immediate)* on page F6-2838.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If the instruction is executed in User mode or in System mode and is accessing the SPSR, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
- If `mask == '0000' && R == '1'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.

  If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

  **Note**

  An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

## MSR (register)

For a description of this instruction and the encoding, see *MSR (register)* on page F6-2840.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If this instruction is executed in User mode or in System mode and is accessing the SPSR, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If `mask == '0000'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.

  If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

  **Note**

  An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

### RFE

For a description of this instruction and the encoding, see *RFE, RFEDA, RFEDB, RFEIA, RFEIB* on page F6-2915.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If this instruction is executed in User mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

  **Note**

  An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

### SRS (T32)

For a description of this instruction and the encoding, see *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F6-3013.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
- If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - R13 of the current mode is used.
  - The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## SRS (A32)

For a description of this instruction and the encoding, see *SRS, SRSDA, SRSDB, SRSIA, SRSIB* on page F6-3013.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
- If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - R13 of the current mode is used.
  - The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

## STM (User registers)

For a description of this instruction and the encoding, see *STM (User registers)* on page F6-3043.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.
- If the instruction is executed from User mode or System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### SUBS PC, LR and related instructions (T32)

For a description of this instruction and the encoding, see the exception return form of *SUB, SUBS (immediate)* on page F6-3088.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

  **Note**

  An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

For this encoding:

- If hw2[3:0] are `0b1110`, and the instruction is executed when not in Hyp mode, System mode, or User mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction is treated as a `NOP`.
  - The instruction is treated as if hw2[3:0] are `0b1110`.
  - The program counter is set using the value in the register specified by hw2[3:0].

### SUBS PC. LR and related instructions (A32)

For a description of this instruction and the encoding, see the exception return forms of *MOV, MOVS (register)* on page F6-2813 and *SUB, SUBS (immediate)* on page F6-3088.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

  **Note**

  An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## K1.1.28 CONSTRAINED UNPREDICTABLE behavior, A32 and T32 Advanced SIMD and floating-point instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A32 and T32 Advanced SIMD and floating-point instructions listed in *Alphabetical list of floating-point and Advanced SIMD instructions* on page F7-3208.

**Note**

- The pseudocode used in this section to describe cases that can result in CONSTRAINED UNPREDICTABLE behavior does not necessarily match the encoding specific pseudocode for a specific instruction.
- If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

### VCVT (between floating-point and fixed-point)

For a description of this instruction and the encoding, see *VCVT (between floating-point and fixed-point, floating-point)* on page F7-3340.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `frac_bits < 0`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The result of the conversion is UNKNOWN.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

### VLD1 (multiple single elements)

For a description of this instruction and the encoding, see *VLD1 (multiple single elements)* on page F7-3400.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d+regs > 32`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### VLD1 (single element to all lanes)

For a description of this instruction and the encoding, see *VLD1 (single element to all lanes)* on page F7-3397.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d+regs > 32`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

### VLD2 (multiple 2-element structures)

For a description of this instruction and the encoding, see *VLD2 (multiple 2-element structures)* on page F7-3409.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d2+regs > 32`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

### VLD2 (single 2-element structure to one lane)

For a description of this instruction and the encoding, see *VLD2 (single 2-element structure to one lane)* on page F7-3403.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d2 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### VLD2 (single 2-element structure to all lanes)

For a description of this instruction and the encoding, see *VLD2 (single 2-element structure to all lanes)* on page F7-3406.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d2 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

### VLD3 (multiple 3-element structures)

For a description of this instruction and the encoding, see *VLD3 (multiple 3-element structures)* on page F7-3418.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d3 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

### VLD3 (single 3-element structure to one lane)

For a description of this instruction and the encoding, see *VLD3 (single 3-element structure to one lane)* on page F7-3412.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d3 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### VLD3 (single 3-element structure to all lanes)

For a description of this instruction and the encoding, see *VLD3 (single 3-element structure to all lanes)* on page F7-3415.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d+regs > 32`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

### VLD4 (multiple 4-element structures)

For a description of this instruction and the encoding, see *VLD4 (multiple 4-element structures)* on page F7-3427.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

### VLD4 (single 4-element structure to one lane)

For a description of this instruction and the encoding, see *VLD4 (single 4-element structure to one lane)* on page F7-3421.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d4 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## VLD4 (single 4-element structure to all lanes)

For a description of this instruction and the encoding, see *VLD4 (single 4-element structure to all lanes)* on page F7-3424.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d4 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

## VLDM

For a description of this instruction and the encoding, see *VLDM, VLDMDB, VLDMIA* on page F7-3430.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `regs == 0`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as a `VLDM` or a `VPOP` with the same addressing mode but loads no registers.
- If the register list includes a register that is out of range, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD&FP registers become UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**Note**

Out of range means that the register list includes:

- A register with a number greater than the number of registers implemented.
- More than 16 double word registers.
- A double word register specified as D16-D31 with an odd value of `imm8`.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### VMOV (between two general-purpose registers and two single-precision registers)

For a description of this instruction and the encoding, see *VMOV (between two general-purpose registers and two single-precision registers)* on page F7-3489.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `to_arm_registers && t == t2`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The destination register becomes UNKNOWN.

  If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.
- If `m == 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD&FP single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

### VMOV (between two general-purpose registers and a doubleword floating-point register)

For a description of this instruction and the encoding, see *VMOV (between two general-purpose registers and a doubleword floating-point register)* on page F7-3474.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `to_arm_registers && t == t2`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The destination register becomes UNKNOWN.

  If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## VMRS

For a description of this instruction and the encoding, see *VMRS* on page F7-3495.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `t == 15 && reg !='0001'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction transfers an UNKNOWN value to the specified general-purpose register.

In addition, `CheckVFPEnabled(FALSE)` can be called by this instruction for the CONSTRAINED UNPREDICTABLE cases.

## VMSR

For a description of this instruction and the encoding, see *VMSR* on page F7-3497.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `reg!= '000x' && reg!='1000'`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using `VMSR` at the same Exception level.

In addition, `CheckVFPEnabled(FALSE)` can be called by this instruction for the CONSTRAINED UNPREDICTABLE cases.

## VPOP

For a description of this instruction and the encoding, see *VPOP* on page F7-3549.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `regs == 0`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as a `VLDM` or a `VPOP` with the same addressing mode but loads no registers.
- If the register list includes a register that is out of range, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - One or more of the SIMD&FP registers become UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**Note**

Out of range means that the register list includes:

- A register with a number greater than the number of registers implemented.
- More than 16 double word registers.
- A double word register specified as D16-D31 with an odd value of `imm8`.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

## VPUSH

For a description of this instruction and the encoding, see *VPUSH* on page F7-3551.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `regs == 0`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as a `VSTM` or `VPUSH` with the same addressing mode but loads no registers.
- If the register list includes a register that is out of range, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**Note**

Out of range means that the register list includes:

- A register with a number greater than the number of registers implemented.
- More than 16 double word registers.
- A double word register specified as D16-D31 with an odd value of `imm8`.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

## VST1 (multiple single elements)

For a description of this instruction and the encoding, see *VST1 (multiple single elements)* on page F7-3690.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d+regs > 32`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## VST2 (multiple 2-element structures)

For a description of this instruction and the encoding, see *VST2 (multiple 2-element structures)* on page F7-3696.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d2+regs > 32`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

## VST2 (single 2-element structure from one lane)

For a description of this instruction and the encoding, see *VST2 (single 2-element structure from one lane)* on page F7-3693.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d2 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## VST3 (multiple 3-element structures)

For a description of this instruction and the encoding, see *VST3 (multiple 3-element structures)* on page F7-3702.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d3 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

## VST3 (single 3-element structure from one lane)

For a description of this instruction and the encoding, see *VST3 (single 3-element structure from one lane)* on page F7-3699.

### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d3 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### VST4 (multiple 4-element structures)

For a description of this instruction and the encoding, see *VST4 (multiple 4-element structures)* on page F7-3708.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d4 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

### VST4 (single 4-element structure from one lane)

For a description of this instruction and the encoding, see *VST4 (single 4-element structure from one lane)* on page F7-3705.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `d4 > 31`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

### VSTM

For a description of this instruction and the encoding, see *VSTM, VSTMDB, VSTMIA* on page F7-3711.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If `regs == 0`, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a `NOP`.
  - The instruction operates as a `VSTM` or `VPUSH` with the same addressing mode but loads no registers.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- If the register list includes a register that is out of range, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**Note**

Out of range means that the register list includes:

- A register with a number greater than the number of registers implemented.
- More than 16 double word registers.
- A double word register specified as D16-D31 with an odd value of imm8.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

### VTBL, VTBX

For a description of this instruction and the encoding, see *VTBL, VTBX* on page F7-3729.

#### *CONSTRAINED UNPREDICTABLE behavior*

For all encodings:

- If n + length > 32, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - One or more of the SIMD&FP registers become UNKNOWN. This behavior does not affect any general-purpose registers.

## K1.1.29 CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR

The following subsections describe the CONSTRAINED UNPREDICTABLE behavior associated with programming the VTCR:

- *VTCR.S*.
- *CONSTRAINED UNPREDICTABLE combinations of the starting level and size fields*.

### VTCR.S

VTCR.S must be programmed to T0SZ[3], or the effect is CONSTRAINED UNPREDICTABLE. For the ARMv8-A architecture, if VTCR.S is not programmed correctly, then the stage 2 T0SZ value is treated as an UNKNOWN value.

### CONSTRAINED UNPREDICTABLE combinations of the starting level and size fields

If the stage 2 input address size, as programmed in VTCR.T0SZ, is out of range with respect to the starting level, as programmed in the VTCR.SL0 field at the time of a translation walk that uses the stage 2 translation, then a stage 2 level 1 Translation Fault is generated.

## K1.1.30 CONSTRAINED UNPREDICTABLE behavior in Debug state

*Behavior in Debug state* on page H2-4832 of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## K1.1.31 CONSTRAINED UNPREDICTABLE behavior of EL2 features

The following sections describe CONSTRAINED UNPREDICTABLE behavior that can occur in an implementation that includes EL2 where EL2 can use AArch32:

- *ERET in User mode or System mode*.
- *Accessing Hyp mode from outside Hyp mode*.
- *Modifying PSTATE.M when in Hyp mode*
- *Use of Hyp mode in Secure state*.
- *Instructions which are UNDEFINED or CONSTRAINED UNPREDICTABLE in Hyp mode* on page K1-5498.
- *Exception return to Hyp mode* on page K1-5498.
- *Accessing registers that cannot be accessed using MSR/MRS instructions* on page K1-5498.
- *Memory type handling* on page K1-5498.
- *Hyp mode TLB maintenance instructions* on page K1-5499.
- *Hyp mode VA to PA address translation instructions* on page K1-5499.
- *Stage 1 default memory type* on page K1-5499.
- *Trapping of general exceptions to Hyp mode* on page K1-5499.
- *Prevention of rootskits using Hyp mode or Secure state* on page K1-5500.
- *HVC* on page K1-5500.
- *MSR/MRS banked registers* on page K1-5500.

### ERET in User mode or System mode

If ERET is executed in User mode or System mode, it behaves as described in *SUBS PC, LR and related instructions (T32)* on page K1-5484.

### Accessing Hyp mode from outside Hyp mode

Attempting to change into Hyp mode or out of Hyp mode using the `MSR` or `CPS` instruction invokes the ARMv8 illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

`SRS` using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in *SRS (T32)* on page K1-5482 and *SRS (A32)* on page K1-5483.

### Modifying PSTATE.M when in Hyp mode

Attempting to change into Hyp mode or out of Hyp mode using the `MSR` or `CPS` instruction invokes the ARMv8 illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

`SRS` using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in *SRS (T32)* on page K1-5482 and *SRS (A32)* on page K1-5483.

### Use of Hyp mode in Secure state

Attempting to change into Hyp mode or out of Hyp mode using the `MSR` or `CPS` instruction invokes the ARMv8 illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

`SRS` using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in *SRS (T32)* on page K1-5482 and *SRS (A32)* on page K1-5483.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Instructions which are UNDEFINED or CONSTRAINED UNPREDICTABLE in Hyp mode

If LDRT, LDRSHT, LDRHT, LDRSBT, LDRBT, STRT, STRHT or STRBT are executed in Hyp mode, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the equivalent, corresponding LDR, LDRSH, LDRH, LDRSB, LDRB, STR, STRH or STRB instruction in Hyp mode.

### Exception return to Hyp mode

Exception returns to Hyp mode when SCR.NS == 0 or from a Non-secure PL1 mode invokes the ARMv8 illegal exception return.

### Accessing registers that cannot be accessed using MSR/MRS instructions

The following MSR and MRS instructions can lead to CONSTRAINED UNPREDICTABLE behavior:

```
MSR <Rm>_<mode>, <Rn>

MSR SPSR_<mode>, <Rn>

MSR ELR_hyp, <Rn>

MRS <Rn>, <Rm>_<mode>

MRS <Rn>, SPSR_<mode>

MRS <Rn>, ELR_hyp
```

If these instructions are executed in either Secure or Non-secure User mode, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.

If the MSR and MRS instructions attempt to access a register that cannot be legally accessed, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- For MRS instructions, the destination general-purpose register becomes UNKNOWN.
- For MSR instructions, if the register specified could be accessed from the current mode by other mechanisms, then this register is UNKNOWN. Otherwise the instruction executes as a NOP.

### Memory type handling

If the attributes for a memory location after combining stage 1 and stage 2 of a translation regime is Normal Inner Non-cacheable, Outer Non-cacheable, then the shareability attributes after combining the two stages of translation is Outer Shareable.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Hyp mode TLB maintenance instructions

If a TLBIMVAH, TLBIMVALH, TLBIMVAHIS, TLBIMVALHIS, TLBIALLNSNH, TLBIALLNSNHIS, TLBIALLH, or a TLBIALLHIS instruction is executed in a Secure Privileged mode other than Monitor mode, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction is executed as if it had been executed in Monitor mode.

For more information about these instructions see *The scope of TLB maintenance instructions* on page G4-4069.

### Hyp mode VA to PA address translation instructions

If an ATS1HR or ATS1HW instruction is executed in a Secure Privileged mode other than Monitor mode, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction is executed as if it had been executed in Monitor mode.

For more information about these instructions see *Address translation instruction naming and operation summary* on page G4-4110.

### Stage 1 default memory type

If HCR.DC == 1, then the behavior of the PE when executing in a Non-secure mode other than Hyp mode is consistent with:

- SCTLR.M == 0, regardless of the actual value of SCTLR.M, other than for the value returned by an explicit read of SCTLR.M.
- HCR.VM == 1, regardless of the actual value of HCR.VM, other than for an explicit read of this bit.

### Trapping of general exceptions to Hyp mode

Attempting to perform an exception return to a Non-secure PL1 mode when HCR.TGE == 1 invokes an illegal exception return.

Attempting to change from Monitor mode to a Non-secure PL1 mode when HCR.TGE == 1 by executing a CPS or MSR instruction generates an Illegal Execution state exception, by not changing the mode, and setting PSTATE.IL to 1.

When EL3 is using AArch32, attempting to change from a Secure PL1 mode to a Non-secure PL1 mode when HCR.TGE is set, by changing SCR.NS from 0 to 1, results in no change of SCR.NS

Because taking an exception into Non-secure PL1 modes leads to a CONSTRAINED UNPREDICTABLE situation, the following additional properties apply when HCR.TGE == 1:

- All exceptions that would be routed to EL1 are routed to EL2.
- Non-secure SCTLR.M is treated as being 0, regardless of its actual value, other than for an explicit read of of this bit.
- HCR.FMO, HCR.IMO, and HCR.AMO are treated as being 1, regardless of their actual value, other than for an explicit read of these bits.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signalling virtual interrupts are disabled.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Prevention of rootskits using Hyp mode or Secure state

If an HVC instruction is executed in Hyp mode when SCR.HCE == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.

If an SMC instruction is executed in a Secure privileged mode when SCR.SCD == 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.

### HVC

For a description of this instruction and the encoding, see *HVC* on page F6-2686.

For the A1 encoding, if `cond field !=1110`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

### MSR/MRS banked registers

Some unallocated encodings for the `MSR/MRS` banked register instructions can cause CONSTRAINED UNPREDICTABLE behavior. If an unallocated encoding for the `MSR/MRS` banked register instructions is executed, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- An allocated MSR/MRS banked register instruction is executed.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K1.2 Constraints on AArch64 state UNPREDICTABLE behaviors

It contains the following sections:


- *Overview of the constraints on AArch64 UNPREDICTABLE behaviors*.
- *Reserved values in System and memory-mapped registers and translation table entries*.
- *CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values* on page K1-5502.
- *CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization* on page K1-5502.
- *Translation table base address alignment* on page K1-5502.
- *The Performance Monitors Extension* on page K1-5502.
- *Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED* on page K1-5503.
- *Out of range virtual address* on page K1-5503.
- *Instruction fetches from Device memory* on page K1-5504.
- *Programming the CSSELR_EL1.Level for a cache level that is not implemented* on page K1-5504.
- *Crossing a page boundary with different memory types or Shareability attributes* on page K1-5504.
- *Crossing a peripheral boundary with a Device access* on page K1-5505.
- *CONSTRAINED UNPREDICTABLE behavior in Debug state* on page K1-5505.
- *CONSTRAINED UNPREDICTABLE behavior for A64 instructions* on page K1-5505.
- *Out of range values of the Set/Way/Index fields in cache maintenance instructions* on page K1-5513.


### K1.2.1 Overview of the constraints on AArch64 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch64 state, the ARMv8-A architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

**Note**

Software designed to be compatible with the ARMv8-A architecture must not rely on these CONSTRAINED UNPREDICTABLE cases being handled in any way other than those listed under the heading CONSTRAINED UNPREDICTABLE.

### K1.2.2 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated in this manual, all unallocated or reserved values of fields with allocated values within AArch64 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The unallocated value maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The unallocated value causes effects that could be achieved by a combination of more than one of the allocated values.
- The unallocated value causes the field to have no functional effect.

**Note**

These constraints are identical to those for the equivalent AArch32 definitions, as given in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-5427.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K1.2.3 CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values

The ARM architecture allows copies of control values or data values to be cached in a cache or TLB. This can lead to UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with:

- The old data or control value.
- The new data or control value.
- An amalgamation of the control or data values.

**Note**

This rules applies where inadequate invalidation of the TLB might cause multiple hits within the TLB. In this situation, a failure to invalidate the TLB by code running at a given Privilege level must not make access to regions of memory with permissions or attributes that could not be accessed at that Privilege level possible.

Alternatively, an implementation might generate a Data Abort exception when detecting multiple hits within a TLB, using the TLB Conflict fault code.

The choice between these behaviors might, in some implementations, vary for each use of a control or data value.

### K1.2.4 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The ARM architecture requires that changes to System registers must be synchronized before they take effect. This can lead to UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the processor is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

### K1.2.5 Translation table base address alignment

Field [x-1:0] in TTBR0_EL1, TTBR1_EL1, TTBR0_EL2, VTTBR_EL2, or TTBR0_EL3 is RES0. If this field does not have a value of 0, this might result in a misaligned translation table base address. In this case, one of the following behaviors must occur:

- The field that is defined to be RES0 is treated as if all the bits had a value of 0:
  - The value read back might be the value written or it might be 0.
- The calculation of an address for a translation table walk using those registers might be corrupted in those bits that are nonzero.

### K1.2.6 The Performance Monitors Extension

The following cases can cause CONSTRAINED UNPREDICTABLE behavior:

- If PMSELR_EL0.SEL is not equal to 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in an Exception level in Secure state or in EL2.
- If PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to MDCR_EL2.HPMN, and the PE is executing in an Exception level in Non-secure state other than in EL2.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

In these cases, one of the following behaviors must occur:

- Any access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state is UNDEFINED.
- Any access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state executes as a NOP.
- Any access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state either:
  - Ignores writes and returns 0 on reads.
  - Behaves as if PMSELR_EL0.SEL contains an UNKNOWN value that is less than PMCR_EL0.N and the PE is executing in an Exception level in Secure state or in EL2, or behaves as if PMSELR_EL0.SEL contains an UNKNOWN value that is less than MDCR_EL2.HPMN if the PE executing in an Exception level in Non-secure state other than in EL2.

If PMSELR_EL0.SEL is equal to 31, then one of the following behaviors must occur:

- Any access to PMXEVCNTR_EL0 is UNDEFINED.
- Any access to PMXEVCNTR_EL0 executes as a NOP.
- Any access to PMXEVCNTR_EL0 either:
  - Ignores writes and returns 0 on reads.
  - Behaves as if PMSELR_EL0.SEL contains an UNKNOWN value that is less than PMCR_EL0.N and the PE is executing in an Exception level in Secure state or in EL2, or behaves as if PMSELR_EL0.SEL contains an UNKNOWN value that is less than MDCR_EL2.HPMN if the PE is executing in an Exception level in Non-secure state other than in EL2.

If MDCR_EL2.HPMN is set to 0, or to a value larger than PMCR_EL0.N, then the following CONSTRAINED UNPREDICTABLE behavior applies:

- The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if MDCR_EL2.HPMN is set to an UNKNOWN non-zero value less than PMCR_EL0.N.
  - All counters are reserved for EL2 use, meaning no counters are accessible from Non-secure EL1 and Non-secure EL0.

### K1.2.7 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, ESR_ELx is UNKNOWN.

**Note**

The value written to ESR_ELx must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but was the result of a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

### K1.2.8 Out of range virtual address

Because of program counter alignment constraints, it is impossible for a PE to fetch an A64 instruction that includes the bytes at virtual address `0xFFFF FFFF FFFF FFFF` and `0x0000 0000 0000 0000`.

If the PE executes a load or store instruction with tagged addressing disabled in the current translation regime, and where the computed virtual address, total access size, and alignment mean that it accesses the bytes at `0xFFFF FFFF FFFF FFFF` and `0x0000 0000 0000 0000`, then the bytes that appear to be from `0x0000 0000 0000 0000` onwards are accessed at an UNKNOWN address.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the PE executes a load or store instruction with tagged addressing enabled in the current translation regime, and where the computed address, total access size, and alignment mean that it accesses the bytes at `0xFFFF FFFF FFFF FFFF` and `0x0000 0000 0000 0000`, then the bytes that appear to be from `0x0000 0000 0000 0000` onwards are accessed at an UNKNOWN address and the tags associated with address also become UNKNOWN.

### K1.2.9 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location at times when address translation is enabled.

If a branch causes the program counter to point to an area of memory with the Device attribute that is not marked as execute-never for the current Exception level for instruction fetches, then an implementation must perform one of the following behaviors:

- It treats the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- It generates a Permission fault.

### K1.2.10 Programming the CSSELR_EL1.Level for a cache level that is not implemented

If the CSSELR_EL1.Level is programmed to a cache level that is not implemented, then a read of CSSELR_EL1 returns an UNKNOWN value in CSSELR_EL1.Level.

If the CSSELR_EL1.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR_EL1 an implementation must perform one of the following behaviors:

- The CCSIDR_EL1 read executes as a `NOP`.
- The CCSIDR_EL1 read is UNDEFINED.
- The CCSIDR_EL1 read returns an UNKNOWN value.

### K1.2.11 Crossing a page boundary with different memory types or Shareability attributes

A memory access from a load or store instruction that crosses a page boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- All memory accesses generated by the instruction use the memory type and Shareability attributes associated with the first address accessed by the instruction.
- All memory accesses generated by the instruction use the memory type and Shareability attributes associated with the last address accessed by the instruction.
- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an Alignment fault caused by the memory type.

  For the Non-secure EL1&0 translation regime:
  - If the stage 1 translation generated the mismatch then the resulting exception is taken to EL1.
  - If the stage 2 translation generated the mismatch then the resulting exception is taken to EL2.
  - If both stages of translation generate the mismatch then the exception can be taken to either EL1 or EL2.
- The instruction executes as a `NOP`.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### K1.2.12 Crossing a peripheral boundary with a Device access

Performing memory accesses from one load or store instruction to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between memory accesses.
- The instruction generates an alignment fault caused by the memory type.

  For the Non-secure EL1&0 translation regime:

  — If the stage 1 translation causes the boundary to be crossed then the resulting exception is taken to EL1.

  — If the stage 2 translation causes the boundary to be crossed then the resulting exception is taken to EL2.

  — If both stages of translation cause the boundary to be crossed then the resulting exception can be taken to either EL1 or EL2.
- The instruction executes as a `NOP`.

**Note**

The boundary referred to is between two Device memory regions that are both:
- Of the size of the smallest implemented translation granule.
- Aligned to the size of the smallest implemented translation granule.

### K1.2.13 CONSTRAINED UNPREDICTABLE behavior in Debug state

*Behavior in Debug state* on page H2-4832 of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.

### K1.2.14 CONSTRAINED UNPREDICTABLE behavior for A64 instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A64 instructions listed in Chapter C6 *A64 Base Instruction Descriptions* and Chapter C7 *A64 Advanced SIMD and Floating-point Instruction Descriptions*.

#### LDAXP

For a description of this instruction and the encoding, see *LDAXP* on page C6-540.

##### *CONSTRAINED UNPREDICTABLE behavior*

If `t == t2`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## LDNP (SIMD&FP)

For a description of this instruction and the encoding, see *LDNP (SIMD&FP)* on page C7-1128.

### *CONSTRAINED UNPREDICTABLE behavior*

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value.

## LDP

For a description of this instruction and the encoding, see *LDP* on page C6-554.

### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

If t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs all of the loads using the specified addressing mode, and the register loaded is set to an UNKNOWN value.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

## LDP (SIMD&FP)

For a description of this instruction and the encoding, see *LDP (SIMD&FP)* on page C7-1130.

### *CONSTRAINED UNPREDICTABLE behavior*

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## LDPSW

For a description of this instruction and the encoding, see *LDPSW* on page C6-558.

### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs all of the loads using the specified addressing mode, and the register loaded is set to an UNKNOWN value.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

## LDR (immediate)

For a description of this instruction and the encoding, see *LDR (immediate)* on page C6-561.

### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### LDRB (immediate)

For a description of this instruction and the encoding, see *LDRB (immediate)* on page C6-570.

#### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

### LDRH (immediate)

For a description of this instruction and the encoding, see *LDRH (immediate)* on page C6-576.

#### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

### LDRSB (immediate)

For a description of this instruction and the encoding, see *LDRSB (immediate)* on page C6-582.

#### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## LDRSH (immediate)

For a description of this instruction and the encoding, see *LDRSH (immediate)* on page C6-588.

### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

## LDRSW (immediate)

For a description of this instruction and the encoding, see *LDRSW (immediate)* on page C6-594.

### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

## LDXP

For a description of this instruction and the encoding, see *LDXP* on page C6-628.

### *CONSTRAINED UNPREDICTABLE behavior*

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### STP

For a description of this instruction and the encoding, see *STP* on page C6-742.

#### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

### STLXP

For a description of this instruction and the encoding, see *STLXP* on page C6-728.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `s == t || (s == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.

### STLXR

For a description of this instruction and the encoding, see *STLXR* on page C6-731.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.

### STLXRB

For a description of this instruction and the encoding, see *STLXRB* on page C6-734.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.

### STLXRH

For a description of this instruction and the encoding, see *STLXRH* on page C6-737.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.

### STR (immediate)

For a description of this instruction and the encoding, see *STR (immediate)* on page C6-746.

#### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

### STRB (immediate)

For a description of this instruction and the encoding, see *STRB (immediate)* on page C6-752.

#### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### STRH (immediate)

For a description of this instruction and the encoding, see *STRH (immediate)* on page C6-758.

#### *CONSTRAINED UNPREDICTABLE behavior*

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.

**Note**

Pre-indexed addressing and post-indexed addressing imply writeback.

### STXP

For a description of this instruction and the encoding, see *STXP* on page C6-777.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `s == t || (s == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.

### STXR

For a description of this instruction and the encoding, see *STXR* on page C6-780.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.

### STXRB

For a description of this instruction and the encoding, see *STXRB* on page C6-783.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.

### STXRH

For a description of this instruction and the encoding, see *STXRH* on page C6-786.

#### *CONSTRAINED UNPREDICTABLE behavior*

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs the store to an UNKNOWN address.

## K1.2.15 Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions DC CISW, DC CSW, and DC ISW, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

**Note**

This CONSTRAINED UNPREDICTABLE behavior applies, also, to the AArch32 cache maintenance by set/way instructions DCCISW, DCCSW, and DCISW.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K2
# Recommended External Debug Interface

This appendix describes the recommended external debug interface. It contains the following sections:


- *About the recommended external debug interface* on page K2-5516.
- *PMUEVENT bus* on page K2-5519.
- *Recommended authentication interface* on page K2-5520.
- *Management registers and CoreSight compliance* on page K2-5523.


**Note**

This recommended external debug interface specification is not part of the ARM architecture specification. Implementers and users of the ARMv8 architecture must not consider this appendix as a requirement of the architecture. It is included as an appendix to this manual only:

- As reference material for users of ARM products that implement this interface.
- As an example of how an external debug interface might be implemented.

The inclusion of this appendix is no indication of whether any ARM products might, or might not, implement this external debug interface. For details of the implemented external debug interface you must always see the appropriate product documentation.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K2.1 About the recommended external debug interface

See the *Note* on the first page of this appendix for information about the architectural status of this recommended debug interface.

This specification provides a recommended external debug interface for ARMv8 to define a standard set of connections for validation environments. In general, the connection between components, such as between the PE and Trace extension, is not described here, although the table does include the signals for the CTI connection. Table K2-1 shows the signals in the recommended interface.

**Table K2-1 Recommended debug interface signals**

| Name | Direction | Description | Notes |
|---|---|---|---|
| **DBGEN** | In | External debug enable | - |
| **SPIDEN** | In | Secure privileged external debug enable | - |
| | | Secure privileged self-hosted debug enable | Only in Secure AArch32 modes when enabled by MDCR_EL3.SPD32 |
| **NIDEN** | In | External profiling and trace enable | - |
| **SPNIDEN** | In | Secure external profiling and trace enable | - |
| **EDBGRQ** | In | External halt request | Provided for legacy connections only. |
| **DBGACK** | Out | External halt request acknowledge | |
| **COMMIRQ** | Out | DCC interrupt | Interface to an interrupt controller. See *Interrupt-driven use of the DCC* on page H4-4902 and the pseudocode for function `CheckForDCCInterrupts()` in *Pseudocode description of the operation of the DCC and ITR registers* on page H4-4903. |
| **PMUIRQ** | Out | Performance Monitor overflow | Interface to an interrupt controller. See *Behavior on overflow* on page D5-1861. |
| **COMMRX** | Out | DTRRX is full | Provided for legacy connection to an interrupt controller only. See *Interrupt-driven use of the DCC* on page H4-4902 and the pseudocode for function `CheckForDCCInterrupts()` in *Pseudocode description of the operation of the DCC and ITR registers* on page H4-4903. |
| **COMMTX** | Out | DTRTX is empty | |
| **PMUEVENT[n:0]** | Out | Performance Monitors event bus | See *PMUEVENT bus* on page K2-5519. |
| **DBGNOPWRDWN** | Out | Core no powerdown request | Interface to a power controller. See DBGPRCR_EL1.CORENPDRQ. |
| **DBGPWRUPREQ** | Out | Core powerup request | Interface to a power controller. See EDPRCR.COREPURQ. |
| **DBGRSTREQ** | Out | Warm reset request | Interface to a power controller. See EDPRCR.CWRR. |
| **DBGBUSCANCELREQ** | Out | All asynchronous entry to Debug state | Extension to the bus interface. See EDRCR.CBRRQ. |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K2-1 Recommended debug interface signals (continued)**

| Name | Direction | Description | Notes |
|---|---|---|---|
| **DBGPWRDUP** | In | Core powerup status | Interface to a power controller. See EDPRSR.PU. |
| **DBGROMADDR[n:12]** | In | MDRAR_EL1.ROMADDR | *n* depends on the size of the physical address space. |
| **DBGROMADDRV** | In | MDRAR_EL1.Valid | - |
| **PRESETDBG** | In | External debug reset | - |
| **CPUPORESET** | In | Cold reset | - |
| **CORERESET** | In | Warm reset | - |
| **PSELDBG** | In | Debug APB slave port | For details see *AMBA APB3*. ARM recommends a single slave port for all integrated debug components. **PADDRDBG31** distinguishes memory-mapped and DAP accesses: **0** Memory-mapped access **1** DAP access |
| **PENABLEDBG** | In | | |
| **PWRITEDBG** | In | | |
| **PRDATADBG[31:0]** | Out | | |
| **PWDATADBG[31:0]** | In | | |
| **PADDRDBG[n:2]**[a] | In | | |
| **PREADYDBG** | Out | | |
| **PSLVERRDBG** | Out | | |
| **PCLKDBG** | In | | |
| **PCLKENDBG** | In | | |
| **CTICHIN** | In | Asynchronous CoreSight channel interface | For details, see *CoreSight™ v1.0 Architecture Specification*. The ACK signals are not required if the channel interface is synchronous. |
| **CTICHOUTACK** | In | | |
| **CTICHOUT** | Out | | |
| **CTICHINACK** | Out | | |
| **CTIIRQ** | Out | CTI interrupt, see *Description and allocation of CTI triggers* on page H5-4915 | Implements a handshake for an edge-sensitive interrupt. |
| **CTIIRQACK** | In | | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K2-1 Recommended debug interface signals (continued)**

| Name | Direction | Description | Notes |
|---|---|---|---|
| **ATDATA[nx8-1:0]** | Out | AMBA 4 ATB interface | For details, see *AMBA 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1*. Only available if the OPTIONAL Trace extension is implemented. |
| **ATBYTES[n-1:0]** | Out | | |
| **ATID[6:0]** | Out | | |
| **ATREADY** | In | | |
| **ATVALID** | Out | | |
| **AFREADY** | Out | | |
| **AFVALID** | Out | | |
| **SYNCREQ** | In | | |
| **ATCLK** | In | | |
| **ATCLKEN** | In | | |
| **ATRESET** | In | | |

a. The value of *n* depends on the size of the address space occupied by the Debug port.

Figure K2-1 shows the recommended debug interface.

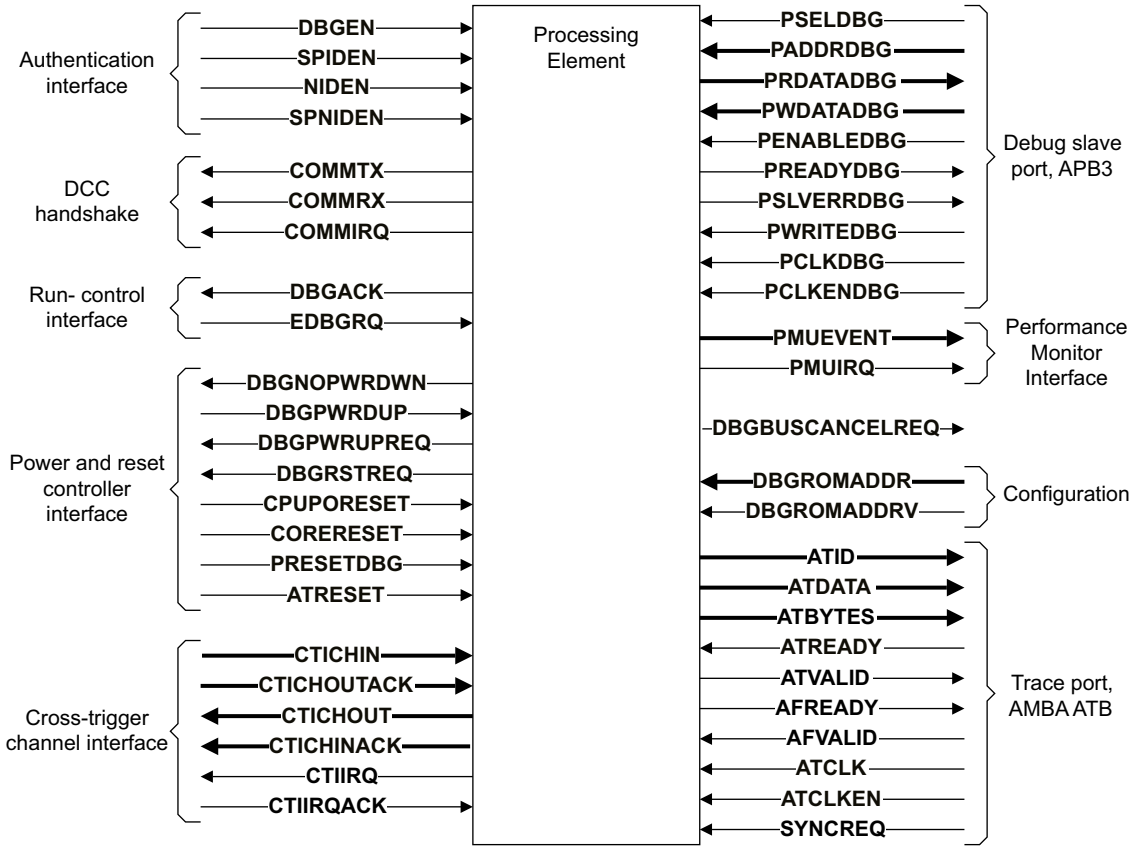



**Figure K2-1 Recommended external debug interface, including the APB3 slave port**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K2.2 PMUEVENT bus

The **PMUEVENT** bus exports Performance Monitor events from the PE to an on-chip agent. ARM recommends that it has the following characteristics:

- The bus is synchronous.
- The width of the bus is IMPLEMENTATION DEFINED.
- It is IMPLEMENTATION DEFINED which events are exported on the bus.
- Each exported event occupies a contiguous sub-field of the bus. ARM recommends that the sub-fields of the bus are occupied in the same order as the event numbers.
- If the event can only occur once per cycle, it occupies a single bit. If the event can occur more than once per cycle, it is IMPLEMENTATION DEFINED how the event is encoded. The encoding depends on constraints such as the designated use of the event bus and the number of pins available. For example, the event can be encoded:
  - As a count, using a plain binary number. This is the most useful encoding when exporting to an external counter. It is not a useful encoding for exporting to a Trace extension external input.
  - As a count, using thermometer encoding. This is the most useful encoding when exporting to a Trace extension.
  - Using a single bit encoding to indicate whether the event count is zero or nonzero. This is useful for exporting to an activity monitor where the number of pins is constrained.

If a Trace extension is implemented, the **PMUEVENT** bus is normally connected to the Trace extension using the external inputs. TRCEXTINSELR multiplexes a wide **PMUEVENT** bus to a narrow set of inputs. An external **PMUEVENT** bus might also be provided. For more information, contact ARM.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K2.3 Recommended authentication interface

The details of the debug authentication interface are IMPLEMENTATION DEFINED.

ARM recommends the use of the CoreSight interface, which has four signals for external debug authentication:

- **DBGEN**,
- **SPIDEN**.
- **NIDEN**.
- **SPNIDEN**.

CoreSight forbids asserting **SPIDEN** without also asserting **DBGEN**. CoreSight also forbids asserting **SPNIDEN** without also asserting **NIDEN**.

ARM recommends an interface in which **DBGEN** and **SPIDEN** are also used for self-hosted secure debug authentication if either:

- EL3 is using AArch32 and SDCR.SPD == `0b00`.
- Secure EL1 is using AArch32 and MDCR_EL3.SPD32 == `0b00`.

If EL3 is not implemented and the PE is in Non-secure state, **SPIDEN** and **SPNIDEN** are not implemented, and the PE behaves as if these signals were tied LOW.

If EL3 is not implemented and the PE is in Secure state, **SPIDEN** is usually connected to **DBGEN** and **SPNIDEN** is connected to **NIDEN**, but this is not required. The recommended interface is defined as if all four signals are implemented.

How the authentication signals are driven is IMPLEMENTATION DEFINED. For example, the signals might be hard-wired, connected to fuses, or to an authentication module. The architecture permits PEs within a cluster to have independent authentication interfaces, but this is not required. ARM recommends that any Trace extension has the same authentication interface as the PE it is connected to.

Table K2-2 shows the debug authentication pseudocode functions and the recommended implementations.

**Table K2-2 Recommended implementation of debug enable pseudocode functions**

| Pseudocode function | Description | Implementation |
|---|---|---|
| `SelfHostedSecurePrivilegedInvasiveDebugEnabled()`<br>See *Pseudocode description of AArch32 Self-Hosted Secure Privileged Invasive Debug Enabled* on page K2-5521 | Secure invasive self-hosted debug enabled in AArch32 state (legacy) | (**DBGEN** AND **SPIDEN**) |
| `ExternalSecureNoninvasiveDebugEnabled()`<br>See *Pseudocode description of External Invasive Debug Enabled* on page K2-5521 | Secure non-invasive debug enabled | (**DBGEN** OR **NIDEN**) AND (**SPIDEN** OR **SPNIDEN**) |
| `ExternalSecureInvasiveDebugEnabled()`<br>See *Pseudocode description of External Secure Invasive Debug Enabled* on page K2-5521 | Secure invasive debug enabled | (**DBGEN** AND **SPIDEN**) |
| `ExternalNoninvasiveDebugEnabled()`<br>See *Pseudocode description of External Non-invasive Debug Enabled* on page K2-5521 | Non-secure non-invasive debug enabled | (**DBGEN** OR **NIDEN**) |
| `ExternalInvasiveDebugEnabled()`<br>See *Pseudocode description of External Secure Non-invasive Debug Enabled* on page K2-5522 | Non-secure invasive debug enabled | **DBGEN** |

The following assertions must apply to all implementations:

if `!ExternalInvasiveDebugEnabled()` then assert `!ExternalSecureInvasiveDebugEnabled()`

if `!ExternalNoninvasiveDebugEnabled()` then assert `!ExternalSecureNoninvasiveDebugEnabled()`


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

if `ExternalInvasiveDebugEnabled()` then assert `ExternalNoninvasiveDebugEnabled()`

if `ExternalSecureInvasiveDebugEnabled()` then assert `ExternalSecureNoninvasiveDebugEnabled()`

The definition for the `Debug_authentication()` function is as follows:

```
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
```

### K2.3.1 Pseudocode description of AArch32 Self-Hosted Secure Privileged Invasive Debug Enabled

The pseudocode for the `SelfHostedSecurePrivilegedInvasiveDebugEnabled()` function is as follows:

```
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// ========================================================

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // In the recommended interface, SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;
```

### K2.3.2 Pseudocode description of External Invasive Debug Enabled

The pseudocode for the `ExternalInvasiveDebugEnabled()` function is as follows:

```
// ExternalInvasiveDebugEnabled()
// ==============================

boolean ExternalInvasiveDebugEnabled()
    // In the recommended interface, ExternalInvasiveDebugEnabled returns the state of the DBGEN
    // signal.
    return DBGEN == HIGH;
```

### K2.3.3 Pseudocode description of External Secure Invasive Debug Enabled

The pseudocode for the `ExternalSecureInvasiveDebugEnabled()` function is as follows:

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================

boolean ExternalSecureInvasiveDebugEnabled()
    // In the recommended interface, ExternalSecureInvasiveDebugEnabled returns the state of the
    // (DBGEN AND SPIDEN) signal.
    // CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

### K2.3.4 Pseudocode description of External Non-invasive Debug Enabled

The pseudocode for the `ExternalNoninvasiveDebugEnabled()` function is as follows:

```
// ExternalNoninvasiveDebugEnabled()
// =================================

boolean ExternalNoninvasiveDebugEnabled()
    // In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN
    // OR NIDEN) signal.
    return ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### K2.3.5 Pseudocode description of External Secure Non-invasive Debug Enabled

The pseudocode for the `ExternalSecureNoninvasiveDebugEnabled()` function is as follows:

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================

boolean ExternalSecureNoninvasiveDebugEnabled()
    // In the recommended interface, ExternalSecureNoninvasiveDebugEnabled returns the state of the
    // (DBGEN OR NIDEN) AND (SPIDEN OR SPNIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K2.4 Management registers and CoreSight compliance

The CoreSight architecture requires the implementation of a set of management registers that occupy the memory map from `0xF00` upwards in each of the debug components.

CoreSight compliance and complete implementation of the management registers is OPTIONAL, but ARM recommends that the registers are implemented.

The CoreSight architecture specification recommends that any integration test registers are implemented starting from `0xEFC` downwards. Each of the debug components has an IMPLEMENTATION DEFINED region from `0xE80` to `0xEFC` for this purpose.

### K2.4.1 Coresight interface register map

Table K2-3 shows the external management register maps for the following registers:

**ED** These are the external debug register.

**CTI** These are the Cross-trigger interface registers.

**PMU** These are the Performance Monitors registers.

**Table K2-3 Coresight interface register map**

| Offset | Mnemonic | | | Name |
|---|---|---|---|---|
| | ED | CTI | PMU | |
| 0xF00 | EDITCTRL | CTIITCTRL | PMITCTRL | Integration Model Control registers |
| 0xF04-0xF9C | - | - | - | Reserved, RES0 |
| 0xFA0 | DBGCLAIMSET_EL1[a] | CTICLAIMSET[b] | - | CLAIM Tag Set registers |
| 0xFA4 | DBGCLAIMCLR_EL1[a] | CTICLAIMCLR[b] | - | CLAIM Tag Clear registers |
| 0xFA8 | EDDEVAFF0[a] | CTIDEVAFF0[c] | PMDEVAFF0 | Device Affinity registers |
| 0xFAC | EDDEVAFF1[a] | CTIDEVAFF1[c] | PMDEVAFF1 | |
| 0xFB0 | EDLAR[d] | CTILAR[d] | PMLAR[d] | Lock Access register |
| 0xFB4 | EDLSR[d] | CTILSR[d] | PMLSR[d] | Lock Status register |
| 0XFB8 | DBGAUTHSTATUS_EL1[a] | CTIAUTHSTATUS | PMAUTHSTATUS | Authentication Status register |
| 0xFBC | EDDEVARCH | CTIDEVARCH | PMDEVARCH | Device Architecture register |
| 0xFC0 | EDDEVID2[a] | CTIDEVID2[a] | - | Device ID register |
| 0xFC4 | EDDEVID1[a] | CTIDEVID1[a] | - | |
| 0xFC8 | EDDEVID[a] | CTIDEVID[a] | - | |
| 0xFCC | EDDEVTYPE | CTIDEVTYPE | PMDEVTYPE | Device Type register |
| 0xFD0 | EDPIDR4 | CTIPIDR4 | PMPIDR4 | Peripheral ID registers |
| 0xFD4-0xFDC | - | - | - | Reserved, RES0 |
| 0xFE0 | EDPIDR0 | CTIPIDR0 | PMPIDR0 | Peripheral ID registers |
| 0xFE4 | EDPIDR1 | CTIPIDR1 | PMPIDR1 | |
| 0xFE8 | EDPIDR2 | CTIPIDR2 | PMPIDR2 | |
| 0xFEC | EDPIDR3 | CTIPIDR3 | PMPIDR3 | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K2-3 Coresight interface register map (continued)**

| Offset | Mnemonic | | | Name |
|---|---|---|---|---|
| | ED | CTI | PMU | |
| 0xFF0 | EDCIDR0 | CTICIDR0 | PMCIDR0 | Component ID registers |
| 0xFF4 | EDCIDR1 | CTICIDR1 | PMCIDR1 | |
| 0xFF8 | EDCIDR2 | CTICIDR2 | PMCIDR2 | |
| 0xFFC | EDCIDR3 | CTICIDR3 | PMCIDR3 | |

a. This register must always be implemented, regardless of whether the component is CoreSight compliant.
b. If implemented, the number of CLAIM bits is IMPLEMENTATION DEFINED and can be discovered by reading CLAIMSET.
c. If the CTI implements CTIv1, this register is not implemented. See the register description for details.
d. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

### K2.4.2 Management register access permissions

Access to the OPTIONAL Integration Control register (ITCTRL) is IMPLEMENTATION DEFINED.

If the Debug power domain is off, all register accesses return an error.

Otherwise, Table K2-4 on page K2-5525, Table K2-5 on page K2-5526, and Table K2-6 on page K2-5527 show the response to accesses by the external debug interface to the CoreSight management registers. For definitions of the terms used in the tables, see *External debug interface register access permissions summary* on page H8-4954.

**Note**

Access to the CoreSight management registers is not affected by the values of EDAD and EPMAD.

Table K2-4 on page K2-5525, Table K2-5 on page K2-5526, and Table K2-6 on page K2-5527 include reserved management registers, because the CoreSight architecture requires that these registers are always RES0. The descriptions in *Reserved and unallocated registers* on page H8-4954 does not apply to reserved management registers if the implementation is CoreSight compliant.

If OPTIONAL memory-mapped access to the external debug interface is supported, there are additional constraints on memory-mapped accesses. See *Register access permissions for memory-mapped accesses* on page H8-4950.

The terms in Table K2-4 on page K2-5525, Table K2-5 on page K2-5526, and Table K2-6 on page K2-5527 are defined as follows:

**Domain** This describes the power domain in which the register is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.

**Conditions** This lists the conditions under which the access is attempted.

To determine the access permissions for a register, read these columns from left to right, and stop at first column which lists the condition as being true.

The conditions are:

**Off** EDPRSR.PU == 0. The Core power domain is completely off, or in low-power state. In these cases the Core power domain registers cannot be accessed.

**Note**

If debug power is off, then all external debug interface accesses return an error.

**DLK** `DoubleLockStatus()` == TRUE. The OS Double Lock is locked, that is, EDPRSR.DLK == 1.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**OSLK** OSLSR.OSLK == 1. The OS Lock is locked.

**Default** This provides the default access permissions, if there are no conditions that prevent access to the register.

**SLK** This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked. See *Register access permissions for memory-mapped accesses* on page H8-4950. For all other accesses, this column is ignored.

The access permissions are:

**-** This means that the default access permission applies. See the Default column, or the SLK column, if applicable.

**RO** This means that the register or field is read-only.

**RW** This means that the register or field is read/write. Individual fields within the register might be RO. See the relevant register description for details.

**RC** This means that the bit clears to 0 after a read.

**(SE)** This means that accesses to this register have indirect write side effects. Aside effect occurs when a direct read or a direct write of a register creates an indirect write to the same register or to another register.

**WO** This means that the register or field is write-only.

**WI** This means that the register or field ignores writes.

**IMP DEF** This means that the access permissions are IMPLEMENTATION DEFINED.

**Table K2-4 External debug interface access permissions, CoreSight registers (debug)**

| Offset | Register | Domain | Conditions (priority left to right) | | | Default | SLK |
|---|---|---|---|---|---|---|---|
| | | | Off | DLK | OSLK | | |
| 0xF00 | EDITCTRL | IMP DEF | IMPLEMENTATION DEFINED | | | IMP DEF | RO/WI |
| 0xF04-0xF8C | Reserved | Debug | - | - | - | RES0 | - |
| 0xFA0 | DBGCLAIMSET_EL1 | Core | Error | Error | Error | RW (SE) | RO |
| 0xFA4 | DBGCLAIMCLR_EL1 | Core | Error | Error | Error | RW (SE) | RO |
| 0xFA8 | EDDEVAFF0 | Debug | - | - | - | RO | - |
| 0xFAC | EDDEVAFF1 | Debug | - | - | - | RO | - |
| 0xFB0 | EDLAR | Debug | - | - | - | WO (SE) | - |
| 0xFB4 | EDLSR | Debug | - | - | - | RO | - |
| 0xFB8 | DBGAUTHSTATUS_EL1 | Debug | - | - | - | RO | - |
| 0xFBC | EDDEVARCH | Debug | - | - | - | RO | - |
| 0xFC0 | EDDEVID2 | Debug | - | - | - | RO | - |
| 0xFC4 | EDDEVID1 | Debug | - | - | - | RO | - |
| 0xFC8 | EDDEVID | Debug | - | - | - | RO | - |
| 0xFCC | EDDEVTYPE | Debug | - | - | - | RO | - |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K2-4 External debug interface access permissions, CoreSight registers (debug) (continued)**

| Offset | Register | Domain | Conditions (priority left to right) | | | Default | SLK |
|---|---|---|---|---|---|---|---|
| | | | Off | DLK | OSLK | | |
| 0xFD0 | EDPIDR4 | Debug | - | - | - | RO | - |
| 0xFD4-0xFDC | Reserved | Debug | - | - | - | RES0 | - |
| 0xFE0-0xFEC | EDPIDR0 | Debug | - | - | - | RO | - |
| 0xFE4 | EDPIDR1 | Debug | - | - | - | RO | - |
| 0xFE8 | EDPIDR2 | Debug | - | - | - | RO | - |
| 0xFEC | EDPIDR3 | Debug | - | - | - | RO | - |
| 0xFF0 | EDCIDR0 | Debug | - | - | - | RO | - |
| 0xFF4 | EDCIDR1 | Debug | - | - | - | RO | - |
| 0xFF8 | EDCIDR2 | Debug | - | - | - | RO | - |
| 0xFFC | EDCIDR3 | Debug | - | - | - | RO | - |

**Table K2-5 External debug interface access permissions, CoreSight registers (CTI)**

| Offset | Register | Domain | Conditions (priority left to right) | | | Default | SLK |
|---|---|---|---|---|---|---|---|
| | | | Off | DLK | OSLK | | |
| 0xF00 | CTIITCTRL | IMP DEF | IMPLEMENTATION DEFINED | | | IMP DEF | RO/WI |
| 0xF04-0xF8C | Reserved | Debug | - | - | - | RES0 | - |
| 0xFA0 | CTICLAIMSET | Debug | - | - | - | RW (SE) | RO |
| 0xFA4 | CTICLAIMCLR | Debug | - | - | - | RW (SE) | RO |
| 0xFA8 | CTIDEVAFF0 | Debug | - | - | - | RO | - |
| 0xFAC | CTIDEVAFF1 | Debug | - | - | - | RO | - |
| 0xFB0 | CTILAR | Debug | - | - | - | WO (SE) | - |
| 0xFB4 | CTILSR | Debug | - | - | - | RO | - |
| 0xFB8 | CTIAUTHSTATUS | Debug | - | - | - | RO | - |
| 0xFBC | CTIDEVARCH | Debug | - | - | - | RO | - |
| 0xFC0 | CTIDEVID2 | Debug | - | - | - | RO | - |
| 0xFC4 | CTIDEVID1 | Debug | - | - | - | RO | - |
| 0xFC8 | CTIDEVID | Debug | - | - | - | RO | - |
| 0xFCC | CTIDEVTYPE | Debug | - | - | - | RO | - |
| 0xFD0 | CTIPIDR4 | Debug | - | - | - | RO | - |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K2-5 External debug interface access permissions, CoreSight registers (CTI) (continued)**

| Offset | Register | Domain | Conditions (priority left to right) | | | Default | SLK |
|---|---|---|---|---|---|---|---|
| | | | Off | DLK | OSLK | | |
| 0xFD4-0xFDC | Reserved | Debug | - | - | - | RES0 | - |
| 0xFE0 | CTIPIDR0 | Debug | - | - | - | RO | - |
| 0xFE4 | CTIPIDR1 | Debug | - | - | - | RO | - |
| 0xFE8 | CTIPIDR2 | Debug | - | - | - | RO | - |
| 0xFEC | CTIPIDR3 | Debug | - | - | - | RO | - |
| 0xFF0 | CTICIDR0 | Debug | - | - | - | RO | - |
| 0xFF4 | CTICIDR1 | Debug | - | - | - | RO | - |
| 0xFF8 | CTICIDR2 | Debug | - | - | - | RO | - |
| 0xFFC | CTICIDR3 | Debug | - | - | - | RO | - |

**Table K2-6 External debug interface access permissions, CoreSight registers (PMU)**

| Offset | Register | Domain | Conditions (priority left to right) | | | Default | SLK |
|---|---|---|---|---|---|---|---|
| | | | Off | DLK | OSLK | | |
| 0xF00 | PMITCTRL | IMP DEF | IMPLEMENTATION DEFINED | | | IMP DEF | RO/WI |
| 0xF04-0xFA4 | Reserved | Debug | - | - | - | RES0 | - |
| 0xFA8 | PMDEVAFF0 | Debug | - | - | - | RO | - |
| 0xFAC | PMDEVAFF1 | Debug | - | - | - | RO | - |
| 0xFB0 | PMLAR | Debug | - | - | - | WO (SE) | - |
| 0xFB4 | PMLSR | Debug | - | - | - | RO | - |
| 0xFB8 | PMAUTHSTATUS | Debug | - | - | - | RO | - |
| 0xFBC | PMDEVARCH | Debug | - | - | - | RO | - |
| 0xFC0-0xFC8 | Reserved | Debug | - | - | - | RES0 | - |
| 0xFCC | PMDEVTYPE | Debug | - | - | - | RO | - |
| 0xFD0 | PMPIDR4 | Debug | - | - | - | RO | - |
| 0xFD4-0xFDC | Reserved | Debug | - | - | - | RES0 | - |
| 0xFE0 | PMPIDR0 | Debug | - | - | - | RO | - |
| 0xFE4 | PMPIDR1 | Debug | - | - | - | RO | - |
| 0xFE8 | PMPIDR2 | Debug | - | - | - | RO | - |
| 0xFEC | PMPIDR3 | Debug | - | - | - | RO | - |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K2-6 External debug interface access permissions, CoreSight registers (PMU) (continued)**

| Offset | Register | Domain | Conditions (priority left to right) | | | Default | SLK |
|---|---|---|---|---|---|---|---|
| | | | Off | DLK | OSLK | | |
| 0xFF0 | PMCIDR0 | Debug | - | - | - | RO | - |
| 0xFF4 | PMCIDR1 | Debug | - | - | - | RO | - |
| 0xFF8 | PMCIDR2 | Debug | - | - | - | RO | - |
| 0xFFC | PMCIDR3 | Debug | - | - | - | RO | - |

### K2.4.3 Management register resets

Table K2-7 shows the management register resets. This table does not include:

- Read-only identification registers that have a fixed value from reset. These registers include those with the DEVAFFn, DEVARCH, DEVID{n}, DEVTYPE, PIDRn, and CIDRn mnemonics.
- Registers that have the AUTHSTATUS mnemonic. This is a read-only status register that reflects the status outside of the reset domain of the register.
- Registers that have the LAR mnemonic. These are write-only registers that only have an effect on writes.

All other fields in the management registers are reset to an IMPLEMENTATION DEFINED value which can be UNKNOWN. The registers are in the reset domain specified in the table.

Table K2-7 shows a summary of the management register resets.

**Table K2-7 Management register resets**

| Register | Reset domain | Field | Value | Description |
|---|---|---|---|---|
| CTIITCTRL<br>EDITCTRL<br>PMITCTRL | IMPLEMENTATION DEFINED | IME | 0 | Integration mode enable |
| DBGCLAIMCLR_EL1 | Cold reset | CLAIM | 0x00 | CLAIM tags |
| CTICLAIMCLR | External debug | CLAIM | 0x00000000 | |
| CTILSR[a]<br>EDLSR[a]<br>PMLSR[a] | External debug | SLK | 1 | Software Lock |

a. Only if the OPTIONAL Software Lock is implemented


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K3
# Recommendations for Performance Monitors Event Numbers for IMPLEMENTATION DEFINED Events

This appendix describes the ARM recommendations for the use of the IMPLEMENTATION DEFINED event numbers. It contains the following sections:


- *ARM recommendations for IMPLEMENTATION DEFINED event numbers* on page K3-5530.
- *Summary of events for exceptions taken to an Exception Level using AArch64* on page K3-5544.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K3.1 ARM recommendations for IMPLEMENTATION DEFINED event numbers

These are the ARM recommendations for the use of the IMPLEMENTATION DEFINED event numbers. ARM does not define these events as rigorously as those in the architectural and microarchitectural event lists, and an implementation might:

- Modify the definition of an event to better correspond to the implementation.
- Not use some, or many, of these event numbers.
- Include cumulative occupancy for resource queues such as data access queues, and entry/exit counts, so that average latencies can be determined and counts for key resources that might exist can be separated.
- Provide registers in the IMPLEMENTATION DEFINED space to further extend such counts by, for example, specifying a minimum latency for an event to be counted.
- Use cumulative occupancy for resource queues, such as data access queues, and entry/exit counts, so that average latencies can be determined, separating out counts for key resources that might exist:
  - An implementation might also provide registers in the IMPLEMENTATION DEFINED space to further extend such counts, by, for example specifying a minimum latency for an event to be counted.

Table K3-1 lists the PMU IMPLEMENTATION DEFINED event numbers in event number order.

**Table K3-1 PMU IMPLEMENTATION DEFINED event numbers**

| Event number | Event mnemonic | Description |
|---|---|---|
| 0x040 | L1D_CACHE_RD | Level 1 data cache access, read |
| 0x041 | L1D_CACHE_WR | Level 1 data cache access, write |
| 0x042 | L1D_CACHE_REFILL_RD[a] | Level 1 data cache refill, read |
| 0x043 | L1D_CACHE_REFILL_WR[a] | Level 1 data cache refill, write |
| 0x044 | L1D_CACHE_REFILL_INNER | Level 1 data cache refill, inner |
| 0x045 | L1D_CACHE_REFILL_OUTER | Level 1 data cache refill, outer |
| 0x046 | L1D_CACHE_WB_VICTIM | Level 1 data cache write-back, victim |
| 0x047 | L1D_CACHE_WB_CLEAN | Level 1 data cache write-back, cleaning and coherency |
| 0x048 | L1D_CACHE_INVAL | Level 1 data cache invalidate |
| 0x049-0x04B | - | Reserved |
| 0x04C | L1D_TLB_REFILL_RD[a] | Level 1 data TLB refill, read |
| 0x04D | L1D_TLB_REFILL_WR[a] | Level 1 data TLB refill, write |
| 0x04E | L1D_TLB_RD | Level 1 data or unified TLB access, read |
| 0x04F | L1D_TLB_WR | Level 1 data or unified TLB access, write |
| 0x050 | L2D_CACHE_RD | Level 2 data cache access, read |
| 0x051 | L2D_CACHE_WR | Level 2 data cache access, write |
| 0x052 | L2D_CACHE_REFILL_RD[a] | Level 2 data cache refill, read |
| 0x053 | L2D_CACHE_REFILL_WR[a] | Level 2 data cache refill, write |
| 0x054-0x055 | - | Reserved |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K3-1 PMU IMPLEMENTATION DEFINED event numbers (continued)**

| Event number | Event mnemonic | Description |
|---|---|---|
| 0x056 | L2D_CACHE_WB_VICTIM | Level 2 data cache write-back, victim |
| 0x057 | L2D_CACHE_WB_CLEAN | Level 2 data cache write-back, cleaning and coherency |
| 0x058 | L2D_CACHE_INVAL | Level 2 data cache invalidate |
| 0x059-0x05B | - | Reserved |
| 0x05C | L2D_TLB_REFILL_RD[a] | Attributable Level 2 data or unified TLB refill, read |
| 0x05D | L2D_TLB_REFILL_WR[a] | Attributable Level 2 data or unified TLB refill, write |
| 0x05E | L2D_TLB_RD | Attributable Level 2 data or unified TLB access, read |
| 0x05F | L2D_TLB_WR | Attributable Level 2 data or unified TLB access, write |
| 0x060 | BUS_ACCESS_RD | Bus access, read |
| 0x061 | BUS_ACCESS_WR | Bus access, write |
| 0x062 | BUS_ACCESS_SHARED | Bus access, Normal, Cacheable, Shareable |
| 0x063 | BUS_ACCESS_NOT_SHARED | Bus access, not Normal, Cacheable, Shareable |
| 0x064 | BUS_ACCESS_NORMAL | Bus access, normal |
| 0x065 | BUS_ACCESS_PERIPH | Bus access, peripheral |
| 0x066 | MEM_ACCESS_RD | Data memory access, read |
| 0x067 | MEM_ACCESS_WR | Data memory access, write |
| 0x068 | UNALIGNED_LD_SPEC | Unaligned access, read |
| 0x069 | UNALIGNED_ST_SPEC | Unaligned access, write |
| 0x06A | UNALIGNED_LDST_SPEC | Unaligned access |
| 0x06B | - | Reserved |
| 0x06C | LDREX_SPEC | Exclusive operation speculatively executed, LDREX or LDX |
| 0x06D | STREX_PASS_SPEC | Exclusive operation speculatively executed, STREX or STX pass |
| 0x06E | STREX_FAIL_SPEC | Exclusive operation speculatively executed, STREX or STX pass |
| 0x06F | STREX_SPEC | Exclusive operation speculatively executed, STREX or STX |
| 0x070 | LD_SPEC | Operation speculatively executed, load |
| 0x071 | ST_SPEC | Operation speculatively executed, store |
| 0x072 | LDST_SPEC | Operation speculatively executed, load or store |
| 0x073 | DP_SPEC | Operation speculatively executed, integer data-processing |
| 0x074 | ASE_SPEC | Operation speculatively executed, Advanced SIMD instruction |
| 0x075 | VFP_SPEC | Operation speculatively executed, floating-point instruction |
| 0x076 | PC_WRITE_SPEC | Operation speculatively executed, software change of the PC |
| 0x077 | CRYPTO_SPEC | Operation speculatively executed, Cryptographic instruction |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K3-1 PMU IMPLEMENTATION DEFINED event numbers (continued)**

| Event number | Event mnemonic | Description |
|---|---|---|
| 0x078 | BR_IMMED_SPEC | Branch speculatively executed, immediate branch |
| 0x079 | BR_RETURN_SPEC | Branch speculatively executed, procedure return |
| 0x07A | BR_INDIRECT_SPEC | Branch speculatively executed, indirect branch |
| 0x07B | - | Reserved |
| 0x07C | ISB_SPEC | Barrier speculatively executed, ISB |
| 0x07D | DSB_SPEC | Barrier speculatively executed, DSB |
| 0x07E | DMB_SPEC | Barrier speculatively executed, DMB |
| 0x07F-0x080 | - | Reserved |
| 0x081 | EXC_UNDEF | Exception taken, Other synchronous |
| 0x082 | EXC_SVC | Exception taken, Supervisor Call |
| 0x083 | EXC_PABORT | Exception taken, Instruction Abort |
| 0x084 | EXC_DABORT | Exception taken, Data Abort and SError |
| 0x085 | - | Reserved |
| 0x086 | EXC_IRQ | Exception taken, IRQ |
| 0x087 | EXC_FIQ | Exception taken, FIQ |
| 0x088 | EXC_SMC | Exception taken, Secure Monitor Call |
| 0x089 | - | Reserved |
| 0x08A | EXC_HVC | Exception taken, Hypervisor Call |
| 0x08B | EXC_TRAP_PABORT | Exception taken, Instruction Abort not taken locally |
| 0x08C | EXC_TRAP_DABORT | Exception taken, Data Abort or SError not taken locally |
| 0x08D | EXC_TRAP_OTHER | Exception taken, Other traps not taken locally |
| 0x08E | EXC_TRAP_IRQ | Exception taken, IRQ not taken locally |
| 0x08F | EXC_TRAP_FIQ | Exception taken, FIQ not taken locally |
| 0x090 | RC_LD_SPEC | Release consistency operation speculatively executed, Load-Acquire |
| 0x091 | RC_ST_SPEC | Release consistency operation speculatively executed, Store-Release |
| 0x092-0x09F | - | Reserved |
| 0x0A0 | L3D_CACHE_RD | Attributable Level 3 data or unified cache access, read |
| 0x0A1 | L3D_CACHE_WR | Attributable Level 3 data or unified cache access, write |
| 0x0A2 | L3D_CACHE_REFILL_RD[a] | Attributable Level 3 data or unified cache refill, read |
| 0x0A3 | L3D_CACHE_REFILL_WR[a] | Attributable Level 3 data or unified cache refill, write |
| 0x0A4-0x0A5 | - | Reserved |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K3-1 PMU IMPLEMENTATION DEFINED event numbers (continued)**

| Event number | Event mnemonic | Description |
|---|---|---|
| 0x0A6 | L3D_CACHE_WB_VICTIM | Attributable Level 3 data or unified cache write-back, victim |
| 0x0A7 | L3D_CACHE_WB_CLEAN | Attributable Level 3 data or unified cache write-back, cache clean |
| 0x0A8 | L3D_CACHE_INVAL | Attributable Level 3 data or unified cache access, invalidate |

a. For more information, see *Relationship between REFILL events and associated access events* on page K3-5543.

**0x040, L1D_CACHE_RD, Level 1 data cache access, read**

This event is similar to Level 1 data cache access, L1D_CACHE, but the counter counts only memory-read operations that access at least the Level 1 data or unified cache.

**0x041, L1D_CACHE_WR, Level 1 data cache access, write**

This event is similar to Level 1 data cache access, L1D_CACHE, but the counter counts only memory-write operations that access at least the Level 1 data or unified cache.

The counter counts DC ZVA as a store instruction.

**0x042, L1D_CACHE_REFILL_RD, Level 1 data cache refill, read**

This event is similar to Level 1 data cache refill, L1D_CACHE_REFILL, but the counter counts only memory-read operations that cause a refill of at least the Level 1 data or unified cache.

See also *Relationship between REFILL events and associated access events.* on page K3-5543.

**0x043, L1D_CACHE_REFILL_WR, Level 1 data cache refill, write**

This event is similar to Level 1 data cache refill, L1D_CACHE_REFILL, but the counter counts only memory-write operations that cause a refill of at least the Level 1 data or unified cache.

The counter counts DC ZVA as a store instruction.

See also *Relationship between REFILL events and associated access events.* on page K3-5543.

**0x044, L1D_CACHE_REFILL_INNER, Level 1 data cache refill, inner**

This event is similar to Level 1 data cache refill, L1D_CACHE_REFILL, but the counter counts only memory-read and memory-write operations that generate refills satisfied by transfer from another cache inside of the immediate cluster.

**Note**

The boundary between *inner* and *outer* is IMPLEMENTATION DEFINED, and it is not necessarily linked to other similar boundaries, such as the boundary between Inner Cacheable and Outer Cacheable or the boundary between Inner Shareable and Outer Shareable.

**0x045, L1D_CACHE_REFILL_OUTER, Level 1 data cache refill, outer**

This event is similar to Level 1 data cache refill, L1D_CACHE_REFILL, but the counter counts only memory-read and memory-write operations that generate refills satisfied from outside of the immediate cluster.

**0x046, L1D_CACHE_WB_VICTIM, Attributable Level 1 data cache write-back, victim**

This event is similar to Level 1 data cache write-back, L1D_CACHE_WB, but the counter counts only write-backs that are a result of the line being allocated for an access made by the PE.

Write-backs caused by the execution of a cache maintenance instruction are not counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache is counted. For example, this might occur if the PE detects streaming writes to memory and does not allocate lines to the cache, or as the result of a DC ZVA.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**0x047, L1D_CACHE_WB_CLEAN, Attributable Level 1 data cache write-back, cleaning and coherency**

This event is similar to Level 1 data cache write-back, L1D_CACHE_WB, but the counter counts only write-backs that are a result of a coherency operation made by another PE or are caused by the execution of a cache maintenance instruction. Whether write-backs caused by the execution of a cache maintenance instruction are counted is IMPLEMENTATION DEFINED.

If a coherency request from a requestor outside the PE results in a write-back, it is an Unattributable event.

**Note**

The transfer of a dirty cache line from the Level 1 data cache of this PE to the Level 1 data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a Level 2 cache or memory.

**0x048, L1D_CACHE_INVAL, Level 1 data cache invalidate**

The counter counts each invalidation of a cache line in the Level 1 data or unified cache.

The counter does not count events:

- If a cache refill invalidates a line.
- For locally-executed cache maintenance instructions that operate by set/way.

If a coherency request from a requestor outside the PE results in a write-back, it is an Unattributable event.

**0x04C, L1D_TLB_REFILL_RD, Level 1 data TLB refill, read**

This event is similar to Level 1 data TLB refill, L1D_TLB_REFILL, but the counter counts only memory-read operations that cause a data TLB refill of a least the Level 1 data or unified TLB.

See also *Relationship between REFILL events and associated access events.* on page K3-5543.

**0x04D, L1D_TLB_REFILL_WR, Level 1 data TLB refill, write**

This event is similar to Level 1 data TLB refill, L1D_TLB_REFILL, but the counter counts only memory-write operations that cause a data TLB refill of a least the Level 1 data or unified TLB.

The counter counts DC ZVA as a store instruction.

See also *Relationship between REFILL events and associated access events.* on page K3-5543.

**0x04E, L1D_TLB_RD, Level 1 data or unified TLB access, read**

This event is similar to Level 1 data or unified TLB access, L1D_TLB, but the counter counts only memory-read operations that cause a TLB access to at least the Level 1 data or unified TLB.

**0x04F, L1D_TLB_WR, Level 1 data or unified TLB access, write**

This event is similar to Level 1 data or unified TLB access, L1D_TLB, but the counter counts only memory-write operations that cause a TLB access to at least the Level 1 data or unified TLB.

**0x050, L2D_CACHE_RD, Level 2 data cache access, read**

This event is similar to Level 2 data cache access, L2D_CACHE, but the counter counts only memory-read operations that access at least the Level 2 data or unified cache.

**0x051, L2D_CACHE_WR, Level 2 data cache access, write**

This event is similar to Level 2 data cache access, L2D_CACHE, but the counter counts only memory-write operations that access at least the Level 2 data or unified cache.

The counter counts DC ZVA as a store instruction.

**0x052, L2D_CACHE_REFILL_RD, Level 2 data cache refill, read**

This event is similar to Level 2 data cache refill, L2D_CACHE_REFILL, but the counter counts only memory-read operations that cause a refill of at least the Level 2 data or unified cache.

See also *Relationship between REFILL events and associated access events.* on page K3-5543.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**0x053, L2D_CACHE_REFILL_WR, Level 2 data cache refill, write**

This event is similar to Level 2 data cache refill, L2D_CACHE_REFILL, but the counter counts only memory-write operations that cause a refill of at least the Level 2 data or unified cache.

The counter counts DC ZVA as a store instruction.

See also *Relationship between REFILL events and associated access events.* on page K3-5543.

**0x056, L2D_CACHE_WB_VICTIM, Attributable Level 2 data cache write-back, victim**

This event is similar to Level 2 data cache write-back, L2D_CACHE_WB, but the counter counts only write-backs that are a result of the line being allocated for an access made by the PE.

Write-backs caused by the execution of a cache maintenance instruction are not counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache is counted. For example, this might occur if the PE detects streaming writes to memory and does not allocate lines to the cache, or as the result of a DC ZVA.

**0x057, L2D_CACHE_WB_CLEAN, Attributable Level 2 data cache write-back, cleaning and coherency**

This event is similar to Level 2 data cache write-back, L2D_CACHE_WB, but the counter counts only write-backs that are a result of a coherency operation made by another PE or are caused by the execution of a cache maintenance instruction. Whether write-backs caused by the execution of a cache maintenance instruction are counted as IMPLEMENTATION DEFINED.

**Note**

The transfer of a dirty cache line from the Level 2 data cache of this PE to the Level 2 data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a Level 3 cache or memory.

If a coherency request from a requestor outside the PE results in a write-back, it is an Unattributable event.

**0x058, L2D_CACHE_INVAL, Level 2 data cache invalidate**

The counter counts each invalidation of a cache line in the Level 2 data or unified cache.

The counter does not count events:

- If a cache refill invalidates a line.
- For locally executed cache maintenance instructions that operate by set/way.

**Note**

Software that uses this event must know whether the Level 2 data cache is shared with other PEs. This event does not follow the general rule of Level 2 data cache events of only counting events that directly affect this PE.

If a coherency request from a requestor outside the PE results in a write-back, it is an Unattributable event.

**0x05C, L2D_TLB_REFILL_RD, Attributable Level 2 data or unified TLB refill, read**

This event is similar to Attributable Level 2 data or unified TLB refill, L2D_TLB_REFILL, but the counter counts only Attributable memory read operations that cause a TLB refill of at least the Level 2 data or unified TLB. See also *Relationship between REFILL events and associated access events.* on page K3-5543.

**0x05D, L2D_TLB_REFILL_WR, Attributable Level 2 data or unified TLB refill, write**

This event is similar to Attributable Level 2 data or unified TLB refill, L2D_TLB_REFILL, but the counter counts only Attributable memory write operations that cause a TLB refill of at least the Level 2 data or unified TLB. See also *Relationship between REFILL events and associated access events.* on page K3-5543.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**0x05E, L2D_TLB_RD, Attributable Level 2 data or unified TLB access, read**

This event is similar to Attributable Level 2 data or unified TLB access, L2D_TLB, but the counter counts only Attributable memory read operations that cause a TLB access to at least the Level 2 data or unified TLB.

**0x05F, L2D_TLB_WR, Attributable Level 2 data or unified TLB access, write**

This event is similar to Attributable Level 2 data or unified TLB access, L2D_TLB, but the counter counts only Attributable memory write operations that cause a TLB access to at least the Level 2 data or unified TLB.

**0x060, BUS_ACCESS_RD, Bus access, read**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-read operations that access outside the boundary of the PE and its closely-coupled caches.

**0x061, BUS_ACCESS_WR, Bus access, write**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-write operations that access outside the boundary of the PE and its closely-coupled caches.

**0x062, BUS_ACCESS_SHARED, Bus access, Normal, Cacheable, Shareable**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-read and memory-write operations that make Normal, Cacheable, Shareable accesses outside the boundary of the PE and its closely-coupled caches.

**Note**

It is IMPLEMENTATION DEFINED how the PE translates the attributes from the translation table entry for a region to the attributes on the bus.

In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.

If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

**0x063, BUS_ACCESS_NOT_SHARED, Bus access, not Normal, Cacheable, Shareable**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-read and memory-write operations that make accesses outside the boundary of the PE and its closely-coupled caches that are not Normal, Cacheable, Shareable. For example, the counter counts accesses marked as:

- Normal, Cacheable, Non-shareable.
- Normal, Non-cacheable.
- Device.

**Note**

It is IMPLEMENTATION DEFINED, how the PE translates the attributes from the translation table entries for a region to the attributes on the bus.

In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.

If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**0x064, BUS_ACCESS_NORMAL, Bus access, normal**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-read and memory-write operations that make Normal accesses outside the boundary of the PE and its closely-coupled caches. For example, the counter counts Normal, Cacheable and Normal, Non-cacheable accesses but does not count Device accesses.

**0x065, BUS_ACCESS_PERIPH, Bus access, peripheral**

This event is similar to Bus access, BUS_ACCESS, but the counter counts only memory-read and memory-write operations that make Device accesses outside the boundary of the PE and its closely-coupled caches.

**0x066, MEM_ACCESS_RD, Data memory access, read**

This event is similar to Data memory access, MEM_ACCESS, but the counter counts only memory-read operations that the PE made.

**0x067, MEM_ACCESS_WR, Data memory access, write**

This event is similar to Data memory access, MEM_ACCESS, but the counter counts only memory-write operations made by the PE.

**0x068, UNALIGNED_LD_SPEC. Unaligned access, read**

This event is similar to Data memory access, MEM_ACCESS, but the counter counts only unaligned memory-read operations that the PE made. It also counts unaligned accesses if they are subsequently transposed into multiple aligned accesses.

**0x069, UNALIGNED_ST_SPEC, Unaligned access, write**

This event is similar to Data memory access, MEM_ACCESS, but the counter counts only unaligned memory-read operations that the PE made. It also counts unaligned accesses if they are subsequently transposed into multiple aligned accesses.

**0x06A, UNALIGNED_LDST_SPEC, Unaligned access**

This event is similar to Data memory access, MEM_ACCESS, but the counter counts only unaligned memory-read operations and unaligned memory-write operations that the PE made. It also counts unaligned accesses if they are subsequently transposed into multiple aligned accesses.

**0x06C, LDREX_SPEC, Exclusive operation speculatively executed, Load-Exclusive**

The counter counts Load-Exclusive instructions speculatively executed.

The definition of speculatively executed is IMPLEMENTATION DEFINED.

**0x06D, STREX_PASS_SPEC, Exclusive operation speculatively executed, Store-Exclusive pass**

The counter counts Store-Exclusive instructions speculatively executed that completed a write.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the LDREX_SPEC event.

**0x06E, STREX_FAIL_SPEC, Exclusive operation speculatively executed, Store-Exclusive fail**

The counter counts Store-Exclusive instructions speculatively executed that fail to complete a write. It is within the IMPLEMENTATION DEFINED definition of speculatively executed whether this includes conditional instructions that fail the condition code check.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the LDREX_SPEC event.

**0x06F, STREX_SPEC, Exclusive operation speculatively executed, Store-Exclusive**

The counter counts Store-Exclusive instructions speculatively executed.

The definition of speculatively executed is IMPLEMENTATION DEFINED but it must be the same as for the LDREX_SPEC event.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

ARM recommends that this event is implemented if it is not possible to implement the exclusive operation speculatively executed, Store-Exclusive pass, and exclusive operation speculatively executed, Store-Exclusive fail, events with the same degree of speculation as the LDREX_SPEC event.

**`0x070`, LD_SPEC, Operation speculatively executed, load**

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only memory-reading instructions, as defined by the LD_RETIRED event.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the INST_SPEC event.

**`0x071`, ST_SPEC, Operation speculatively executed, store**

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only memory-writing instructions, as defined by the ST_RETIRED event.

The counter counts DC ZVA as a store operation.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the INST_SPEC event.

**`0x072`, LDST_SPEC, Operation speculatively executed, load or store**

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only memory-reading instructions and memory-writing instructions, as defined by the LD_RETIRED and ST_RETIRED events.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the INST_SPEC event.

**`0x073`, DP_SPEC, Operation speculatively executed, integer data-processing**

This event is similar to Operation speculatively executed, INST_SPEC, but counts only integer data-processing instructions. It counts the following operations that operate on the general-purpose registers:

- In AArch64 state, *Data processing - immediate* on page C3-153 and *Data processing - register* on page C3-158.
- In AArch32 state, *Data-processing instructions* on page F1-2412.

This includes `MOV` and `MVN` operations.

This event also counts the following miscellaneous instructions:

- In AArch64 state, *System register instructions* on page C3-140, *System instructions* on page C3-140, and *Hint instructions* on page C3-141.
- In AArch32 state, *PSTATE and banked register access instructions* on page F1-2420, *Banked register access instructions* on page F1-2420, *Miscellaneous instructions* on page F1-2425, other than `ISB` and preloads, and *Coprocessor instructions* on page F1-2427, other than coprocessor load and store instructions.

If the preload instructions `PRFM`, `PLD`, `PLDW`, and `PLI`, do not count as memory-reading instructions then they must count as integer data-processing instructions.

If `ISB`s do not count as software change of the PC then they must count as integer data-processing instructions.

The definition of speculatively executed is IMPLEMENTATION DEFINED, but must be the same as for the INST_SPEC event.

It is IMPLEMENTATION DEFINED whether the following instructions are counted as integer data-processing operations, SIMD operations, or floating-point operations, but ARM recommends that the instructions are all counted as integer data-processing operations:

- For AArch64 state, from the A64 floating-point convert to integer class, operations that move a value between a general-purpose register and a SIMD and floating-point register without type conversion:
  - `FMOV` (general).


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

- For AArch64 state, from the SIMD Move group, operations that move a values between a general-purpose register and an element or elements in a SIMD and floating-point register:
  - `DUP` (general).
  - `SMOV`.
  - `UMOV`.
  - `INS` (general).
- In AArch32 state:
  - `VDUP` (general-purpose register) and all `VMOV` instructions that transfer data between a general-purpose register and a SIMD and floating-point register.
  - `VMRS`.
  - `VMSR`.

**`0x074`, ASE_SPEC, Operation speculatively executed, Advanced SIMD**

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only Advanced SIMD data-processing instructions, see:

- For AArch64 state, the SIMD operations listed in *Data processing - SIMD and floating-point* on page C3-165.
- For AArch32 state, *Advanced SIMD data-processing instructions* on page F1-2431.

This includes all operations that operate on the SIMD and floating-point registers, except those that are counted as:

- Integer data-processing operations.
- Floating-point data-processing operations.
- Memory-reading operations.
- Memory-writing operations.
- Cryptographic operations other than `PMULL`, in AArch64 state.
- `VMULL`, in AArch32 state.

Advanced SIMD scalar operations are counted as Advanced SIMD operations, including those which operate on floating-point values. In AArch64 state, `PMULL`, and in AArch32 state, `VMULL` are counted as Advanced SIMD operations.

The definition of speculatively executed is IMPLEMENTATION DEFINED, but must be the same as for the INST_SPEC event.

**`0x075`, VFP_SPEC, Operation speculatively executed, floating-point**

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only floating-point data-processing instructions, see:

- In AArch64 state, the floating-point operations listed in *Data processing - SIMD and floating-point* on page C3-165.
- In AArch32 state, *Floating-point data-processing instructions* on page F1-2439.

This includes all operations that operate on the SIMD and floating-point registers as floating-point values, except for SIMD scalar operations and those that are counted as one of:

- Integer data-processing.
- Memory-reading operations.
- Memory-writing operations.

The following instructions that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point), are counted as floating-point data-processing operations:

- In AArch64 state, `FCVT{<mode>}`, `UCVTF`, and `SCVTF`.
- In AArch32, `VCVT<mode>`(floating-point), `VCVT`, `VCVTT`, and `VCVTB`.

The definition of speculatively executed is IMPLEMENTATION DEFINED, but must be the same as for the INST_SPEC event.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

`0x076`**, PC_WRITE_SPEC, Operation speculatively executed, software change of the PC**

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only software changes of the PC. Defined by the instruction architecturally executed, condition code check pass, software change of the PC event, see *Common event numbers* on page D5-1875.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the INST_SPEC event.

See also PC_WRITE_RETIRED.

`0x077`**, CRYPTO_SPEC, Operation speculatively executed, Cryptographic instruction**

This event is similar to Operation speculatively executed, INST_SPEC, but the counter counts only Cryptographic instructions, except `PMULL` and `VMULL`, see *The Cryptographic Extensions* on page C3-182.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the INST_SPEC event.

`0x078`**, BR_IMMED_SPEC, Branch speculatively executed, immediate branch**

The counter counts immediate branch instructions speculatively executed. Defined by the instruction architecturally executed, immediate branch event, see *Common event numbers* on page D5-1875.

The definition of speculatively executed is IMPLEMENTATION DEFINED.

See also BR_IMMED_RETIRED.

`0x079`**, BR_RETURN_SPEC, Branch speculatively executed, procedure return**

The counter counts procedure return instructions speculatively executed. Defined by the BR_RETURN_RETIRED event.

The definition of speculatively executed is IMPLEMENTATION DEFINED.

See also BR_RETURN_RETIRED.

`0x07A`**, BR_INDIRECT_SPEC, Branch speculatively executed, indirect branch**

The counter counts indirect branch instructions speculatively executed. This includes software change of the PC other than exception-generating instructions and immediate branch instructions.

The definition of speculatively executed is IMPLEMENTATION DEFINED.

`0x07C`**, ISB_SPEC, Barrier speculatively executed,** `ISB`

The counter counts instruction synchronization barrier instructions speculatively executed, including CP15ISB.

The definition of speculatively executed is IMPLEMENTATION DEFINED.

`0x07D`**, DSB_SPEC, Barrier speculatively executed,** `DSB`

The counter counts data synchronization barrier instructions speculatively executed, including CP15DSB.

The definition of speculatively executed is IMPLEMENTATION DEFINED.

`0x07E`**, DMB_SPEC, Barrier speculatively executed,** `DMB`

The counter counts data memory barrier instructions speculatively executed, including CP15DSB. It does not include the implied barrier operations of load/store operations with release consistency semantics.

The definition of speculatively executed is IMPLEMENTATION DEFINED.

`0x081`**, EXC_UNDEF, Exception taken, other synchronous**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only synchronous exceptions that are not counted by the other Exception taken events. This event counts only exceptions taken locally.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**0x082, EXC_SVC, Exception taken, Supervisor Call**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Supervisor Call exceptions. This event counts only exceptions taken locally.

**0x083, EXC_PABORT, Exception taken, Instruction Abort**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Instruction Abort exceptions. This event counts only exceptions taken locally.

**0x084, EXC_DABORT, Exception taken, Data Abort or SError**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Data Abort or SError interrupt exceptions. The counter counts only exceptions taken locally.

**0x086, EXC_IRQ, Exception taken, IRQ**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only IRQ exceptions. The counter counts only exceptions taken locally, including Virtual IRQ exceptions.

**0x087, EXC_FIQ, Exception taken, FIQ**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only FIQ exceptions. The counter counts only exceptions taken locally, including Virtual FIQ exceptions.

**0x088, EXC_SMC, Exception taken, Secure Monitor Call**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Secure Monitor Call exceptions. The counter does not increment on SMC instructions trapped as a Hyp Trap exception.

**0x08A, EXC_HVC, Exception taken, Hypervisor Call**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Hypervisor Call exceptions. The counter counts for both Hypervisor Call exceptions taken locally in the hypervisor and those taken as an exception from Non-secure EL1.

**0x08B, EXC_TRAP_PABORT, Exception taken, Instruction Abort not taken locally**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Instruction Abort exceptions not taken locally.

**0x08C, EXC_TRAP_DABORT, Exception taken, Data Abort or SError not taken locally**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only Data Abort or SError interrupt exceptions not taken locally.

**0x08D, EXC_TRAP_OTHER, Exception taken, other traps not taken locally**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only those traps that are not counted as:

- Exception taken, Hypervisor Call.
- Exception taken, Instruction Abort not taken locally.
- Exception taken, Data Abort or SError not taken locally.
- Exception taken, IRQ not taken locally.
- Exception taken, FIQ not taken locally.

**0x08E, EXC_TRAP_IRQ, Exception taken, IRQ not taken locally**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only IRQ exceptions not taken locally.

**0x08F, EXC_TRAP_FIQ, Exception taken, FIQ not taken locally**

This event is similar to Exception taken, EXC_TAKEN, but the counter counts only FIQ exceptions not taken locally.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**0x090, RC_LD_SPEC, Release consistency operation speculatively executed, Load-Acquire**

The counter counts Load-Acquire operations that are speculatively executed. The definition of speculatively executed is IMPLEMENTATION DEFINED.

**0x091, RC_ST_SPEC, Release consistency operation speculatively executed, Store-Release**

The counter counts Store-Release operations that are speculatively executed. The definition of speculatively executed is IMPLEMENTATION DEFINED.

**0x0A0, L3D_CACHE_RD, Attributable Level 3 data or unified cache access, read**

This event is similar to Attributable Level 3 data or unified cache access, L3D_CACHE, but the counter counts only attributable memory read operations that cause a cache access to at least the Level 3 data or unified cache.

**0x0A1, L3D_CACHE_WR, Level 3 data or unified cache access, write**

This event is similar to Attributable Level 3 data or unified cache access, L3D_CACHE, but the counter counts only attributable memory write operations that cause a cache access to at least the Level 3 data or unified cache.

**0x0A2, L3D_CACHE_REFILL_RD, Level 3 data or unified cache refill, read**

This event is similar to Attributable Level 3 data or unified cache refill, L3D_CACHE_REFILL, but the counter counts only attributable memory read operations that cause a refill of at least the Level 3 data or unified cache from outside the Level 3cache. See also *Relationship between REFILL events and associated access events.* on page K3-5543

**0x0A3, L3D_CACHE_REFILL_WR, Level 3 data or unified cache refill, write**

This event is similar to Attributable Level 3 data or unified cache refill, L3D_CACHE_REFILL, but the counter counts only attributable memory write operations that cause a refill of at least the Level 3 data or unified cache from outside the Level 3cache. See also *Relationship between REFILL events and associated access events.* on page K3-5543

**0x0A6, L3D_CACHE_WB_VICTIM, Level 3 data or unified cache write-back, victim**

This event is similar to Attributable Level 3 data cache write-back, L3D_CACHE_WB, but the counter counts only write-backs that are a result of the line being allocated for an access made by the PE.

Write-backs caused by the execution of a cache maintenance instruction are not counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache is counted. For example, this might occur if the PE detects streaming writes to memory and does not allocate lines to the cache, or as the result of a DC ZVA.

**0x0A7, L3D_CACHE_WB_CLEAN, Attributable Level 3 data or unified cache write-back, cache clean**

This event is similar to Level 3 data cache write-back, L3D_CACHE_WB, but the counter counts only write-backs that are a result of a coherency operation made by another PE or are caused by the execution of a cache maintenance instruction. Whether write-backs that are caused by the execution of a cache maintenance instruction are counted is IMPLEMENTATION DEFINED.

**Note**

The transfer of a dirty cache line from the Level 3 data cache of this PE to the Level 3 data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a Level 3 cache or memory.

If a coherency request from a requestor outside the PE results in a write-back, it is an Unattributable event.

**0x0A8, L3D_CACHE_INVAL, Level 3 data or unified cache access, invalidate**

The counter counts each invalidation of a cache line in the Level 3 data or unified cache.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

The counter does not count events:

- If a cache refill invalidates a line.
- For locally-executed cache maintenance instructions that operate by set/way.

**Note**

Software that uses this event must know whether the Level 3 data cache is shared with other PEs. This event does not follow the general rule of Level 3 data cache events of only counting Attributable events.

### K3.1.1 Relationship between REFILL events and associated access events.

CACHE_REFILL and TLB_REFILL events count the refills for accesses that are counted by the corresponding CACHE or TLB event. Table K3-2 shows this correspondence.

**Table K3-2 Relationship between REFILL events and associated access events**

| REFILL event | Access event | Ratio REFILL/Access |
|---|---|---|
| 0x042 L1D_CACHE_REFILL_RD | 0x040 L1D_CACHE_RD | Level 1 cache refill rate, read |
| 0x043 L1D_CACHE_REFILL_WR | 0x041 L1D_CACHE_WR | Level 1 cache refill rate, write |
| 0x04C L1D_TLB_REFILL_RD | 0x04E L1D_TLB_RD | Level 1 TLB refill rate, read |
| 0x04D L1D_TLB_REFILL_WR | 0x04F L1D_TLB_WR | Level 1 TLB refill rate, write |
| 0x052 L2D_CACHE_REFILL_RD | 0x050 L2D_CACHE_RD | Level 2 data cache refill rate, read |
| 0x053 L2D_CACHE_REFILL_WR | 0x051 L2D_CACHE_WR | Level 2 data cache refill rate, write |
| 0x05C L2D_TLB_REFILL_RD | 0x05E L2D_TLB_RD | Level 2 data TLB refill rate, read |
| 0x05D L2D_TLB_REFILL_WR | 0x05F L2D_TLB_WR | Level 2 data TLB refill rate, write |
| 0x0A2 L3D_CACHE_REFILL_RD | 0x0A0 L3D_CACHE_RD | Level 3 data cache refill rate, read |
| 0x0A3 L3D_CACHE_REFILL_WR | 0x0A1 L3D_CACHE_WR | Level 3 data cache refill rate, write |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K3.2 Summary of events for exceptions taken to an Exception Level using AArch64

Table K3-3 shows the events for exceptions taken to an Exception level using AArch64.

**Table K3-3 Events for exceptions taken to an Exception level using AArch64**

| ESR.EC | Description | | Event number and classification for exception taken to | |
|---|---|---|---|---|
| | | | EL1, or the current Exception level | EL2 or EL3, from below |
| 0x00 | Unknown or uncategorized | | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x01 | WFE/WFI traps | | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x03 | AArch32 CP15 MCR/MRC traps | | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x04 | AArch32 CP15 MCRR/MRRC traps | | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x05 | AArch32 CP14 MCR/MRC traps | | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x06 | AArch32 CP14 LDC/STC traps | | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x07 | Advanced SIMD or FP traps | | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x08 | AArch32 MVFR* and FPSID traps | | - | 0x08D, Other traps not taken locally |
| 0x0C | AArch32 CP14 MCRR/MRRC traps | | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x0E | Illegal instruction set state | | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x11 | AArch32 SVC | | 0x082, Supervisor Call | 0x08D, Other traps not taken locally |
| 0x12 | AArch32 HVC that is not disabled | | - | 0x08A, Hypervisor Call |
| 0x13 | AArch32 SMC that is not disabled | to EL2 | - | 0x08D, Other traps not taken locally |
| | | to EL3 | - | 0x088, Secure Monitor Call |
| 0x15 | AArch64 SVC | | 0x082, Supervisor Call | 0x08D, Other traps not taken locally |
| 0x16 | AArch64 HVC that is not disabled | | 0x08A, Hypervisor Call | 0x08A, Hypervisor Call |
| 0x17 | AArch64 SMC that is not disabled | to EL2 | - | 0x08D, Other traps not taken locally |
| | | to EL3 | 0x088, Secure Monitor Call | 0x088, Secure Monitor Call |
| 0x18 | AArch64 MSR, MRS and system instruction traps | | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x20 | Instruction abort from below | | 0x083, Instruction Abort | 0x08B, Instruction Abort not taken locally |
| 0x21 | Instruction abort from current Exception level | | 0x083, Instruction Abort | - |
| 0x22 | PC alignment | | 0x083, Instruction Abort | 0x08B, Instruction Abort not taken locally |
| 0x24 | Data Abort from below | | 0x084, Data Abort or SError | 0x08C, Data Abort or SError not taken locally |
| 0x25 | Data Abort from current Exception level | | 0x084, Data Abort or SError | - |
| 0x26 | Stack pointer alignment | | 0x084, Data Abort or SError | 0x08C, Data Abort or SError not taken locally |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K3-3 Events for exceptions taken to an Exception level using AArch64 (continued)**

| ESR.EC | Description | Event number and classification for exception taken to | |
|---|---|---|---|
| | | EL1, or the current Exception level | EL2 or EL3, from below |
| 0x28 | AArch32 FP exception | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x2C | AArch64 FP exception | 0x081, Other synchronous | 0x08D, Other traps not taken locally |
| 0x2F | SError interrupt | 0x084, Data Abort or SError | 0x08C, Data Abort or SError not taken locally |
| 0x30 | Breakpoint from below | 0x083, Instruction Abort | 0x08B, Instruction Abort not taken locally |
| 0x31 | Breakpoint from current Exception level | 0x083, Instruction Abort | - |
| 0x32 | Software step from below | 0x083, Instruction Abort | 0x08B, Instruction Abort not taken locally |
| 0x33 | Software step from current Exception level | 0x083, Instruction Abort | - |
| 0x34 | Watchpoint from below | 0x084, Data Abort or SError | 0x08C, Data Abort or SError not taken locally |
| 0x35 | Watchpoint from current Exception level | 0x084, Data Abort or SError | - |
| 0x38 | AArch32 BKPT instruction | 0x083, Instruction Abort | 0x08B, Instruction Abort not taken locally |
| 0x3A | AArch32 Vector Catch debug event | 0x083, Instruction Abort | 0x08B, Instruction Abort not taken locally |
| 0x3C | AArch64 BRK instruction | 0x083, Instruction Abort | 0x08B, Instruction Abort not taken locally |
| - | IRQ interrupt | 0x086, IRQ | 0x08E, IRQ not taken locally |
| - | FIQ interrupt | 0x087, FIQ | 0x08F, FIQ not taken locally |

**Note**

In these definitions, an exception that is *taken locally* means an exception that is taken to the default Exception level, and is not routed to another Exception level. See *Exception levels* on page D1-1514 for more information.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K4
# Recommendations for reporting memory attributes on an interconnect

This appendix describes the ARM recommendations for reporting the memory attributes that are assigned by the PE. It contains the following section:

- *ARM recommendations for reporting memory attributes on an interconnect* on page K4-5548.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K4.1 ARM recommendations for reporting memory attributes on an interconnect

The ARM architecture defines the architectural interface between software and the PE hardware. This means the mechanisms by which different memory type and cacheability attributes are presented on an interface to an interconnect fabric such as AMBA® AXI are, strictly, outside the scope of the architecture. This appendix describes an approach for the interface between a PE implementation and an interconnect fabric that ARM strongly recommends, but these recommendations do not form part of the ARMv8 architecture.

### K4.1.1 Effect of microarchitectural choices on memory attributes

Implementations of the ARM architecture permit considerable variability in the presentation of memory attributes on the interconnect fabric, particularly in cases where the PE implementation does not provide optimized support for a memory type. For example, an implementation might treat Write-through locations as Non-cacheable at some level of cache, because functionally this is consistent with the definition of Write-through, but for the particular implementation the performance trade-off does not merit the hardware directly providing Write-through capability. However, in such implementations, the assigned memory attributes are not changed by the microarchitectural choices. The microarchitecture simply implements different ways of handling some memory attributes.

Therefore, ARM strongly recommended that where any or all of the following memory attributes are presented on the interface between a PE and an interconnect fabric, the attributes that are presented are completely consistent with the attributes defined by the translation system:

- The memory type, Normal or Device.
- The Early write acknowledgement attribute.
- The ordering requirements.
- The Shareability.
- The Cacheability, including where practicable, the allocation hints.

#### Effect when the cache is disabled

ARM also strongly recommends that the effects of the caches being disabled, as described in *Cache enabling and disabling* on page D3-1710 for AArch64 and in *Cache enabling and disabling in AArch32 state* on page G3-3955 for AArch32, are reflected on the interconnect by the memory type and attributes used for memory transactions generated while the cache is disabled.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K5
# ARMv8 Changes to the T32 and A32 Instruction Sets

This appendix summarizes the changes that ARMv8 makes to the T32 and A32 instruction sets. It contains the following sections:


- *The A32 and T32 instruction sets* on page K5-5550.
- *Partial deprecation of IT* on page K5-5551.
- *New A32 and T32 Load-Acquire/Store-Release instructions* on page K5-5552.
- *New A32 and T32 scalar floating-point instructions* on page K5-5553.
- *New A32 and T32 Advanced SIMD floating-point instructions* on page K5-5556.
- *New A32 and T32 instructions provided by the Cryptographic Extension* on page K5-5558.
- *New A32 and T32 System instructions* on page K5-5559.
- *CRC32 instructions* on page K5-5561.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K5.1 The A32 and T32 instruction sets

This chapter describes the changes that ARMv8-A makes to the T32 and A32 instruction sets, compared to an ARMv7-A implementation that includes all of the following extensions:

- Multiprocessing Extensions.
- Large Physical Address Extension.
- Virtualization Extensions.
- Security Extensions.
- VFPv4.
- Advanced SIMDv2.

These ARMv8 changes are not affected by whether the ARMv8-A implementation includes either or both of EL2 and EL3.

ARMv8-A obsoletes the A32 `SWP` and `SWPB` instructions.

ARM deprecates any use of the following instructions. The first two of these deprecations are made for performance reasons. In ARMv8-A, privileged software can disable this functionality, see the descriptions of the SCTLR.{CP15BEN, ITD, SED} and HSCTLR.{CP15BEN, ITD, SED} bits:

- The A32 and T32 CP15 memory barriers CP15DSB, CP15ISB, and CP15DMB.
- A subset of T32 `IT` instruction functionality, as described in *Partial deprecation of IT* on page K5-5551.
- The A32 and T32 `SETEND` instruction.

ARMv8-A adds new A32 and T32 instructions to align with some of the features introduced in the A64 instruction set. These are described in:

- *Partial deprecation of IT* on page K5-5551.
- *New A32 and T32 Load-Acquire/Store-Release instructions* on page K5-5552.
- *New A32 and T32 scalar floating-point instructions* on page K5-5553.
- *New A32 and T32 Advanced SIMD floating-point instructions* on page K5-5556.
- *New A32 and T32 instructions provided by the Cryptographic Extension* on page K5-5558.
- *New A32 and T32 System instructions* on page K5-5559.

**Note**

The existing A32 and T32 assembler syntax is unchanged from ARMv7 UAL. Where the syntax term <c> is used in this chapter, it represents a standard ARM condition code. Mnemonics that do not include <c> can not be conditionally executed.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K5.2 Partial deprecation of IT

ARMv8-A deprecates some uses of the T32 IT instruction, for performance reasons. All uses of IT that apply to instructions other than a single subsequent 16-bit instruction from a restricted set are deprecated, as are explicit references to the PC within that single 16-bit instruction. This permits the non-deprecated forms of IT and subsequent instructions to be treated as a single 32-bit conditional instruction. Table K5-1 shows the restricted set of 16-bit instructions that are not deprecated when used in conjunction with IT.

**Table K5-1 Non-deprecated IT 16-bit conditional instructions**

| Permitted 16-bit instructions | Class | Notes |
|---|---|---|
| MOV, MVN | Move | Deprecated when Rm or Rd is the PC. |
| LDR, LDRB, LDRH, LDRSB, LDRSH | Load | Deprecated for PC-relative load literal forms |
| STR, STRB, STRH | Store | - |
| ADD, ADC, RSB, SBC, SUB | Add/Subtract | Deprecated for ADD SP,SP,#imm, SUB SP,SP,#imm, and when Rm, Rdn, or Rdm is the PC |
| CMP, CMN | Compare | Deprecated when Rm or Rn is the PC |
| MUL | Multiply | - |
| ASR, LSL, LSR, ROR | Shift | - |
| AND, BIC, EOR, ORR, TST | Logical | - |
| BX, BLX | Branch to register | Deprecated when Rm is the PC |

The full ARMv7 IT instruction functionality remains available in order to execute legacy T32 code. It is IMPLEMENTATION DEFINED whether an ARMv8 implementation provides an ITD control, that software can use to disable the deprecated uses of the IT instruction. In an implementation that included the ITD control, setting an ITD field to 1 disables the deprecated uses of the IT instruction, making those uses of the IT instruction UNDEFINED. The ITD control fields are:

**HSCTLR.ITD** When EL2 is using AArch32, makes execution of the deprecated uses of the IT UNDEFINED at EL2.

**SCTLR.ITD** When EL1 is using AArch32, makes execution of the deprecated uses of the IT UNDEFINED at EL0 and EL1.

**SCTLR_EL1.ITD**

When EL1 is using AArch64, makes execution of the deprecated uses of the IT UNDEFINED at EL0 when EL0 is using AArch32.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## K5.3 New A32 and T32 Load-Acquire/Store-Release instructions

The new Load-Acquire/Store-Release instructions must be naturally aligned. LDAEXD and STLEXD must be aligned to 8 bytes. An unaligned address causes an alignment fault. For more information about the ordering of Load-Acquire/Store-Release, see *Load-Acquire, Store-Release* on page E2-2378.

### K5.3.1 A32 and T32 Load-Acquire/Store-Release (non-exclusive) instructions

Table K5-2 shows the new A32 and T32 Load-Acquire/Store-Release (non-exclusive) instructions.

**Table K5-2 A32 and T32 Load-Acquire/Store-Release (non-exclusive) instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDA | Load-Acquire Word | *LDA* on page F6-2692 |
| LDAB | Load-Acquire Byte | *LDAB* on page F6-2693 |
| LDAH | Load-Acquire Halfword | *LDAH* on page F6-2702 |
| STL | Store-Release Word | *STL* on page F6-3030 |
| STLB | Store-Release Byte | *STLB* on page F6-3031 |
| STLH | Store-Release Halfword | *STLH* on page F6-3040 |

### K5.3.2 A32 and T32 Load-Acquire/Store-Release Exclusive instructions

Table K5-3 shows the new A32 and T32 Load-Acquire/Store-Release Exclusive instructions.

**Table K5-3 A32 and T32Load-Acquire/Store-Release Exclusive instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| LDAEX | Load-Acquire Exclusive Word | *LDAEX* on page F6-2694 |
| LDAEXB | Load-Acquire Exclusive Byte | *LDAEXB* on page F6-2696 |
| LDAEXD | Load-Acquire Exclusive Double | *LDAEXD* on page F6-2698 |
| LDAEXH | Load-Acquire Exclusive Halfword | *LDAEXH* on page F6-2700 |
| STLEX | Store-Release Exclusive Word | *STLEX* on page F6-3032 |
| STLEXB | Store-Release Exclusive Byte | *STLEXB* on page F6-3034 |
| STLEXD | Store-Release Exclusive Double | *STLEXD* on page F6-3036 |
| STLEXH | Store-Release Exclusive Halfword | *STLEXH* on page F6-3038 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K5.4 New A32 and T32 scalar floating-point instructions

This section describes the new A32 and T32 scalar floating-point instructions. It contains the following subsections:

- *A32 and T32 floating-point conditional select*.
- *A32 and T32 floating-point minimum and maximum numeric*.
- *A32 and T32 floating-point to integer conversion*.
- *A32 and T32 floating-point conversion between half-precision and double-precision* on page K5-5554.
- *A32 and T32 floating-point round to integer* on page K5-5554.

### K5.4.1 A32 and T32 floating-point conditional select

The new `VSEL` instruction conditionally copies one of its two source registers to the destination register. For A32 it provides an alternative to a pair of conditional `VMOV` instructions, while for T32 it compensates for the partial deprecation of `IT` instruction described in *Partial deprecation of IT* on page K5-5551, since it does not require an `IT` prefix.

Table K5-4 shows the A32 and T32 floating-point conditional select instructions

**Table K5-4 A32 and T32 Conditional select**

| Mnemonic | Instruction | See |
|---|---|---|
| `VSEL` | Conditional select | *VSELEQ, VSELGE, VSELGT, VSELVS* on page F7-3661 |

### K5.4.2 A32 and T32 floating-point minimum and maximum numeric

The new `VMAXNM` and `VMINNM` instructions implement the `minNum(x,y)` and `maxNum(x,y)` operations defined by the IEEE754-2008 standard. They return the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to `VFP VMAX` and `VMIN`. These instructions cannot be conditionally executed.

Table K5-5 shows the A32 and T32 floating-point minNum and maxNum instructions.

**Table K5-5 A32 and T32 floating-point minNum and maxNum instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| `VMAXNM` | Single-precision maxNum (scalar) | *VMAXNM* on page F7-3440 |
| `VMINNM` | Double-precision minNum (scalar) | *VMINNM* on page F7-3447 |

### K5.4.3 A32 and T32 floating-point to integer conversion

These new instructions extend the ARMv7 VFP `VCVT` instructions by providing four additional explicit rounding modes. The instruction syntax is `VCVT`*r*, where *r* selects the rounding mode as follows:

`N` Round to Nearest.

`A` Round to Nearest with Ties to Away.

`P` Round towards Plus Infinity.

`M` Round towards Minus Infinity.

The rounding modes are defined by the IEEE 754 standards, see *Floating-point standards, and terminology* on page A1-48.

These instructions cannot be conditionally executed.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Table K5-6 shows the A32 and T32 FP to integer conversion instructions.

**Table K5-6 A32 and T32 floating-point to integer conversion instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| VCVTA | Convert floating-point to integer with Round to Nearest with Ties to Away | *VCVTA (floating-point)* on page F7-3345 |
| VCVTM | Convert floating-point to integer with Round towards Minus Infinity | *VCVTM (floating-point)* on page F7-3352 |
| VCVTN | Convert floating-point to integer with Round to Nearest | *VCVTN (floating-point)* on page F7-3356 |
| VCVTP | Convert floating-point to integer with Round towards Plus Infinity | *VCVTP (floating-point)* on page F7-3360 |

### K5.4.4 A32 and T32 floating-point conversion between half-precision and double-precision

The VFP VCVTT and VCVTB instructions are extended to permit direct conversion between half-precision and double-precision floating-point as a single operation, preventing double rounding errors. The syntax term <y> in Table K5-7 is either T, top half, and B, bottom half.

Table K5-7 shows the A32 and T32 instructions to convert between half-precision and double-precision floating-point values.

**Table K5-7 A32 and T32 floating-point precision conversion**

| Mnemonic | Instruction | See |
|---|---|---|
| VCVTB | Floating-point convert half-precision to double-precision | *VCVTB* on page F7-3347 |
| VCVTT | Floating-point convert double-precision to half-precision | *VCVTT* on page F7-3365 |

### K5.4.5 A32 and T32 floating-point round to integer

The new round to integer instructions round a floating-point value to the nearest integer floating-point value of the same size. The floating-point exceptions that can be raised by these instructions are the Invalid operation, for a signaling NaN input, or Input Denormal, for a denormal input when flush-to-zero mode is enabled. For VRINTX only an Inexact exception can be raised if the result is numeric and does not have the same numerical value as the source. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal floating-point arithmetic.

The instruction syntax is VRINT*r*, where *r* selects the rounding mode as follows:

- N Round to Nearest.
- A Round to Nearest with Ties to Away.
- P Round towards Plus Infinity.
- M Round towards Minus Infinity.
- Z Round towards Zero.
- R FPSCR rounding mode.
- X FPSCR rounding mode, signaling inexactness.

When *r* is R, X, or Z, the round to integer instruction can be conditionally executed. The remaining round to integer instructions must be unconditional.

Table K5-8 on page K5-5555 shows the A32 and T32 round to integer instructions.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K5-8 A32 and T32 floating-point round to integer instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| Round to integer instructions that can be conditionally executed | | |
| VRINTR | Round floating-point to integer using FPSCR rounding mode | *VRINTR* on page F7-3633 |
| VRINTX | Round floating-point to integer using FPSCR rounding mode, signaling inexactness. | *VRINTX (floating-point)* on page F7-3637 |
| VRINTZ | Round floating-point to integer towards Zero | *VRINTZ (floating-point)* on page F7-3641 |
| Round to integer instructions that must be unconditional | | |
| VRINTA | Round floating-point to integer to Nearest with Ties to Away | *VRINTA (floating-point)* on page F7-3619 |
| VRINTM | Round floating-point to integer towards Minus Infinity | *VRINTM (floating-point)* on page F7-3623 |
| VRINTN | Round floating-point to integer to Nearest with Ties to Even | *VRINTN (floating-point)* on page F7-3627 |
| VRINTP | Round floating-point to integer towards Plus Infinity | *VRINTP (floating-point)* on page F7-3631 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K5.5 New A32 and T32 Advanced SIMD floating-point instructions

The AArch32 Advanced SIMD instructions support only single-precision, 32-bit, floating-point data types, with fixed operating modes of Round to Nearest, Default NaN, and Flush-to-Zero. However, they are extended by the addition of the instructions described in the following subsections:

- *A32 and T32 floating-point minimum and maximum numeric*.
- *A32 and T32 floating-point conversion*.
- *A32 and T32 floating-point round to integral*.

### K5.5.1 A32 and T32 floating-point minimum and maximum numeric

Vector forms of the new `VMAXNM` and `VMINNM` instructions are described in *A32 and T32 floating-point minimum and maximum numeric* on page K5-5553.

Table K5-9 shows the A32 and T32 floating-point minNum/maxNum instructions.

**Table K5-9 A32 and T32 floating-point minNum/maxNum instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| `VMAXNM` | Single-precision maxNum (vector) | *VMAXNM* on page F7-3440 |
| `VMINNM` | Double-precision minNum (vector) | *VMINNM* on page F7-3447 |

### K5.5.2 A32 and T32 floating-point conversion

Vector forms of the floating-point to integer conversion instructions are described in *A32 and T32 floating-point to integer conversion* on page K5-5553. The instruction syntax is `VCVT`*r*, where *r* selects the rounding mode as follows:

`N` Round to Nearest.

`A` Round to Nearest with Ties to Away.

`P` Round towards Plus Infinity.

`M` Round towards Minus Infinity.

The rounding modes are defined by the IEEE 754 standards, see *Floating-point standards, and terminology* on page A1-48.

Table K5-10 shows the A32 and T32 floating-point conversion instructions.

**Table K5-10 A32 and T32 floating-point conversion instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| `VCVTA` | Vector Convert floating-point to integer with Round to Nearest with Ties to Away | *VCVTA (Advanced SIMD)* on page F7-3343 |
| `VCVTM` | Vector Convert floating-point to integer with Round towards Minus Infinity | *VCVTM (Advanced SIMD)* on page F7-3350 |
| `VCVTN` | Vector Convert floating-point to integer with Round to Nearest | *VCVTN (Advanced SIMD)* on page F7-3354 |
| `VCVTP` | Vector Convert floating-point to integer with Round towards Plus Infinity | *VCVTP (Advanced SIMD)* on page F7-3358 |

### K5.5.3 A32 and T32 floating-point round to integral

Vector forms of the floating-point rounding instructions are described in *A32 and T32 floating-point round to integer* on page K5-5554. The instruction syntax is `VRINT`*r*, where *r* selects the rounding mode as follows:

`N` Round to Nearest.

`A` Round Nearest with Ties to Away.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

`P` Round towards Plus Infinity.

`M` Round towards Minus Infinity.

`Z` Round towards Zero.

`X` Round to Nearest, signaling inexactness.

The rounding modes are defined by the IEEE 754 standards, see *Floating-point standards, and terminology* on page A1-48.

Table K5-11 shows the A32 and T32 floating-point round to integral instructions.

**Table K5-11 A32 and T32 SIMD floating-point round to integral instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| `VRINTA` | Vector Round floating-point to integer towards Nearest with Ties to Away | *VRINTA (Advanced SIMD)* on page F7-3617 |
| `VRINTM` | Vector Round floating-point to integer towards Minus Infinity | *VRINTM (Advanced SIMD)* on page F7-3621 |
| `VRINTN` | Vector Round floating-point to integer to Nearest | *VRINTN (Advanced SIMD)* on page F7-3625 |
| `VRINTP` | Vector Round floating-point to integer towards Plus Infinity | *VRINTP (Advanced SIMD)* on page F7-3629 |
| `VRINTX` | Vector round floating-point to integer to nearest signalling inexactness | *VRINTX (Advanced SIMD)* on page F7-3635 |
| `VRINTZ` | Vector round floating-point to integer towards Zero | *VRINTZ (Advanced SIMD)* on page F7-3639 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K5.6 New A32 and T32 instructions provided by the Cryptographic Extension

The optional Cryptographic Extension instructions use the SIMD and floating-point register file. For more information see:

- Announcing the Advanced Encryption Standard.
- *The Galois/Counter Mode of Operation*.
- *Announcing the Secure Hash Standard*.

Table K5-12 shows the A32 and T32 Cryptographic Extension instructions.

**Table K5-12 A32 and T32 Cryptographic Extension instructions**

| Mnemonic | Instruction | See |
|---|---|---|
| AESD | AES single round decryption | *AESD* on page F7-3209 |
| AESE | AED single round encryption | *AESE* on page F7-3211 |
| AESIMC | AES inverse mix columns | *AESIMC* on page F7-3213 |
| AESMC | AES mix columns | *AESMC* on page F7-3214 |
| SHA1C | SHA1 hash update accelerator, choose | *SHA1C* on page F7-3224 |
| SHA1M | SHA1 hash update accelerator, majority | *SHA1M* on page F7-3227 |
| SHA1P | SHA1 hash update accelerator, parity | *SHA1P* on page F7-3229 |
| SHA1H | SHA1 hash update accelerator, rotate left by 30 | *SHA1H* on page F7-3226 |
| SHA1SU0 | SHA1 schedule update accelerator, first part | *SHA1SU0* on page F7-3231 |
| SHA1SU1 | SHA1 schedule update accelerator, second part | *SHA1SU1* on page F7-3233 |
| SHA256H | SHA256 hash update accelerator | *SHA256H* on page F7-3235 |
| SHA256H2 | SHA256 hash update accelerator upper part | *SHA256H2* on page F7-3236 |
| SHA256SU0 | SHA256 schedule update accelerator, first part | *SHA256SU0* on page F7-3237 |
| SHA256SU1 | SHA256 schedule update accelerator, second part | *SHA256SU1* on page F7-3239 |
| VMULL | Polynomial multiply long, 64×64 to 128-bit | *VMULL (integer and polynomial)* on page F7-3508 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## K5.7 New A32 and T32 System instructions

The section describes the new System instructions. It contains the following subsections:

- *External Debug*.
- *Barriers and hints*.
- *TLB Maintenance*.

### K5.7.1 External Debug

Table K5-13 shows the new External Debug support instructions.

**Table K5-13 External Debug support instructions**

| Mnemonic | Instructions | Notes |
|---|---|---|
| DCPS1 | Debug switch to EL1, valid in Debug state only | - |
| DCPS2 | Debug switch to EL2, valid in Debug state only | - |
| DCPS3 | Debug switch to EL3, valid in Debug state only | - |
| HLT #uimm6 | Halting mode software breakpoint | Enters Debug state if allowed, with a 6-bit payload in uimm6, otherwise treated as UNDEFINED |

### K5.7.2 Barriers and hints

There are new A32 and T32 barrier options and hint instructions.

Table K5-14 shows the new A32 and T32 barrier instructions.

**Table K5-14 Additional barrier instructions**

| Mnemonic | Notes |
|---|---|
| DMB {ISHLD, OSHLD, NSHLD, LD} | Data Memory Barrier is extended to support the new Load-Load/Store options |
| DSB {ISHLD, OSHLD, NSHLD, LD} | Data Synchronization Barrier is extended to support the new Load-Load/Store options |
| SEVL | Send Event Locally without the requirement to affect other processors, for example to prime a wait-loop that starts with a WFE instructions |

### K5.7.3 TLB Maintenance

TLB maintenance instructions that are only required to apply to the last level translation table walk of the first stage of translation are added to A32 and T32 as shown in Table K5-15. For more information see *Translation Lookaside Buffers (TLBs)* on page G4-4058 and *TLB maintenance requirements* on page G4-4061.

**Table K5-15 Additional A32 and T32 TLB maintenance instructions**

| Mnemonic | Relation to existing A32/T32 operation |
|---|---|
| TLBIMVALIS | Related to the TLBIMVAIS operation |
| TLBIMVAALIS | Related to the TLBIMVAAIS operation |
| TLBIMVALHIS | Related to the TLBIMVAHIS operation |
| TLBIMVAL | Related to the TLBIMVA operation |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K5-15 Additional A32 and T32 TLB maintenance instructions (continued)**

| Mnemonic | Relation to existing A32/T32 operation |
|---|---|
| TLBIMVAAL | Related to the TLBIMVAA operation |
| TLBIMVALH | Related to the TLBIMVAH operation |

A32 and T32 include TLB maintenance instructions that must apply to individual entries from stage 2 TLB structures, that hold IPA to PA translations. These are consistent with the A64 TLBI system instructions described in *New A32 and T32 System instructions* on page K5-5559. The relation between the A32 and T32 instructions and the A64 instructions is shown in Table K5-16.

**Table K5-16 Relation of A32 TLB maintenance instructions to A64 instructions**

| T32 and A32 instruction | Relation to A64 instruction |
|---|---|
| TLBIIPAS2IS | Equivalent to IPAS2E1IS |
| TLBIIPAS2LIS | Equivalent to IPAS2LE1IS<br>Related to existing A32/T32 TLBIIPAS2IS operation |
| TLBIIPAS2 | Equivalent to the A64 IPAS2E1 operation |
| TLBIIPAS2L | Equivalent to IPAS2LE1 operation<br>Related to existing A32/T32 TLBIIPAS2 operation |

**Note**

These new system operations are accessed using the MCR instruction or, if implemented, by an assembler using the SYS mnemonic followed by the TLBI operation name.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K5.8 CRC32 instructions

ARMv8 introduces CRC32 instructions to the T32 and A32 instruction sets. Table K5-17 shows these instructions:

**Table K5-17 CRC32 instructions**

| Mnemonic | Instructions | See |
|---|---|---|
| `CRC32` | CRC32 using polynomial `0x04C11DB7` | *CRC32* on page F6-2665 |
| `CRC32C` | CRC32 using polynomial `0x1EDC6F41` | *CRC32C* on page F6-2667 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K6
# Legacy Instruction Syntax for AArch32 Instruction Sets

This appendix describes the legacy instruction syntax in the ARM instruction sets, and their *Unified Assembler Language* (UAL) equivalents. It contains the following section:


- *Legacy Instruction Syntax* on page K6-5564.




*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K6.1 Legacy Instruction Syntax

Early versions of the ARM Architecture defined an assembly language for A32 (ARM) instructions, and a separate assembly language for T32 (Thumb) instructions. UAL is based on the A32 assembly language, with some changes to the instruction syntax. The appendix describes those changes. The pre-UAL mnemonics are compatible with UAL, and might be supported by an assembler.

The original T32 assembly language is not compatible with UAL, and is not described in the manual.

### K6.1.1 Pre-UAL instruction syntax for the A32 base instructions

Table K6-1 lists the syntax for the A32 base instructions that have changed after UAL was introduced.

**Table K6-1 Pre-UAL instruction syntax for the A32 base instructions**

| Pre-UAL syntax | UAL equivalent | See |
|---|---|---|
| ADC<c>S | ADCS<c> | *ADC, ADCS (immediate)* on page F6-2577, *ADC, ADCS (register)* on page F6-2579, *ADC, ADCS (register-shifted register)* on page F6-2583 |
| ADD<c>S | ADDS<c> | *ADD, ADDS (immediate)* on page F6-2585, *ADD, ADDS (register)* on page F6-2589, *ADD, ADDS (register-shifted register)* on page F6-2593, *ADD, ADDS (SP plus immediate)* on page F6-2595, *ADD, ADDS (SP plus register)* on page F6-2598 |
| AND<c>S | ANDS<c> | *AND, ANDS (immediate)* on page F6-2607, *AND, ANDS (register)* on page F6-2609, *AND, ANDS (register-shifted register)* on page F6-2613 |
| BIC<c>S | BICS<c> | *BIC, BICS (immediate)* on page F6-2630, *BIC, BICS (register)* on page F6-2632, *BIC, BICS (register-shifted register)* on page F6-2635 |
| EOR<c>S | EORS<c> | *EOR, EORS (immediate)* on page F6-2675, *EOR, EORS (register)* on page F6-2677, *EOR, EORS (register-shifted register)* on page F6-2681 |
| LDC<c>L | LDCL<c> | *LDC, LDC2 (immediate)* on page F6-2704, *LDC, LDC2 (literal)* on page F6-2708 |
| LDM<c>IA, LDM<c>FD | LDM<c> | *LDM, LDMIA, LDMFD* on page F6-2711 |
| LDM<c>DA, LDM<c>FA | LDMDA<c> | *LDMDA, LDMFA* on page F6-2718 |
| LDM<c>DB, LDM<c>EA | LDMDB<c> | *LDMDB, LDMEA* on page F6-2720 |
| LDM<c>IB, LDM<c>ED | LDMIB<c> | *LDMIB, LDMED* on page F6-2722 |
| LDR<c>B | LDRB<c> | *LDRB (immediate)* on page F6-2733, *LDRB (literal)* on page F6-2736, *LDRB (register)* on page F6-2738 |
| LDR<c>BT | LDRBT<c> | *LDRBT* on page F6-2740 |
| LDR<c>D | LDRD<c> | *LDRD (immediate)* on page F6-2742, *LDRD (literal)* on page F6-2744, *LDRD (register)* on page F6-2746 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K6-1 Pre-UAL instruction syntax for the A32 base instructions (continued)**

| Pre-UAL syntax | UAL equivalent | See |
|---|---|---|
| LDR<c>H | LDRH<c> | *LDRH (immediate)* on page F6-2756,<br>*LDRH (literal)* on page F6-2759,<br>*LDRH (register)* on page F6-2761 |
| LDR<c>SB | LDRSB<c> | *LDRSB (immediate)* on page F6-2765.<br>*LDRSB (literal)* on page F6-2768,<br>*LDRSB (register)* on page F6-2770 |
| LDR<c>SH | LDRSH<c> | *LDRSH (immediate)* on page F6-2774,<br>*LDRSH (literal)* on page F6-2777,<br>*LDRSH (register)* on page F6-2779 |
| LDR<c>T | LDRT<c> | *LDRT* on page F6-2783 |
| MLA<c>S | MLAS<c> | *MLA, MLAS* on page F6-2806 |
| LSLS <Rd>, <Rn>, #0 | MOVS <Rd>, <Rm> | *MOV, MOVS (immediate)* on page F6-2810,<br>*MOV, MOVS (register)* on page F6-2813 |
| MOV<c>S | MOVS<c> | |
| MUL<c>S | MULS<c> | *MUL, MULS* on page F6-2842 |
| MVN<c>S | MVNS<c> | *MVN, MVNS (immediate)* on page F6-2844,<br>*MVN, MVNS (register)* on page F6-2846,<br>*MVN, MVNS (register-shifted register)* on page F6-2849 |
| ORR<c>S | ORRS<C> | *ORR, ORRS (immediate)* on page F6-2856,<br>*ORR, ORRS (register)* on page F6-2858,<br>*ORR, ORRS (register-shifted register)* on page F6-2862 |
| QADDSUBX | QASX | *QASX* on page F6-2893 |
| QSUBADDX | QSAX | *QSAX* on page F6-2899 |
| RSB<c>S | RSBS<c> | *RSB, RSBS (immediate)* on page F6-2929,<br>*RSB, RSBS (register)* on page F6-2932,<br>*RSB, RSBS (register-shifted register)* on page F6-2935 |
| RSC<c>S | RSCS<c> | *RSC, RSCS (immediate)* on page F6-2937,<br>*RSC, RSCS (register)* on page F6-2939,<br>*RSC, RSCS (register-shifted register)* on page F6-2941 |
| SADDSUBX | SASX | *SASX* on page F6-2947 |
| SBC<c>S | SBCS<c> | *SBC, SBCS (immediate)* on page F6-2949,<br>SBC, SBCS (register),<br>*SBC, SBCS (register-shifted register)* on page F6-2954 |
| SHADDSUBX | SHASX | *SHASX* on page F6-2971 |
| SHSUBADDX | SHSAX | *SHSAX* on page F6-2973 |
| SMI<c> | SMC<c> | *SMC* on page F6-2979 |
| SMLAL<c>S | SMLALS<c> | *SMLAL, SMLALS* on page F6-2985 |
| SMULL<c>S | SMULLS<c> | *SMULL, SMULLS* on page F6-3007 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K6-1 Pre-UAL instruction syntax for the A32 base instructions (continued)**

| Pre-UAL syntax | UAL equivalent | See |
|---|---|---|
| SSUBADDX<c> | SSAX<c> | *SSAX* on page F6-3020 |
| STC<c>L | STCL<c> | *STC, STC2* on page F6-3026 |
| STM<c>EA, STM<c>IA | STM<c> | *STM, STMIA, STMEA* on page F6-3041 |
| STM<c>DA, STM<c>ED | STMDA<c> | *STMDA, STMED* on page F6-3045 |
| STM<c>DB, STM<c>FD | STMDB<c> | *STMDB, STMFD* on page F6-3047 |
| STM<c>IB, STM<c>FA | STMIB<c> | *STMIB, STMFA* on page F6-3049 |
| STR<c>B | STRB<c> | *STRB (immediate)* on page F6-3058, *STRB (register)* on page F6-3061 |
| STR<c>BT | STRBT<c> | *STRBT* on page F6-3063 |
| STR<c>D | STRD<c> | *STRD (immediate)* on page F6-3065, *STRD (register)* on page F6-3067 |
| STR<c>H | STRH<c> | *STRH (immediate)* on page F6-3077, *STRH (register)* on page F6-3080 |
| STR<c>T | STRT<c> | *STRT* on page F6-3084 |
| SUB<c>S | SUBS<c> | *SUB, SUBS (immediate)* on page F6-3088, *SUB, SUBS (register)* on page F6-3092, *SUB, SUBS (register-shifted register)* on page F6-3095, *SUB, SUBS (SP minus immediate)* on page F6-3097, *SUB, SUBS (SP minus register)* on page F6-3100 |
| SWI | SVC | *SVC* on page F6-3103 |
| UADDSUBX | UASX | *UASX* on page F6-3132 |
| UHADDSUBX | UHASX | *UHASX* on page F6-3144 |
| UHSUBADDX | UHSAX | *UHSAX* on page F6-3146 |
| UMLAL<c>S | UMLALS<c> | *UMLAL, UMLALS* on page F6-3154 |
| UMULL<c>S | UMULLS<c> | *UMULL, UMULLS* on page F6-3156 |
| UQADDSUBX | UQASX | *UQASX* on page F6-3162 |
| UQSUBADDX | UQSAX | *UQSAX* on page F6-3164 |
| USUBADDX | USAX | *USAX* on page F6-3178 |
| UXT8 | UXTB | *UXTB* on page F6-3190 |
| UXT16 | UXTH | *UXTH* on page F6-3194 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K6.1.2 Pre-UAL instruction syntax for the A32 floating-point instructions

Table K6-2 lists the syntax for A32 floating-point instructions that have changed after UAL was introduced.

**Table K6-2 Pre-UAL instruction syntax for A32 floating-point instructions**

| Pre-UAL syntax | UAL equivalent | See |
|---|---|---|
| FABSD | VABS.F64 | *VABS* on page F7-3251 |
| FABSS | VABS.F32 | |
| FADDD | VADD.F64 | *VADD (floating-point)* on page F7-3262 |
| FADDS | VADD.F32 | |
| FCMPEZD | VCMPE.F64 | *VCMPE* on page F7-3321 |
| FCMPEZS | VCMPE.F32 | |
| FCMPZD | VCMP.F64 | *VCMP* on page F7-3318, |
| FCMPZS | VCMP.F32 | |
| FCONSTD <Dd>, #<imm8> | VMOV.F64 <Dd>, #<fpimm> | *VMOV (immediate)* on page F7-3476 |
| FCONSTS <Sd>, #<imm8> | VMOV.F32 <Sd>, #<fpimm> | For more information, see *FCONST* on page K6-5570. |
| FCPYD | VMOV.F64 | *VMOV (register)* on page F7-3479 |
| FCPYS | VMOV.F32 | |
| FCVTDS | VCVT.F64.F32 | *VCVT (between double-precision and single-precision)* on page F7-3326 |
| FCVTSD | VCVT.F32.F64 | |
| FDIVD | VDIV.F64 | *VDIV* on page F7-3368 |
| FDIVS | VDIV.F32 | |
| FLDD | VLDR.F64 | *VLDR* on page F7-3433 |
| FLDMD, FLDMIAD | VLDM.F64 | *VLDM, VLDMDB, VLDMIA* on page F7-3430 |
| FLDMS | VLDM.F32 | |
| FLDS | VLDR.F32 | *VLDR* on page F7-3433 |
| FMACD | VMLA.F64 | *VMLA (floating-point)* on page F7-3450 |
| FMACS | VMLA.F32 | |
| FMDHR <Dd>, <Rt> | VMOV <Dd[1]>, <Rt> | *VMOV (general-purpose register to scalar)* on page F7-3483 |
| FMDLR <Dd>, <Rt> | VMOV <Dd[0]>, <Rt> | |
| FMDRR | VMOV | *VMOV (between two general-purpose registers and a doubleword floating-point register)* on page F7-3474 |
| FMRDH <Rt>, <Dd> | VMOV <Rt>, <Dd[1]> | *VMOV (scalar to general-purpose register)* on page F7-3487 |
| FMRDL <Rt>, <Dd> | VMOV <Rt>, <Dd[0]> | |
| FMRRD | VMOV | *VMOV (between two general-purpose registers and a doubleword floating-point register)* on page F7-3474 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K6-2 Pre-UAL instruction syntax for A32 floating-point instructions (continued)**

| Pre-UAL syntax | UAL equivalent | See |
|---|---|---|
| FMRRS | VMOV | *VMOV (between two general-purpose registers and two single-precision registers)* on page F7-3489 |
| FMRS | VMOV | *VMOV (between general-purpose register and single-precision register)* on page F7-3485 |
| FMRX | VMRS | *VMRS* on page F7-3495 |
| FMSCD | VNMLS.F64 | *VNMLS* on page F7-3521 |
| FMSCS | VNMLS.F32 | |
| FMSR | VMOV | *VMOV (between general-purpose register and single-precision register)* on page F7-3485 |
| FMSRR | VMOV | *VMOV (between two general-purpose registers and two single-precision registers)* on page F7-3489 |
| FMSTAT | VMRS APSR_nzcv, FPSCR | *VMRS* on page F7-3495 |
| FMULD | VMUL.F64 | *VMUL (floating-point)* on page F7-3499 |
| FMULS | VMUL.F32 | |
| FMXR | VMSR | *VMSR* on page F7-3497 |
| FNEGD | VNEG.F64 | *VNEG* on page F7-3516 |
| FNEGS | VNEG.F32 | |
| FNMACD | VMLS.F64 | *VNMLS* on page F7-3521 |
| FNMACS | VMLS.F32 | |
| FNMSCD | VNMLA.F64 | *VNMLA* on page F7-3519 |
| FNMSCS | VNMLA.F32 | |
| FNMULD | VNMUL.F64 | *VNMUL* on page F7-3523 |
| FNMULS | VNMUL.F32 | |
| FSHTOD | VCVT.F64.S16 | *VCVT (between floating-point and fixed-point, floating-point)* on page F7-3340 |
| FSHTOS | VCVT.F32.S16 | |
| FSITOD | VCVT.F64.S32 | *VCVT (between floating-point and integer, Advanced SIMD)* on page F7-3330, *VCVTR* on page F7-3362 |
| FSITOS | VCVT.F32.S32 | |
| FSLTOD | VCVT.F64.S32 | *VCVT (between floating-point and fixed-point, floating-point)* on page F7-3340 |
| FSLTOS | VCVT.F32.S32 | |
| FSQRTD | VSQRT.F64 | *VSQRT* on page F7-3681 |
| FSQRTS | VSQRT.F32 | |
| FSTD | VSTR | *VSTR* on page F7-3714 |
| FSTMD, FSTMIAS | VSTM.F64 | *VSTM, VSTMDB, VSTMIA* on page F7-3711 |
| FSTMS | VSTM.F32 | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K6-2 Pre-UAL instruction syntax for A32 floating-point instructions (continued)**

| Pre-UAL syntax | UAL equivalent | See |
|---|---|---|
| FSTS | VSTR | *VSTR* on page F7-3714 |
| FSUBD | VSUB.F64 | *VSUB (floating-point)* on page F7-3716 |
| FSUBS | VSUB.F32 | |
| FTOSHD | VCVT.S16.F64 | *VCVT (between floating-point and fixed-point, floating-point)* on page F7-3340 |
| FTOSHS | VCVT.S16.F23 | |
| FTOSID | VCVT.S32.F64 | *VCVT (between floating-point and integer, Advanced SIMD)* on page F7-3330 |
| FTOSIS | VCVT.S32.F32 | |
| FTOSIZD | VCVTR.S32.F64 | *VCVTR* on page F7-3362 |
| FTOSIZS | VCVTR.S32.F32 | |
| FTOSLD | VCVT.S32.F64 | *VCVT (between floating-point and fixed-point, floating-point)* on page F7-3340 |
| FTOSLS | VCVT.S32.F32 | |
| FTOUHD | VCVT.U16.F64 | |
| FTOUHS | VCVT.U16.F32 | |
| FTOUID | VCVT.U32.F64 | *VCVT (between floating-point and integer, Advanced SIMD)* on page F7-3330 |
| FTOUIS | VCVT.U32.F32 | |
| FTOUIZD | VCVTR.U32.F64 | *VCVTR* on page F7-3362 |
| FTOUIZS | VCVTR.U32.F32 | |
| FTOULD | VCVT.U32.F64 | *VCVT (between floating-point and fixed-point, floating-point)* on page F7-3340 |
| FTOULS | VCVT.U32.F32 | |
| FUHTOD, | VCVT.F64.U16 | |
| FUHTOS | VCVT.F64.U16 | |
| FUITOD | VCVT.F64.U32 | *VCVT (between floating-point and integer, Advanced SIMD)* on page F7-3330 |
| FUITOS | VCVT.F32.U32 | |
| FULTOD | VCVT.F64.U32 | *VCVT (between floating-point and fixed-point, floating-point)* on page F7-3340 |
| FULTOS | VCVT.F32.U32 | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K6.1.3 FCONST

The syntax of FCONST is

```
FCONST<dest>{<c>} <Fd>, #<imm8>
```

where:

<dest> Specifies the destination data type. It must be one of:

- S . Single-precision floating-point.
- D . Double-precision floating-point.

<c> This is an optional field. It specifies the condition under which the instruction is executed. See *Conditional execution* on page F2-2447 for the range of available conditions and their encoding. If <c> is omitted, it defaults to *always* (AL).

<Fd> Specifies the destination register. It must be one of:

- <Dd> 64-bit name of the SIMD&FP destination register.
- <Sd> 32-bit name of the SMID&FP destination register.

<imm8> Specifies the immediate value used to generate the floating-point constant.

`FCONSTD{<c>} <Dd>, #<imm8>` maps to `VMOV.F64 <Dd>, #<fpimm>`

`FCONSTS{<c>} <Sd>, #<imm8>` maps to `VMOV.F32 <Sd>, #<fpimm>`


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K7
# Address translation examples

This appendix gives examples of address translations using the translation regimes described in Chapter D4 *The AArch64 Virtual Memory System Architecture* and Chapter G4 *The AArch32 Virtual Memory System Architecture*. It contains the following sections:


- *AArch64 Address translation examples* on page K7-5572.
- *AArch32 Address translation examples* on page K7-5585.


**Note**

This chapter gives examples of translation table lookups for the ARMv8 address translation stages. It does not define any part of the address translation mechanism. If any information in this appendix appears to contradict the information in Chapter D4 *The AArch64 Virtual Memory System Architecture* or Chapter G4 *The AArch32 Virtual Memory System Architecture* then the information in Chapter D4 or Chapter G4 must be taken as the definition of the required behavior.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K7.1 AArch64 Address translation examples

Figure D4-2 on page D4-1746 shows the VMSAv8 address translation stages that are controlled by an Exception level that is using AArch64. *The VMSAv8-64 address translation system* on page D4-1745 describes the VMSAv8-64 address translation scheme. This section gives examples of the use of that scheme, for common translation requirements.

*System registers relevant to MMU operation* on page D4-1750 specifies the relevant registers, including the TCR and TTBR, or TTBRs, for each stage of address translation.

For any stage of translation, a TCR.T*n*SZ field indicates the supported input address size. For a stage of address translation controlled from an Exception level using AArch64, the supported input address size is $2^{(64-T\mathit{n}SZ)}$.

This section describes:

- Performing the initial lookup, for an address for which the initial lookup is either:
  - At the highest lookup level used for the appropriate translation granule size.
  - Because of the concatenation of translation tables at the initial lookup level, one level down from the highest level used for the translation granule size.

  These descriptions take account of the following cases:
  - The IA size is smaller than the largest size for the translation level, see *Reduced IA width* on page D4-1758.
  - For a stage 2 translation, translation tables are concatenated, to move the initial lookup level down by one level, see *Concatenated translation tables* on page D4-1759.

  For examples of performing the initial lookup, see *Examples of performing the initial lookup*.
- The full translation flow for resolving a page of memory. These examples describe resolving the largest IA size supported by the initial lookup level. For these examples, see *Full translation flows for VMSAv8-64 address translation* on page K7-5579.

### K7.1.1 Examples of performing the initial lookup

The address ranges used for the initial translation table lookup depend on the translation granule, as described in:

- *Performing the initial lookup using the 4KB translation granule*.
- *Performing the initial lookup using the 16KB granule* on page K7-5575.
- *Performing the initial lookup using the 64KB translation granule* on page K7-5577.

#### Performing the initial lookup using the 4KB translation granule

This subsection describes examples of the initial lookup when using the 4KB translation granule that Table D4-12 on page D4-1764 shows as starting at level 0 or at level 1. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 1. This means that it gives specific examples of the mechanisms described in *The VMSAv8-64 address translation system* on page D4-1745.

**Note**

For stage 2 translations, the same principles apply to an initial lookup that Table D4-12 on page D4-1764 shows as starting at level 1. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 2.

The following subsections describe these examples of the initial lookup:

- *Initial lookup at level 0, 4KB translation granule* on page K7-5573.
- *Initial lookup at level 1, 4KB translation granule* on page K7-5573.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see *Overview of stage 2 translations, 4KB granule* on page D4-1763.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### *Initial lookup at level 0, 4KB translation granule*

This subsection describes initial lookups with an input address width of ($n$+1) bits, meaning the input address is IA[$n$:0]. As Table D4-12 on page D4-1764 shows, a stage 1 or stage 2 initial lookup at level 0 is required when $39 \le n \le 47$. For these lookups:

- TTBR[47:($n$-35)] specify the translation table base address.
- Bits[$n$:39] of the input address are bits[($n$-36):3] of the descriptor offset in the translation table.

**Note**

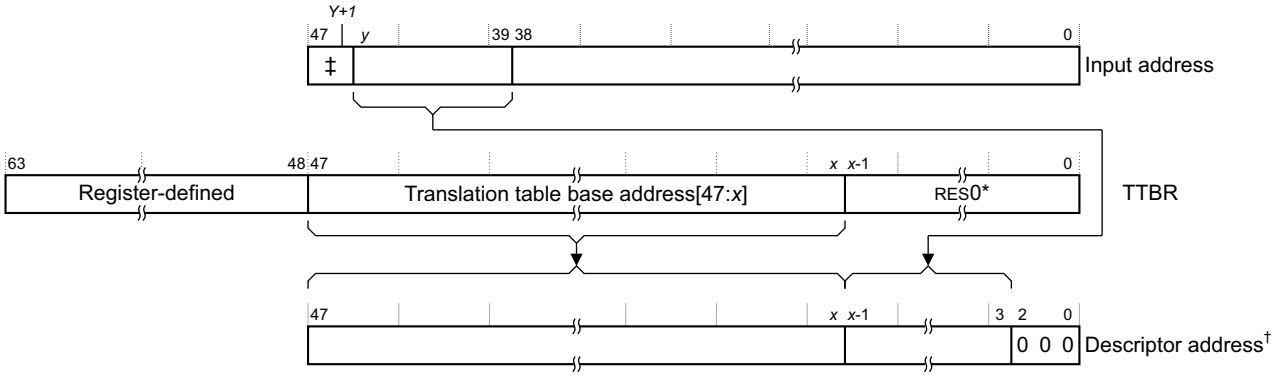
**This means that, when the input address width is less than 48 bits**

- The size of the translation table is reduced.
- More low-order bits of the TTBR are required to specify the translation table base address.
- Fewer input address bit are used to specify the descriptor offset in the translation table.

For example, if the input address width is 46 bits:

- The translation table size is 1KB,
- TTBR bits[47:10] specify the translation table base address.
- Input address bits[45:39] specify bits[9:3] of the descriptor offset.

Figure K7-1 shows this lookup.




Supported input address range is IA[$y$:0], $4 \le x \le 12$, $y = x + 35$. When $y$ is 47 the field marked ‡ is absent.
† For a Non-secure EL1&0 stage 1 translation, the IPA of the descriptor. Otherwise, the PA of the descriptor.
* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-1 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 0**

### *Initial lookup at level 1, 4KB translation granule*

This subsection describes initial lookups with an input address width of ($n$+1) bits, meaning the input address is IA[$n$:0].

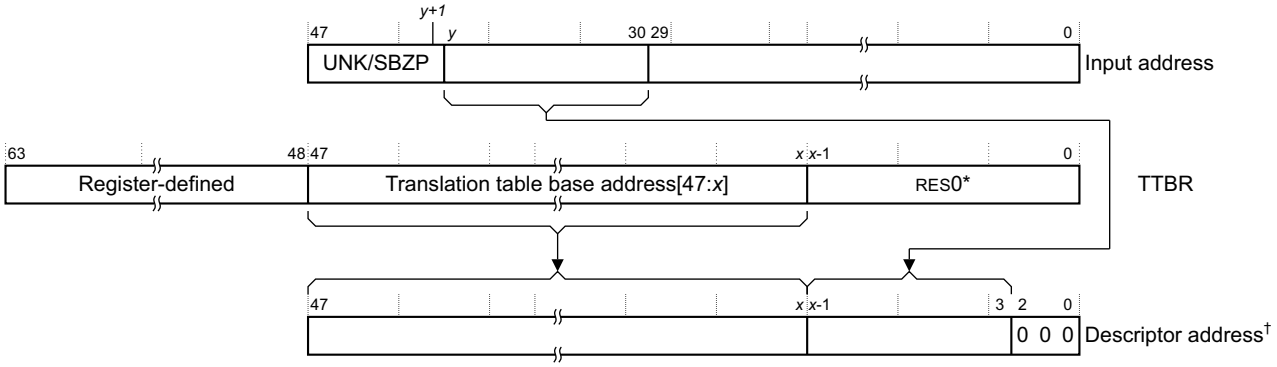
**For a stage 1 or stage 2 initial lookup at level 1, without use of concatenated translation tables**

As Table D4-12 on page D4-1764 shows, this applies to IA[$n$:0], where $30 \le n \le 38$. For these lookups:

- There is a single translation table at this level.
- TTBR[47:($n$-26)] specify the translation table base address.
- Bits[$n$:30] of the input address are bits[($n$-27):3] of the descriptor offset in the translation table.

Figure K7-2 on page K7-5574 shows this lookup.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta




Supported input address range is IA[*y*:0], 4 ≤ *x* ≤ 12, *y* = *x* + 26.

† For a Non-secure EL1&0 stage 1 translation, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

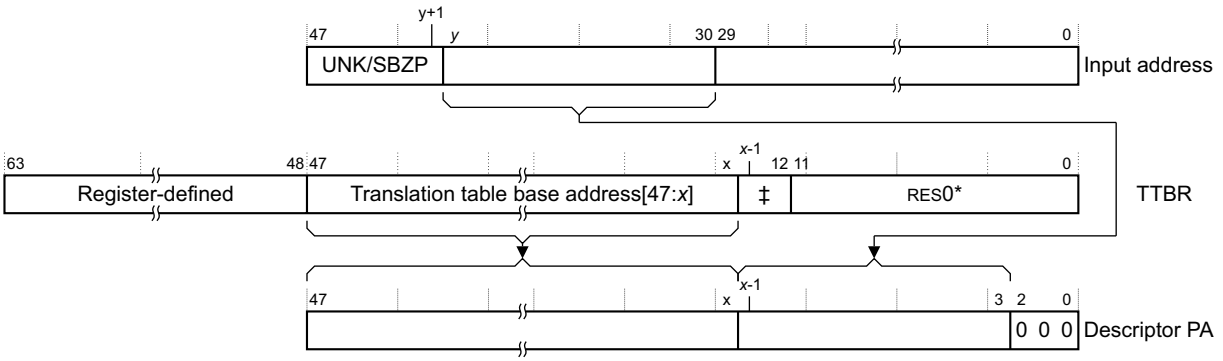
**Figure K7-2 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 1, without concatenation**

**For a stage 2 initial lookup at level 1, with concatenated translation tables**

As Table D4-12 on page D4-1764 shows, this applies to IA[*n*:0], where 39 ≤ *n* ≤ 42. For these lookups:

- There are $2^{(n-38)}$ concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2^{(n-38)}$×4KB. This means TTBR[(*n*-27):12] must be zero.
- TTBR[47:(*n*-26)] specify the base address of the block of concatenated translation tables.
- Bits[*n*:30] of the input address are bits[(*n*-27):3] of the descriptor offset from the base address of the block of concatenated translation tables.

Figure K7-3 shows this lookup.




Supported input address range is IPA[*y*:0], 4 ≤ *x* ≤ 16, *y* = *x* + 26. The field marked ‡ must be zero.

* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-3 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 1, with concatenation**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## Performing the initial lookup using the 16KB granule

This subsection describes examples of the initial lookup when using the 16KB translation granule that Table D4-15 on page D4-1767 shows as starting at level 0 or at level 1. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 1. This means that it gives specific examples of the mechanisms described in *The VMSAv8-64 address translation system* on page D4-1745.

**Note**

For stage 2 translations, the same principles apply to an initial lookup that Table D4-15 on page D4-1767 shows as starting at level 1. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 2.

The following subsections describe these examples of the initial lookup:

- *Initial lookup at level 0, 16KB translation granule*.
- *Initial lookup at level 1, 16KB translation granule* on page K7-5576.

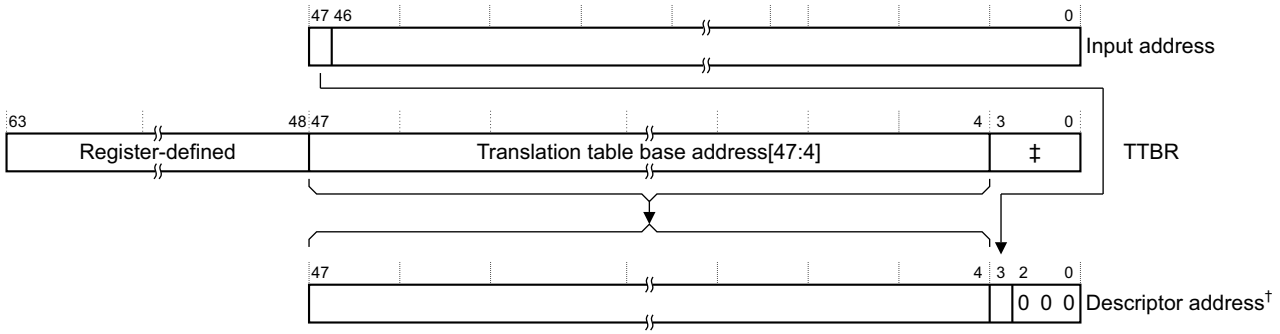
In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see *Overview of stage 2 translations, 16KB granule* on page D4-1767.

### *Initial lookup at level 0, 16KB translation granule*

This subsection describes initial lookups with an input address width of ($n$+1) bits, meaning the input address is IA[$n$:0]. As Table D4-14 on page D4-1766 shows, the only case where an address translation using the 16KB granule starts at level 0 is a stage 1 translation of a 48-bit input address, IA[47:0]. For this lookup:

- The required translation table has only two entries, meaning its size is 16bytes, and it must be aligned to 16 bytes.
- TTBR[47:4] specify the translation table base address.
- Bit[47] of the input address is bits[3] of the descriptor offset in the translation table.

Figure K7-4 shows this lookup.




Supported input address range is IA[47:0]. The field marked ‡ is RES0*.
† For a Non-secure EL1&0 stage 1 translation, the IPA of the descriptor. Otherwise, the PA of the descriptor.
* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-4 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 0**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### *Initial lookup at level 1, 16KB translation granule*

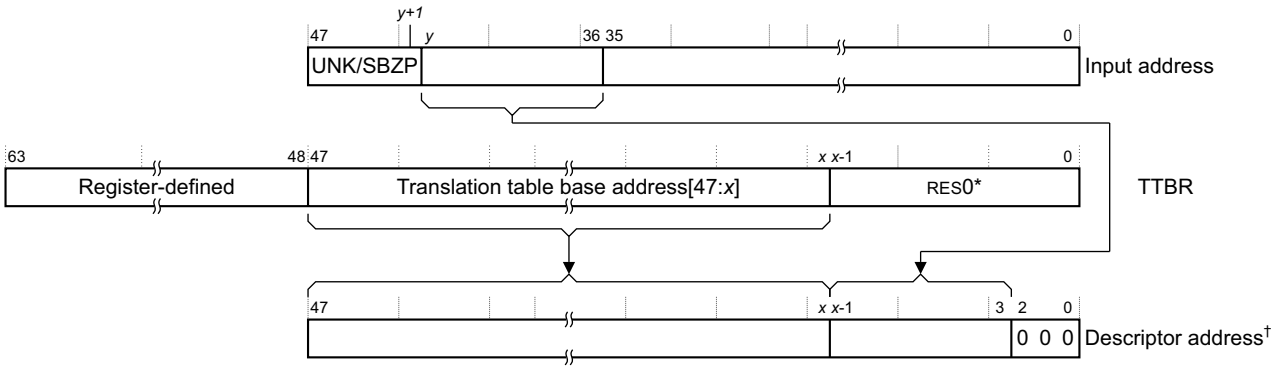
This subsection describes initial lookups with an input address width of ($n$+1) bits, meaning the input address is IA[$n$:0].

**For a stage 1 or stage 2 initial lookup at level 1, without use of concatenated translation tables**

As Table D4-15 on page D4-1767 shows, this applies to IA[$n$:0], where $36 \le n \le 46$. For these lookups:

- There is a single translation table at this level.
- TTBR[47:($n$-32)] specify the translation table base address.
- Bits[$n$:36] of the input address are bits[($n$-33):3] of the descriptor offset in the translation table.

Figure K7-5 shows this lookup.




Supported input address range is IA[$y$:0], $4 \le x \le 14$, $y = x + 32$.

† For a Non-secure EL1&0 stage 1 translation, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-5 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 1, without concatenation**

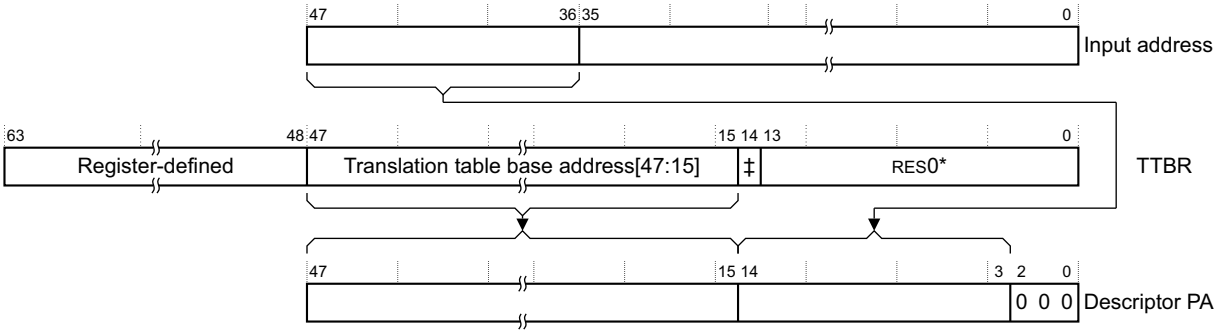
**For a stage 2 initial lookup at level 1, with concatenated translation tables**

As Table D4-15 on page D4-1767 shows, the only case where an address translation using the 16KB granule starts at level 1 because of concatenation of translation tables is a stage 2 translation of a 48-bit input address, IA[47:0]. For this lookup:

- There are two concatenated translation tables at this level.
- These concatenated translation tables must be aligned to 2×16KB. This means TTBR[14] must be zero.
- TTBR[47:15] specify the base address of the block of two concatenated translation tables.
- Bits[47:36] of the input address are bits[14:3] of the descriptor offset from the base address of the block of concatenated translation tables.

Figure K7-6 on page K7-5577 shows this lookup.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*




Supported input address range is IPA[47:0]. The bit marked ‡ must be zero.
* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-6 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 1, with concatenation**

## Performing the initial lookup using the 64KB translation granule

This subsection describes examples of the initial lookup when using the 64KB translation granule that Table D4-18 on page D4-1770 shows as starting at level 1 or at level 2. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 2. This means that it gives specific examples of the mechanisms described in *The VMSAv8-64 address translation system* on page D4-1745.

**Note**

For stage 2 translations, the same principles apply to an initial lookup that Table D4-18 on page D4-1770 shows as starting at level 2. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 3.

The following subsections describe these examples of the initial lookup:

- *Initial lookup at level 1, 64KB translation granule*.
- *Initial lookup at level 2, 64KB translation granule* on page K7-5578.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see *Overview of stage 2 translations, 64KB granule* on page D4-1770.

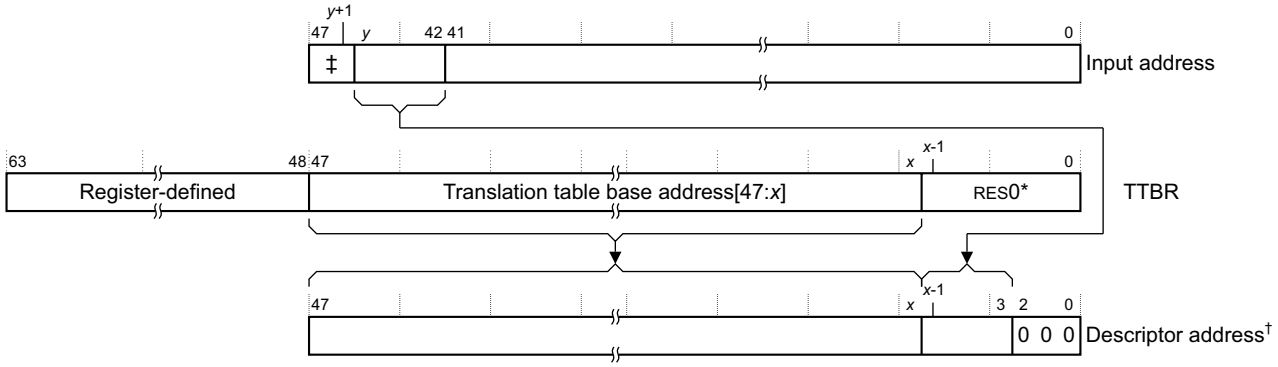
### *Initial lookup at level 1, 64KB translation granule*

This subsection describes initial lookups with an input address width of ($n$+1) bits, meaning the input address is IA[$n$:0]. As Table D4-18 on page D4-1770 shows, a stage 1 or stage 2 initial lookup at level 1 is required when $42 \le n \le 47$. For these lookups:

- The size of the translation table is $2^{(n-39)}$ bytes. This means the size of the translation table, at this level, is always less than the granule size. The address of this translation table must align to the size of the table.
- Bits[$n$:42] of the input address are bits[($n$-39):3] of the descriptor offset in the translation table.
- Bits[47:($n$-38)] of the TTBR specify the translation table base address.

Figure K7-7 on page K7-5578 shows this lookup.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*




Supported input address range is IA[*y*:0], 4 ≤ *x* ≤ 9, *y* = *x* + 38. When *y* is 47 the field marked ‡ is absent.

† For a Non-secure EL1&0 stage 1 translation, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-7 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 1**

### *Initial lookup at level 2, 64KB translation granule*

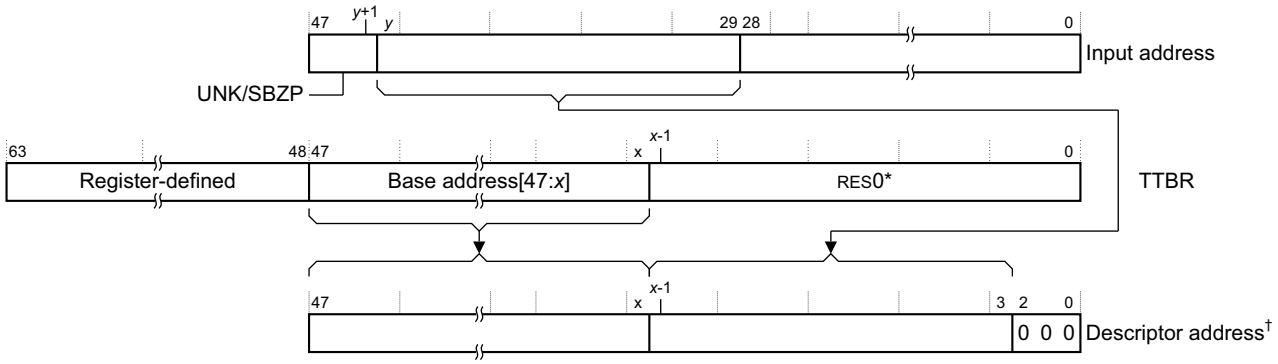
This subsection describes initial lookups with an input address width of (*n*+1) bits, meaning the input address is IA[*n*:0].

**For a stage 1 or stage 2 initial lookup at level 2, without the use of concatenated translation tables**

As Table D4-18 on page D4-1770 shows, this applies to IA[*n*:0], where 29 ≤ *n* ≤ 41. For these lookups:

- There is a single translation table at this level.
- TTBR[47:(*n*-25)] of the specify the translation table base address.
- Bits[*n*:29] of the input address are bits[(*n*-26):3] of the descriptor offset in the translation table.

Figure K7-8 shows this lookup.




Supported input address range is IA[*y*:0]. 4 ≤ *x* ≤ 16, *y* = *x* + 25.

† For a Non-secure EL1&0 stage 1 translation, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-8 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, without concatenation**

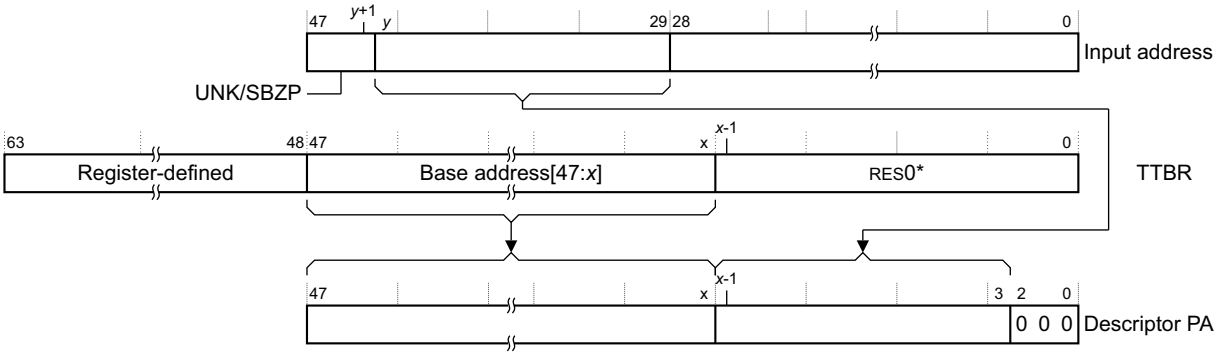

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**For a stage 2 initial lookup at level 2, with concatenated translation tables**

As Table D4-18 on page D4-1770 shows, this applies to IA[$n$:0], where $42 \le n \le 45$. For these lookups:

- There are $2^{(m-41)}$ concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2^{(m-41)} \times 64$KB. This means TTBR[($n$-26):16] must be zero.
- TTBR[47:($n$-25)] specify the base address of the block of translation tables.
- Bits[$n$:42] of the input address are bits[($n$-26):16] of the descriptor offset from the base address of the block of translation tables.

Figure K7-9 shows this lookup.




Supported input address range is IPA[$y$:0], $4 \le x \le 20$, $y = x + 25$.

* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-9 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, with concatenation**

## K7.1.2 Full translation flows for VMSAv8-64 address translation

In a translation table walk, only the first lookup uses the translation table base address from the appropriate TTBR. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

This section describes example full translation flows, from the initial lookup to the address of a memory page. The example flows:

- Resolve the maximum-sized IA range supported by the initial lookup level.
- Do not have any concatenation of translation tables.
- Cover only the 4KB and the 64KB translation granules.

*Examples of performing the initial lookup* on page K7-5572 described how either reducing the IA range or concatenating translation tables affects the initial lookup.

**Note**

Reducing the IA range or concatenating translation tables affects only the initial lookup.

The following sections describe full VMSAv8-64 translation flows, down to an entry for a memory page:

- *The address and properties fields shown in the translation flows* on page K7-5580.
- *Full translation flow using the 4KB granule and starting at level 0* on page K7-5581.
- *Full translation flow using the 4KB granule and starting at level 1* on page K7-5582.
- *Full translation flow using the 64KB granule and starting at level 1* on page K7-5583.
- *Full translation flow using the 64KB granule and starting at level 2* on page K7-5584.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### The address and properties fields shown in the translation flows

For the Non-secure EL1&0 stage 1 translation:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the block or page.

In these cases, an EL1&0 stage 2 translation is performed to translate the IPA to the required PA.

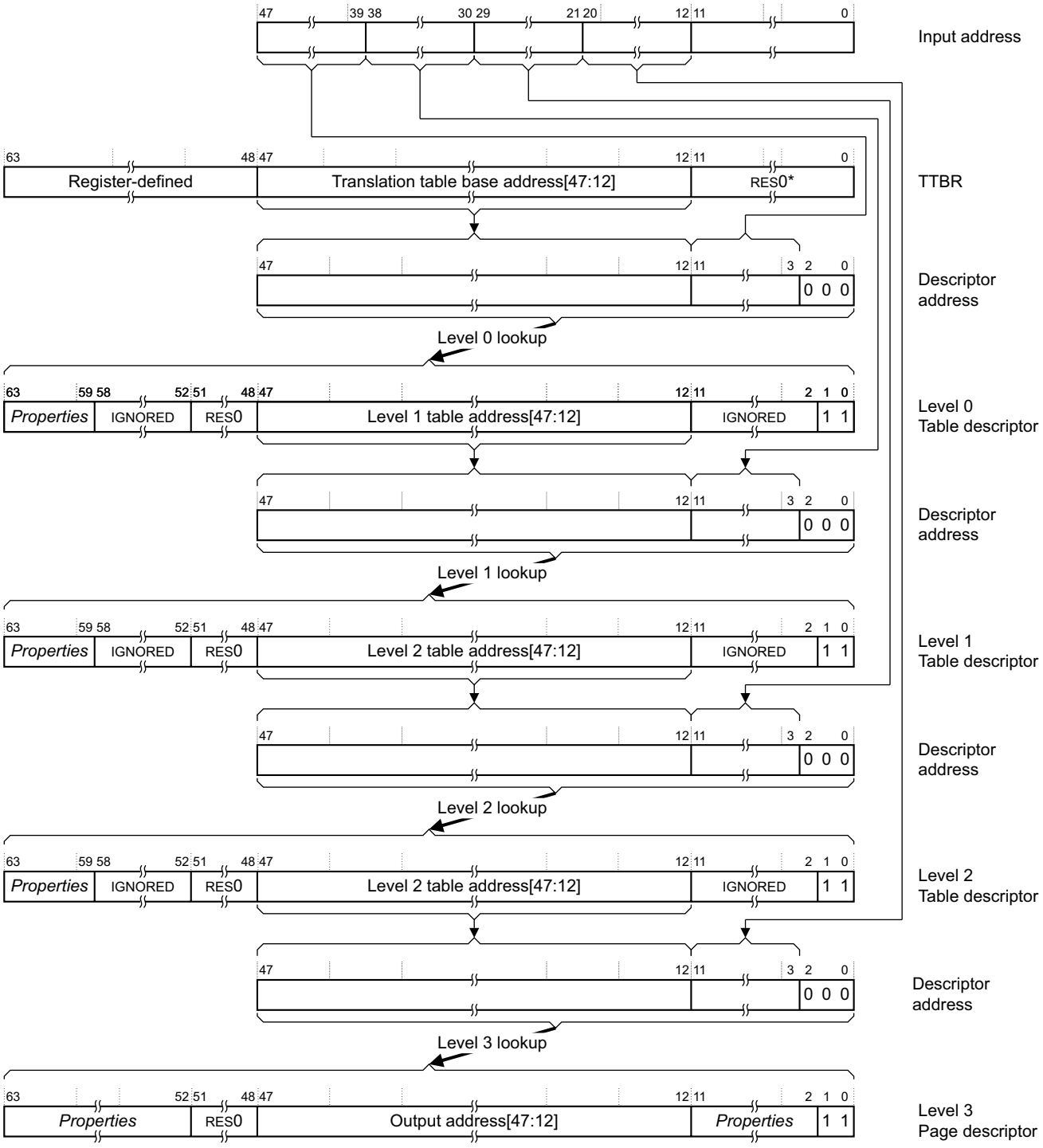
For all other translations, the final output address is the PA of the block or page, and any descriptor address is the PA of the descriptor.

*Properties* indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information see *Memory attribute fields in the VMSAv8-64 translation table format descriptors* on page D4-1803.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Full translation flow using the 4KB granule and starting at level 0

Figure K7-10 shows the complete translation flow for a stage 1 translation table walk for a 48-bit input address. This lookup must start with a level 0 lookup. For more information about the fields shown in the figure see *The address and properties fields shown in the translation flows* on page K7-5580.




For details of *Properties* fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-10 Complete stage 1 translation of a 48-bit address using the 4KB translation granule**

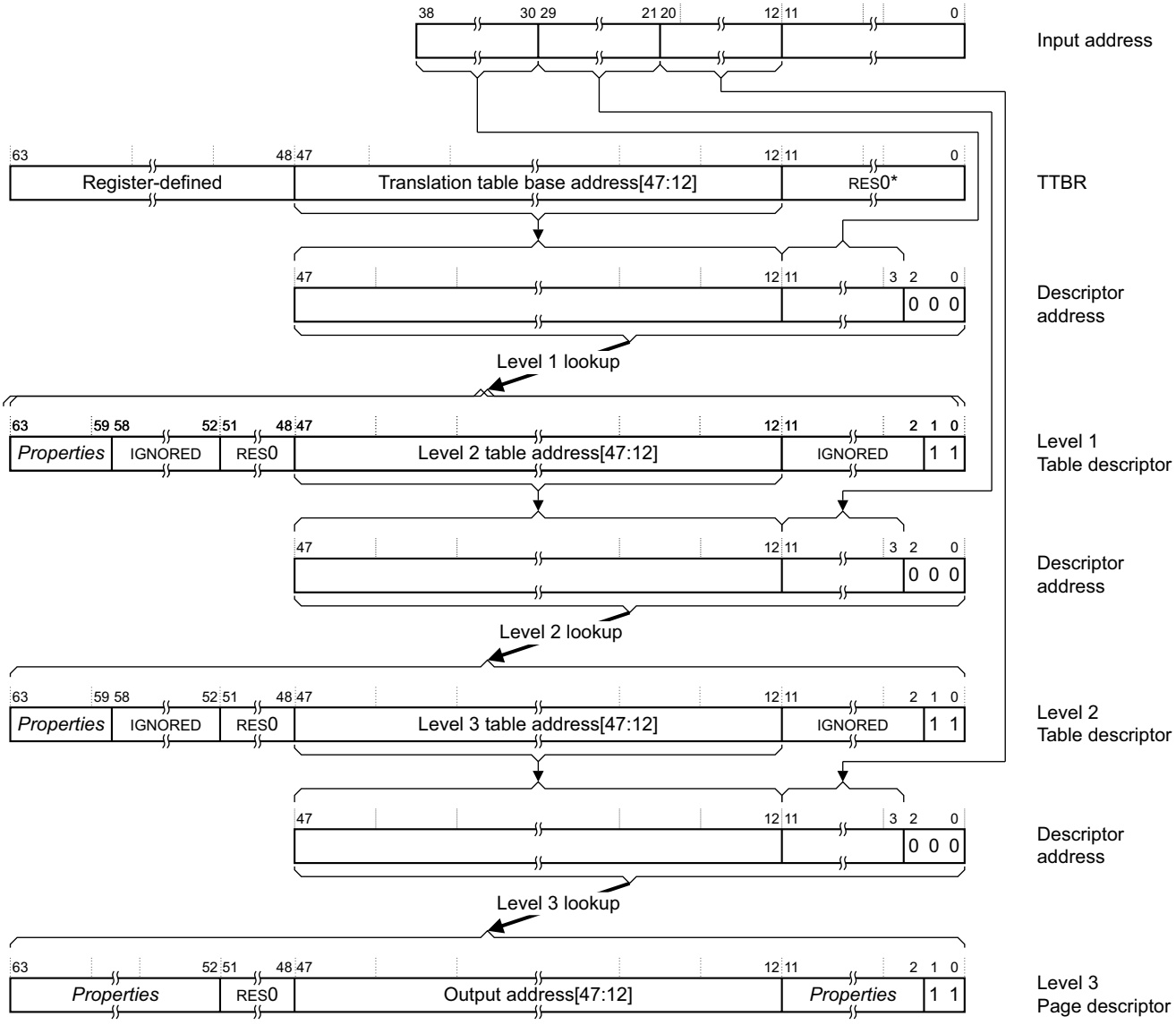

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

If the level 1 lookup or level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K7-10 on page K7-5581 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

### Full translation flow using the 4KB granule and starting at level 1

Figure K7-11 shows the complete translation flow for a stage 1 translation table walk for a 39-bit input address. This lookup must start with a level 1 lookup. For more information about the fields shown in the figure see *The address and properties fields shown in the translation flows* on page K7-5580.




For details of *Properties* fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-11 Complete stage 1 translation of a 39-bit address using the 4KB translation granule**

If the level 1 lookup or the level 2 lookup returns a block descriptor then the translation table walk completes at that level.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

Figure K7-11 on page K7-5582 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Comparing this translation with the translation for a 48-bit address, shown in Figure K7-10 on page K7-5581, shows how the translation for the 42-bit address start the same lookup process one stage later.

### Full translation flow using the 64KB granule and starting at level 1

Figure K7-10 on page K7-5581 shows the complete translation flow for a stage 1 translation table walk for a 48-bit input address. This lookup must start with a level 0 lookup. For more information about the fields shown in the figure see *The address and properties fields shown in the translation flows* on page K7-5580.

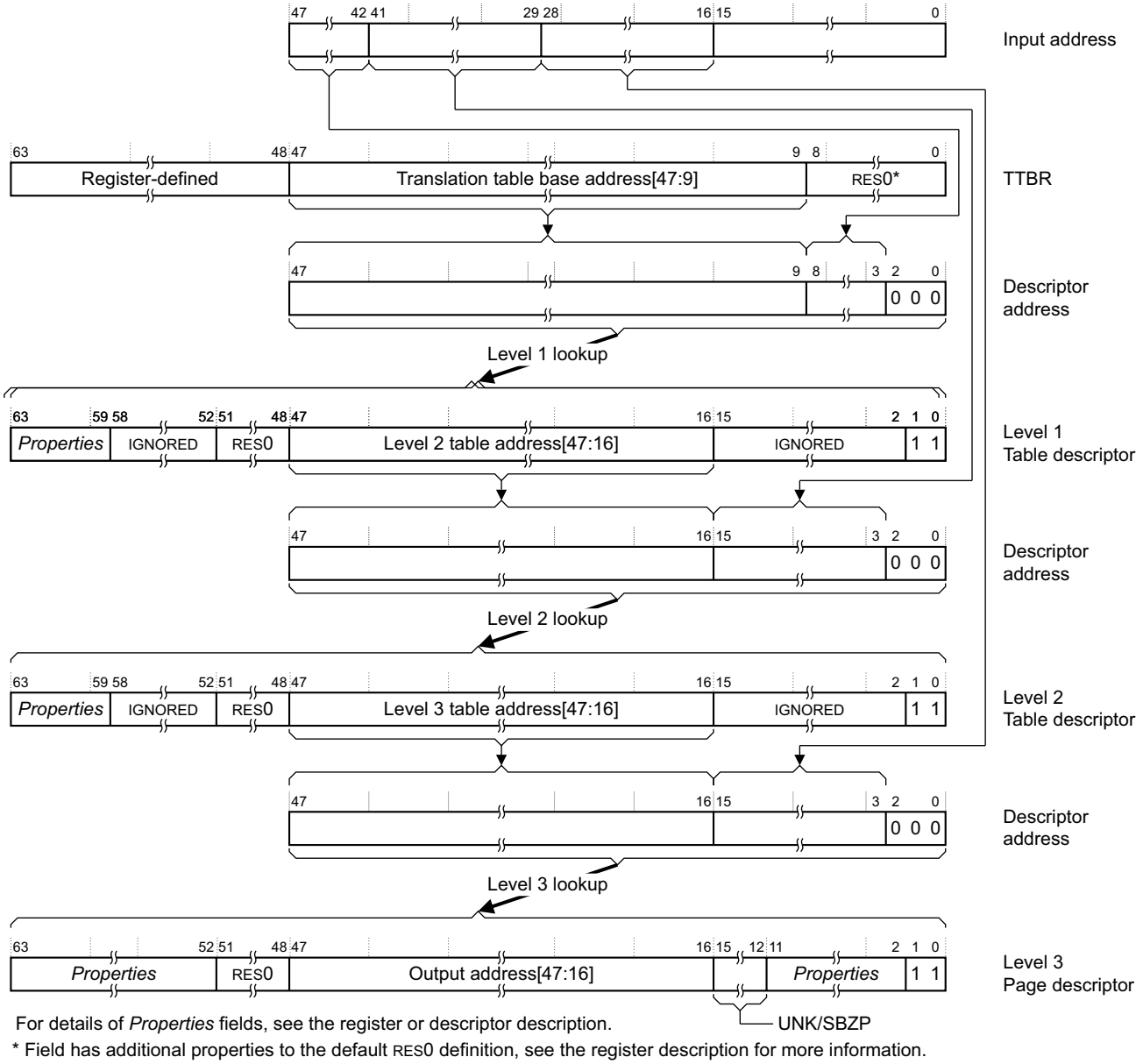



**Figure K7-12 Complete stage 1 translation of a 48-bit address using the 64KB translation granule**

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K7-12 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

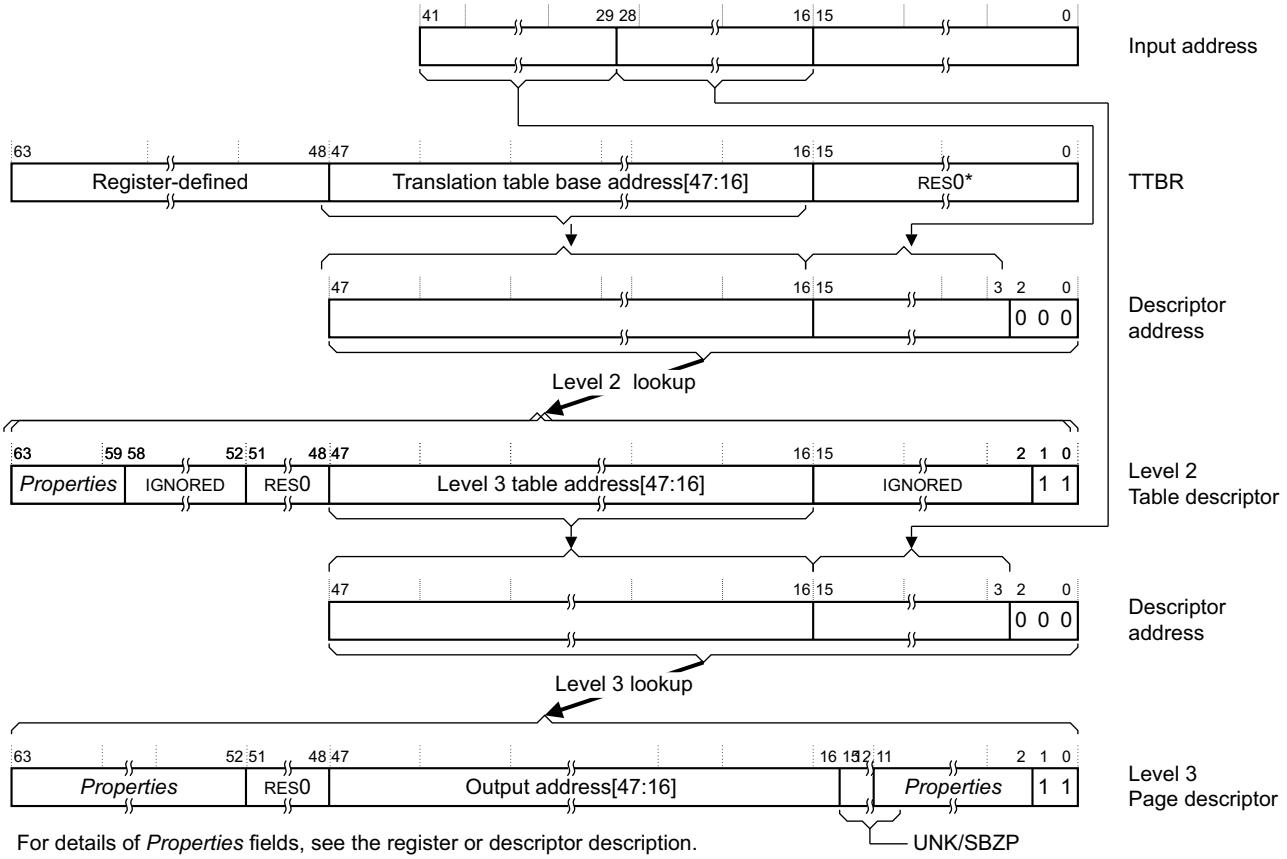

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

The level 1 lookup resolves only 6 bits of the input address. As described in *Performing the initial lookup using the 64KB translation granule* on page K7-5577, this means:

- The translation table size for this level is only 512 bytes.
- The required translation table alignment for this level is 512 bytes.
- The Base address field in the TTBR is extended, at the low-order end, to be bits[47:9].

### Full translation flow using the 64KB granule and starting at level 2

Figure K7-11 on page K7-5582 shows the complete translation flow for a stage 1 translation table walk for a 42-bit input address. This lookup must start with a level 2 lookup. For more information about the fields shown in the figure see *The address and properties fields shown in the translation flows* on page K7-5580.




For details of *Properties* fields, see the register or descriptor description.
* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K7-13 Complete stage 1 translation of a 42-bit address using the 64KB translation granule**

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K7-13 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Comparing this translation with the translation for a 48-bit address, shown in Figure K7-12 on page K7-5583, shows:

- The translation for the 42-bit address starts the same lookup process one stage later.
- Because the initial lookup resolves 13 bits of address:
  - The translation table size for this level is 64KB.
  - The required translation table alignment for this level is 64KB.
  - The Base address field in the TTBR is bits[47:16].


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K7.2 AArch32 Address translation examples

The following sections give address translation examples for the VMSAv8-32 address translation formats:

- *Address translation examples using the VMSAv8-32 Short descriptor translation table format*.
- *Address translation examples using the VMSAv8-32 Long descriptor translation table format* on page K7-5590.

### K7.2.1 Address translation examples using the VMSAv8-32 Short descriptor translation table format

*VMSAv8-32 Short-descriptor translation table format descriptors* on page G4-4010 describes the memory section and page option for a single VMSAv8-32 address translation. The following sections show the full translation flow for each of these options:

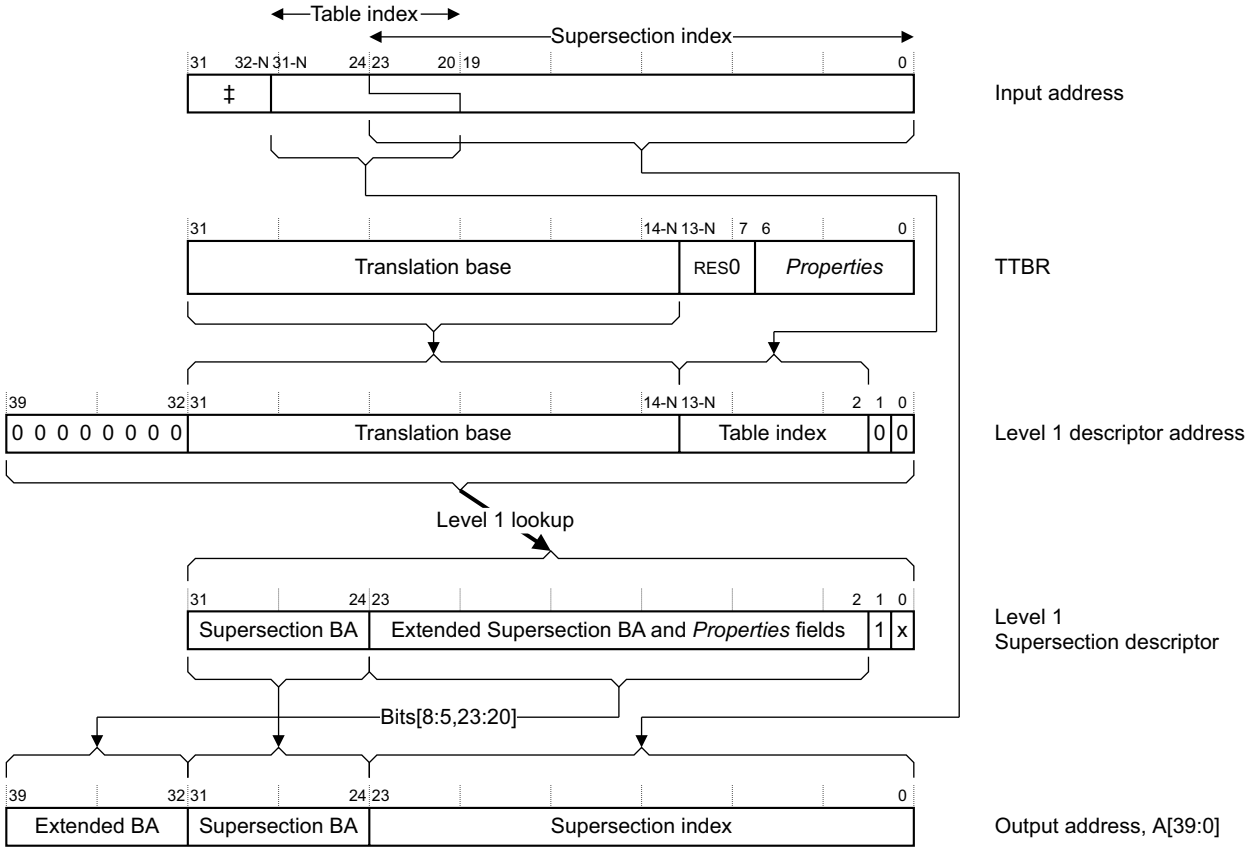
- *Translation flow for a Supersection* on page K7-5586.
- *Translation flow for a Section* on page K7-5587.
- *Translation flow for a Large page* on page K7-5588.
- *Translation flow for a Small page* on page K7-5589.

*The address and Properties fields shown in the translation flows* on page K7-5590 summarizes the information returned by the lookup.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Translation flow for a Supersection

Figure K7-14 shows the complete translation flow for a Supersection. For more information about the fields shown in this figure see *The address and Properties fields shown in the translation flows* on page K7-5590.




‡ This field is absent if N is 0
BA = Base address
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description

**Figure K7-14 VMSAv8-32 Short-descriptor Supersection address translation**

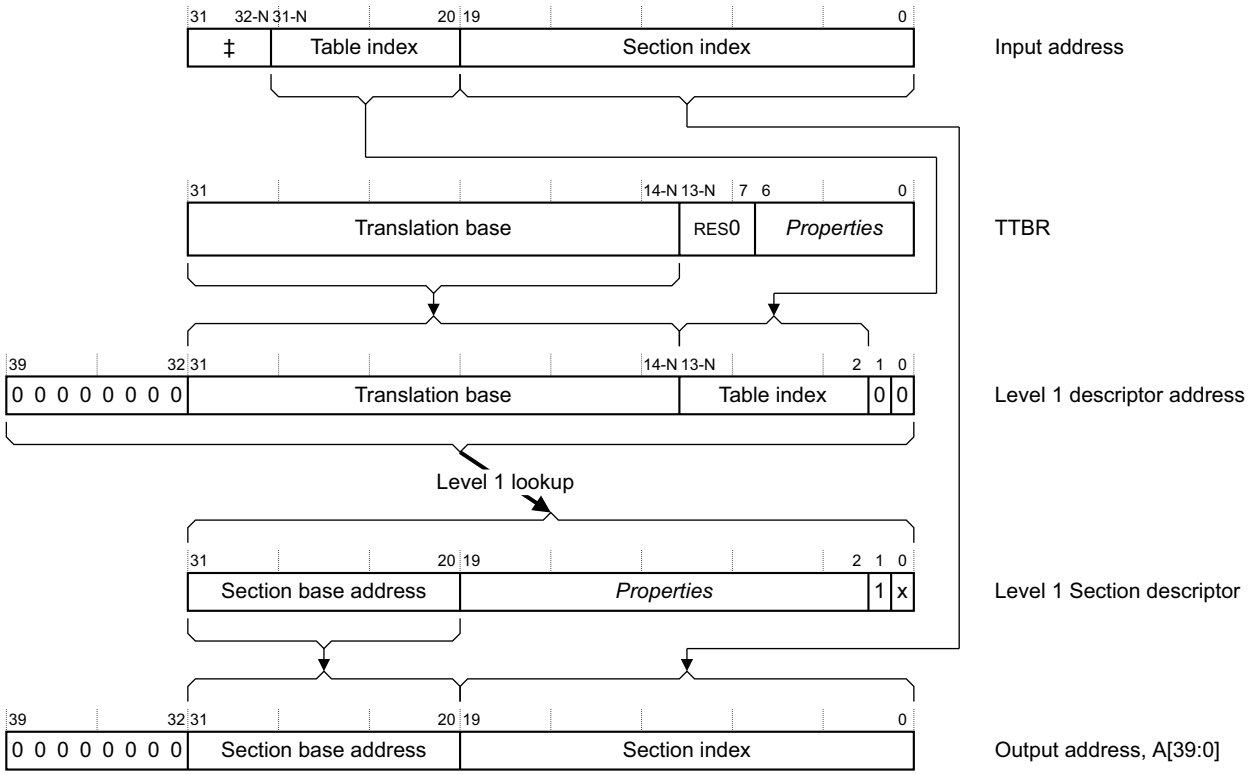
**Note**

Figure K7-14 shows how, when the input address, the VA, addresses a Supersection, the top four bits of the *Supersection index* bits of the address overlap the bottom four bits of the *Table index* bits. For more information, see *Additional requirements for Short-descriptor format translation tables* on page G4-4013.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Translation flow for a Section

Figure K7-15 shows the complete translation flow for a Section. For more information about the fields shown in this figure see *The address and Properties fields shown in the translation flows* on page K7-5590.



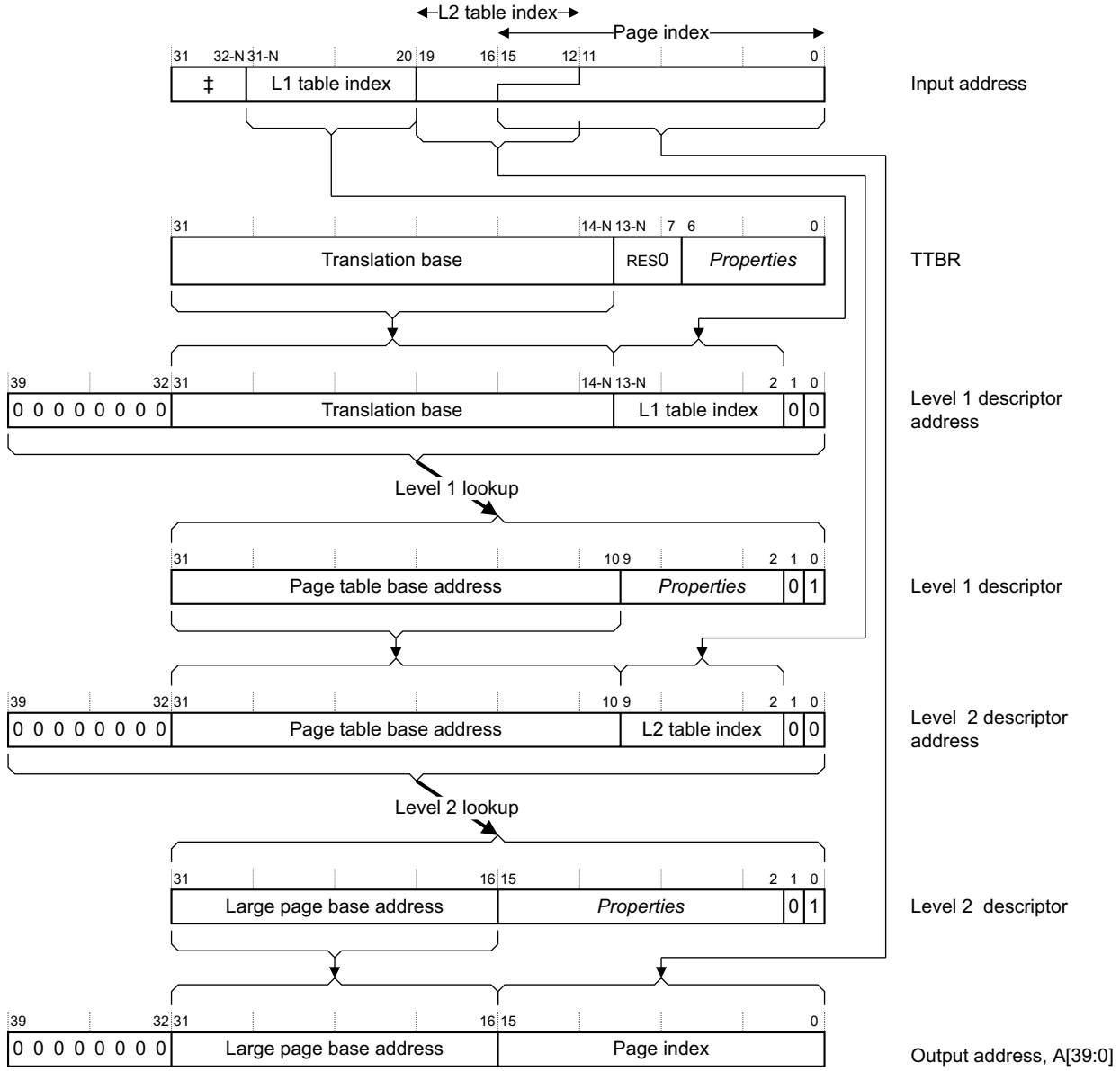

‡ This field is absent if N is 0
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description.

**Figure K7-15 VMSAv8-32 Short-descriptor Section address translation**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Translation flow for a Large page

Figure K7-16 shows the complete translation flow for a Large page. For more information about the fields shown in this figure see *The address and Properties fields shown in the translation flows* on page K7-5590.




‡ This field is absent if N is 0
L1 = Level 1, L2 = Level 2
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description

**Figure K7-16 VMSAv8-32 Short-descriptor Large page address translation**

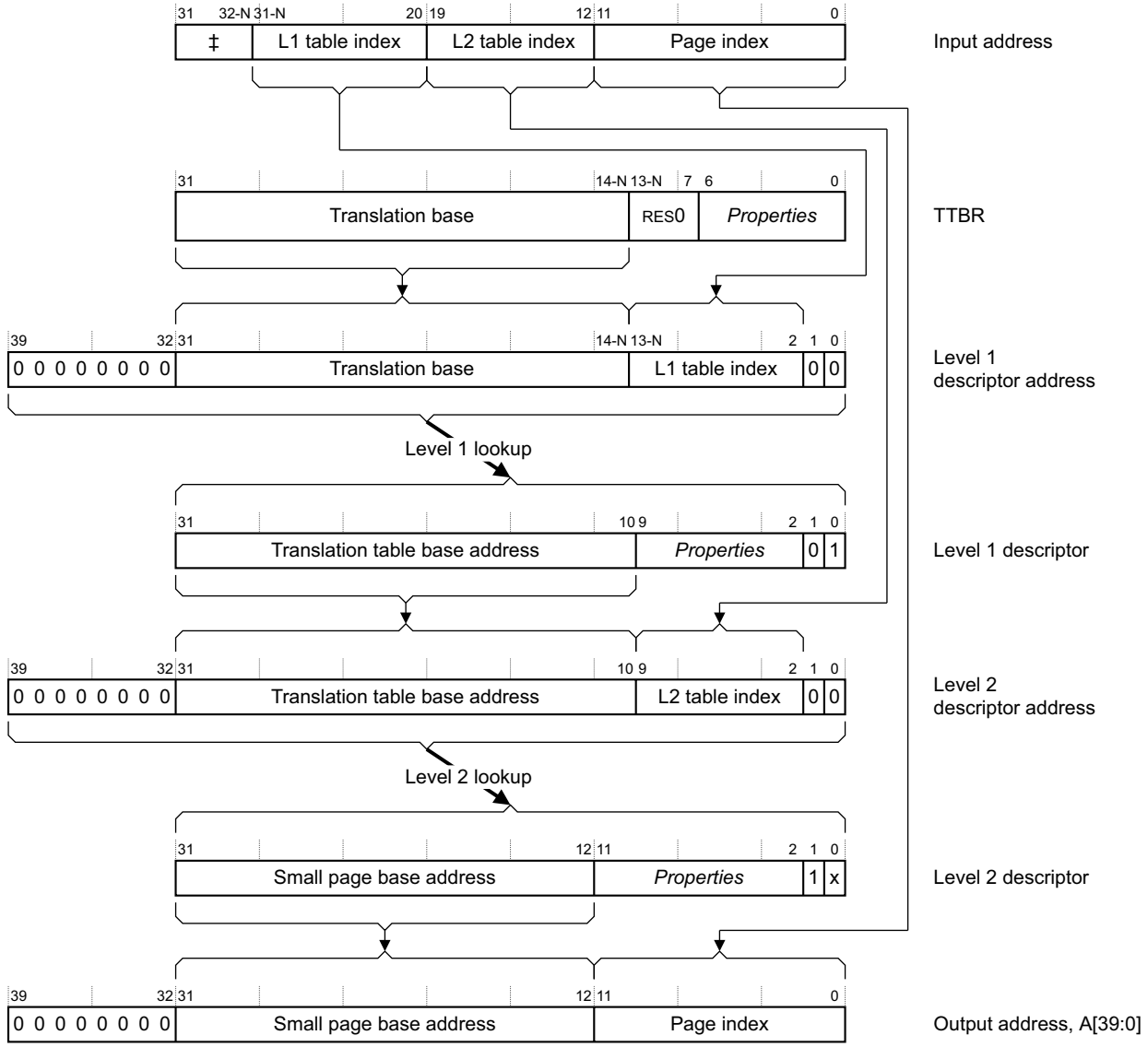
**Note**

Figure K7-16 shows how, when the input address, the VA, addresses a Large page, the top four bits of the *page index* bits of the address overlap the bottom four bits of the *level 1 table index* bits. For more information, see *Additional requirements for Short-descriptor format translation tables* on page G4-4013.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Translation flow for a Small page

Figure K7-17 shows the complete translation flow for a Small page. For more information about the fields shown in this figure see *The address and Properties fields shown in the translation flows* on page K7-5590.




‡ This field is absent if N is 0
L1 = Level 1, L2 = Level 2
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description.

**Figure K7-17 VMSAv8-32 Short-descriptor Small page address translation**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### The address and Properties fields shown in the translation flows

For the Non-secure PL1&0 stage 1 translation tables:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the Section, Supersection, Large page, or Small page.

In these cases, a PL1&0 stage 2 translation is performed to translate the IPA to the required PA.

Otherwise, the address is the PA of the descriptor, Section, Supersection, Large page, or Small page.

*Properties* indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information see *Information returned by a translation table lookup* on page G4-4005, and the description of the register or translation table descriptor.

For translations using the Short-descriptor translation table format, *VMSAv8-32 Short-descriptor translation table format descriptors* on page G4-4010 describes the descriptors formats.

## K7.2.2 Address translation examples using the VMSAv8-32 Long descriptor translation table format

As described in *Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format* on page G4-4032, in a translation table walk, only the first lookup uses the translation table base address from the appropriate TTBR. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

The following sections give examples of full VMSAv8-32 Long-descriptor format address translation flows, down to an entry for a 4KB page:

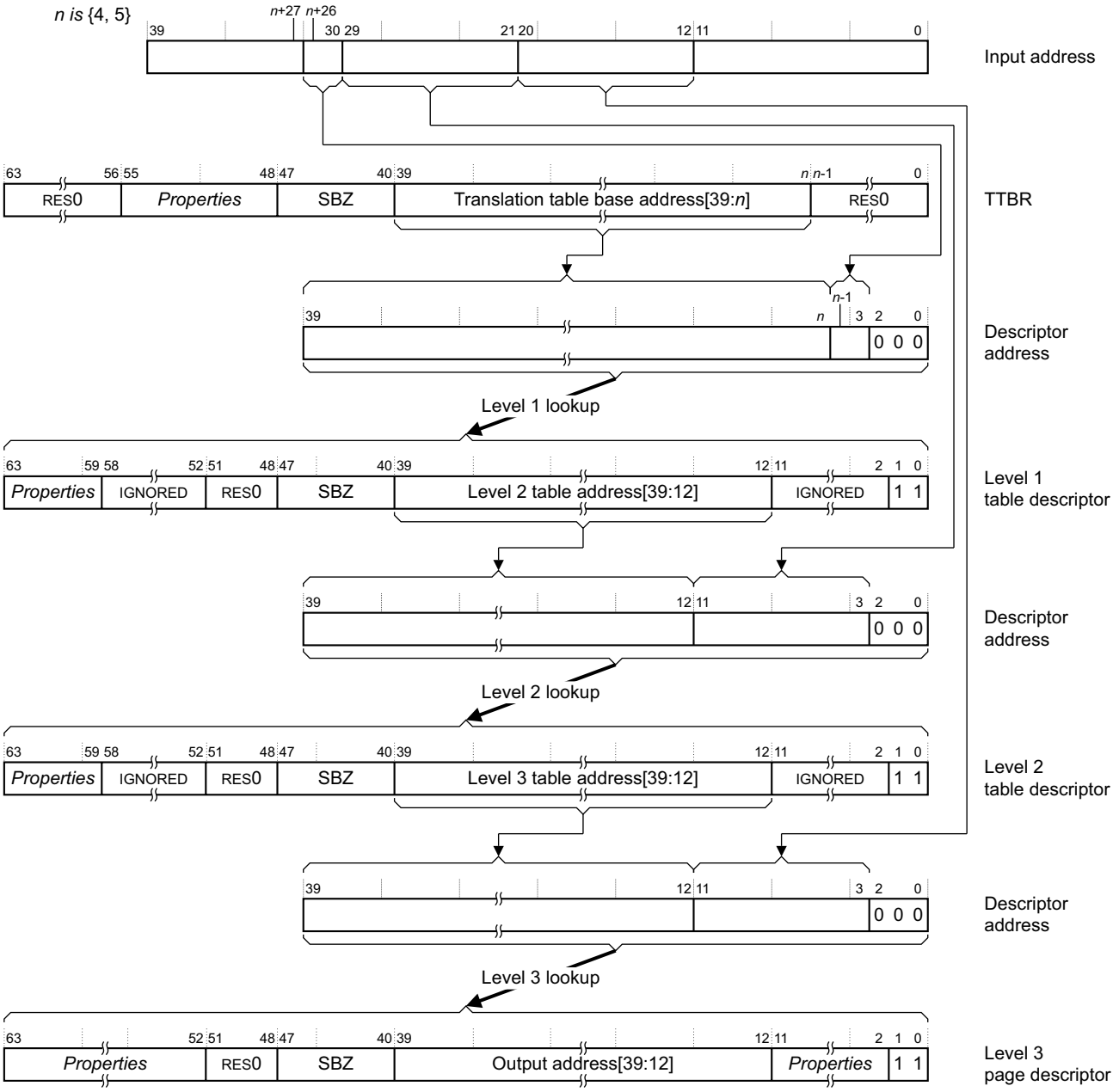
- *Full translation flow, starting at level 1 lookup* on page K7-5591.
- *Full translation flow, starting at level 2 lookup* on page K7-5592.

*The address and Properties fields shown in the translation flows* summarizes the information returned by the lookup.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Full translation flow, starting at level 1 lookup

Figure K7-18 shows the complete translation flow for a VMSAv8-32 Long-descriptor stage 1 translation table walk that starts with a level 1 lookup. For more information about the fields shown in the figure see *The address and Properties fields shown in the translation flows* on page K7-5590.




For details of *Properties* fields, see the register or descriptor description.

**Figure K7-18 Complete VMSAv8-32 Long-descriptor format stage 1 translation, starting at level 1**

If the level 1 lookup or the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

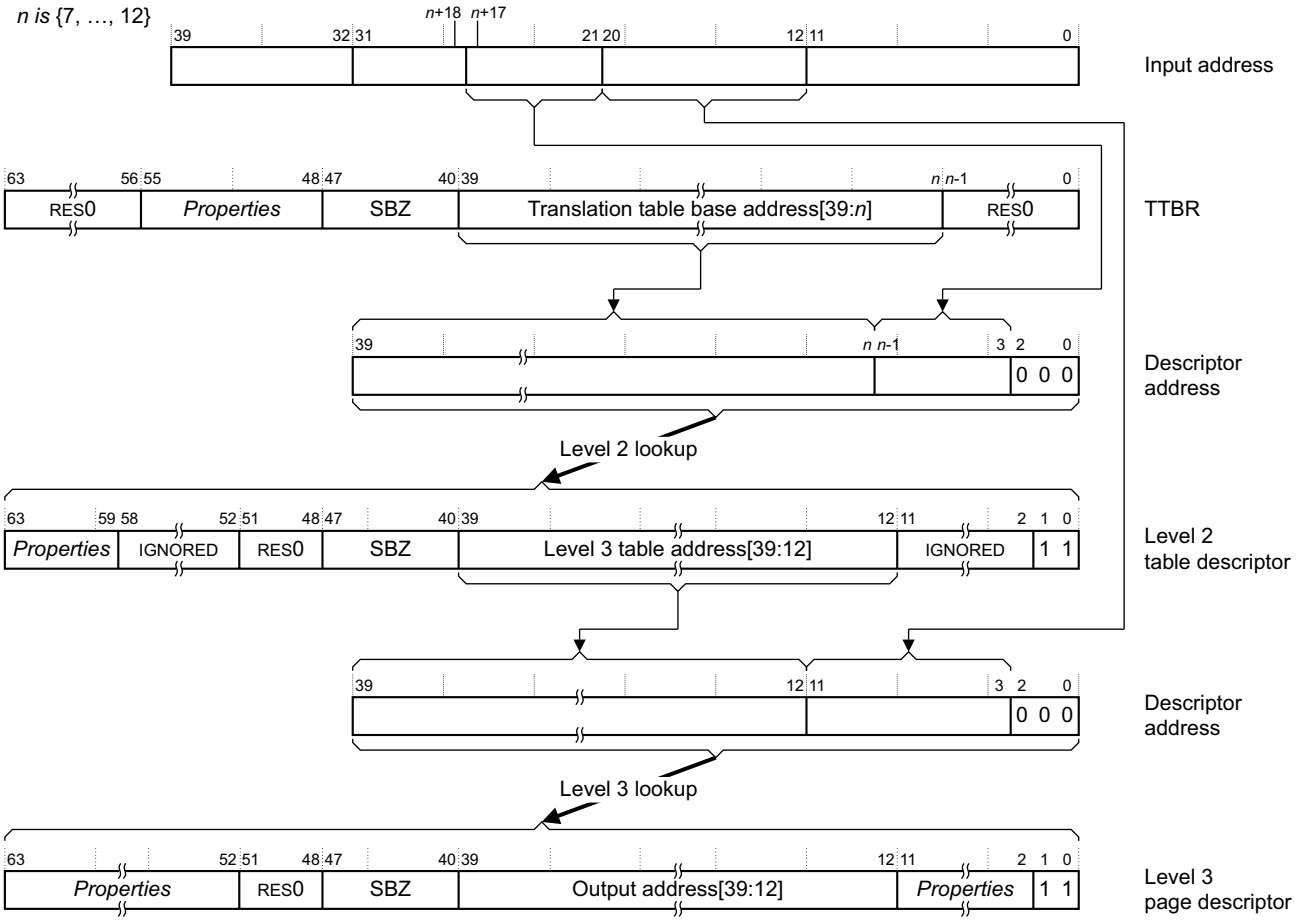

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

A stage 2 translation that starts at a level 1 lookup differs from the translation shown in Figure K7-18 on page K7-5591 only as follows:

- The possible values of *n* are 4-13, to support an input address of between 31 and 40 bits.
- A descriptor and output addresses are always PAs.

### Full translation flow, starting at level 2 lookup

Figure K7-19 shows the complete translation flow for a stage 1 VMSAv8-32 Long-descriptor translation table walk that starts at a level 2 lookup. For more information about the fields shown in the figure see *The address and Properties fields shown in the translation flows* on page K7-5590.




For details of *Properties* fields, see the register or descriptor description.

**Figure K7-19 Complete VMSAv8-32 Long-descriptor format stage 1 translation, starting at level 2**

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

A stage 2 translation that starts at a level 2 lookup differs from the translation shown in Figure K7-19 only as follows:

- The possible values of *n* are 7-16, to support an input address of up to 34 bits.
- The descriptor and output addresses are always PAs.

### The address and Properties fields shown in the translation flows

For the Non-secure PL1&0 stage 1 translation:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the block or page.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

In these cases, a PL1&0 stage 2 translation is performed to translate the IPA to the required PA.

For all other translations, the final output address is the PA of the block or page, and any descriptor address is the PA of the descriptor.

*Properties* indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information see *Information returned by a translation table lookup* on page G4-4005, and the description of the register or translation table descriptor.

For translations using the Long-descriptor translation table format, *VMSAv8-32 Long-descriptor translation table format descriptors* on page G4-4019 describes the descriptors formats.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K8
# Example OS Save and Restore Sequences

This appendix provides possible OS Save and Restore sequences for a v8 Debug implementation. It contains the following sections:


- *Save Debug registers* on page K8-5596.
- *Restore Debug registers* on page K8-5598.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K8.1 Save Debug registers

This section shows how to save the registers that are used by an external debugger.

```
; On entry, X0 points to a block to save the debug registers in.
; Returns the pointer beyond the block and corrupts X1-X3

SaveDebugRegisters
    ; (1) Set OS lock.
    MOV     X2,#1                       ; Set the OS lock. In AArch64 state, the OS lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.
    ISB                                 ; Context synchronization operation

    ; (2) Walk through the registers, saving them
    MRS     X1,OSDTRRX_EL1              ; Read DTRRX
    MRS     X2,OSDTRTX_EL1              ; Read DTRTX
    STP     W1,W2,[X0],#8               ; Save { DTRRX, DTRTX }
    MRS     X1,OSECCR_EL1               ; Read ECCR
    MRS     X2,MDSCR_EL1                ; Read DSCR
    STP     W1,W2,[X0],#8               ; Save { ECCR, DSCR }
    [ AARCH32_SUPPORTED
    MRS     X1,DBGVCR32_EL2             ; Read DBGVCR
    MRS     X2,DBGCLAIMCLR_EL1          ; Read CLAIM - note, have to read via CLAIMCLR
    STP     W1,W2,[X0],#8               ; Save { VCR, CLAIM }
    ]

    ;; Macros for saving off a "register pair"
    ;; $WB      is W for watchpoint, B for breakpoint
    ;; $num     is the pair's number
    ;; X0 contains a pointer for the value words
    ;; X1 contains a pointer for the control words
    ;; W2 contains the max index
    MACRO
    SaveRP  $WB,$num, $exit
    MRS     X3,DBG$WB.VR$num._EL1       ; Read DBGxVRn
    STR     X3,[X0],#8                  ; Save { xVRn }
    MRS     X3,DBG$WB.CR$num._EL1       ; Read DBGxCRn
    STR     W3,[X0],#4                  ; Save { xCRn }.
    [ $num > 1 :LAND: $num < 15
    CMP     W1,#$num
    BEQ     $exit
    ]
    MEND

    ; (3) Breakpoints
    MRS     X1,ID_AA64DFR0_EL1
    UBFX    W1,W1,#12,#4                ; Extract BRPs field
    MACRO
    SaveBRP $num                        ; Save a Breakpoint Register Pair
    SaveRP  B,$num,SaveDebugRegisters_Watchpoints
    MEND
    SaveBRP 0
    SaveBRP 1
    SaveBRP 2
    ;; and so on to ...
    SaveBRP 15

SaveDebugRegisters_Watchpoints
    ; (4) Watchpoints
    MRS     X1,ID_AA64DFR0_EL1          ; Read DBGDIDR
    UBFX    W1,W1,#20,#4                ; Extract WRPs field
    MACRO
    SaveWRP $num                        ; Save a Watchpoint Register Pair
    SaveRP  W,$num,SaveDebugRegisters_Exit
    MEND
    SaveWRP 0
    SaveWRP 1
    SaveWRP 2
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
    ;; and so on to ...
    SaveWRP 15

SaveDebugRegisters_Exit
    ; (5) Return the pointer to first word not read. This pointer is already in X0, so
    ; all that is needed is to return from this function. The OS double-lock (OSDLR_EL1.DLK) is
    ; locked later, just before the final entry to WFI state.
    RET
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K8.2 Restore Debug registers

This section shows how to restore the registers that are used by an external debugger.

```
; On entry, X0 points to a block of saved debug registers.
; Returns the pointer beyond the block and corrupts R1-R3,R12.

RestoreDebugRegisters
    ; (1) Lock OS lock. The lock will already be set, but this write is included to ensure it
    ; is locked.
    MOV     X2,#1                       ; Lock the OS lock. In AArch64 state, the OS lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.
    ISB                         ; Context synchronization operation

    MSR MDSCR_EL1, XZR                  ; Initialize MDSCR_EL1

    ; (2) Walk through the registers, restoring them
    LDP     W1,W2,[X0],#8               ; Read { DTRRX,DTRTX }
    MSR     OSDTRRX_EL1,X1              ; Restore DTRRX
    MSR     OSDTRTX_EL1,X2              ; Restore DTRTX
    LDP     W1,W3,[X0],#8               ; Read { DSCR, ECCR }
    MSR     OSECCR_EL1,X2               ; Restore ECCR
    [ AARCH32_SUPPORTED
    LDP     W1,W2,[X0],#8               ; Read { VCR,CLAIM }
    MSR     DBGVCR32_EL2,X1             ; Restore DBGVCR
    MSR     DBGCLAIMSET_EL1,X2          ; Restore CLAIM - note, writes CLAIMSET
    ]

    ;; Macro for restoring a "register pair"
    MACRO
    RestoreRP $WB,$num,$exit
    LDR     X3,[X0],#8                  ; Read { xVRn }
    MSR     DBG$WB.VR$num._EL1,X3       ; Restore DBGxVRn
    LDR     W3,[X0],#4                  ; Read { xCRn }
    MSR     DBG$WB.CR$num._EL1,X3       ; Restore DBGxCRn
    [ $num >= 1 :LAND: $num < 15
    CMP     W1,#$num
    BEQ     $exit
    ]
    MEND

    ; (3) Breakpoints
    MRS     X1,ID_AA64DFR0_EL1
    UBFX    W1,W1,#12,#4                ; Extract BRPs field
    MACRO
    RestoreBRP $num                     ; Restore a Breakpoint Register Pair
    RestoreRP  B,$num,RestoreDebugRegisters_Watchpoints
    MEND
    RestoreBRP 0
    RestoreBRP 1
    RestoreBRP 2
    ;; and so on until ...
    RestoreBRP 15

RestoreDebugRegisters_Watchpoints
    ; (4) Watchpoints
    MRS     X1,ID_AA64DFR0_EL1          ; Read DBGDIDR
    UBFX    W1,W1,#20,#4                ; Extract WRPs field
    MACRO
    RestoreWRP $num                     ; Restore a Watchpoint Register Pair
    RestoreRP  W,$num,RestoreDebugRegisters_Exit
    MEND
    RestoreWRP 0
    RestoreWRP 1
    RestoreWRP 2
    ;; and so on until ...
    RestoreWRP 15
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
RestoreDebugRegisters_Exit
    MSR MDSCR_EL1, X3                   ; Restore DSCR

    ; (5) Clear the OS lock.
    ISB
    MOV     X2,#0                       ; Clear the OS lock. In AArch64 state, the OS lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.

    ; (6) A final ISB guarantees the restored register values are visible to subsequent
    ; instructions.
    ISB

    ; (7) Return the pointer to first word not read. This pointer is already in X0, so
    ; all that is needed is to return from this function.
    RET
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K9
# Recommended Upload and Download Processes for External Debug

This appendix contains the following section:


- *Using memory access mode in AArch64 state* on page K9-5602.


**Note**

This description is not part of the ARM architecture specification. It is included here as supplementary information, for the convenience of developers and users who might find this information useful.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K9.1 Using memory access mode in AArch64 state

Figure K9-1 and Figure K9-2 on page K9-5603 show the processes for using memory access mode to implement a download (external host to target) and an upload (target to external host).

To transfer *n* words of data:

- The download sequence needs *n*+6 accesses by the external debug interface.
- The upload sequence needs *n*+8 accesses by the external debug interface.

In both cases, in the innermost loop the debugger can make an external access to a DTR without polling EDSCR after each write as underrun and overrun detection prevent failure. Normally external accesses from the debugger are outpaced by the memory accesses of the PE, making underruns and overruns unlikely. If this is not the case, the EDSCR.ERR flag is set to 1. This is checked once at the end of the sequence, although a debugger can check it more often, for example once for each page. If the EDSCR.ERR flag is set to 1 because of overrun or underrun, the debugger can restart. The address to restart from is frozen in X0. EDSCR.ERR might also be set because of a Data abort.

If underruns and overruns are common, the debugger can pace itself accordingly.

**Note**

- The base address must be a multiple of 4.
- The order of the writes that set up the address does not matter in Debug state.

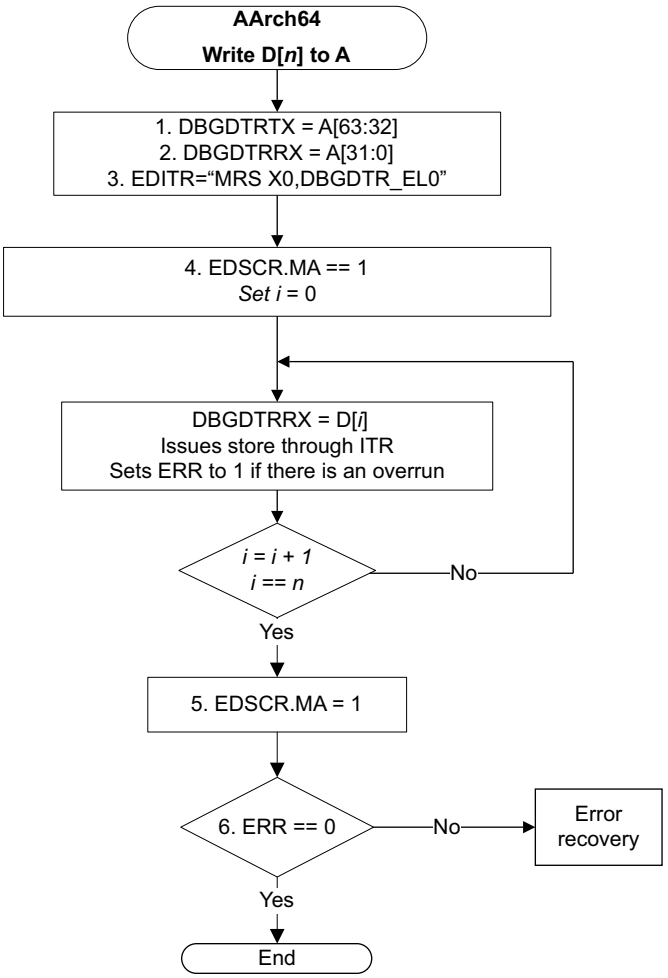



**Figure K9-1 Fast code download in AArch64 (external host to target)**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

In Figure K9-1 on page K9-5602, the sequence for the fast code download is as follows:

1. Setup. From the external debug interface:
   a. Write address [31:0] to DBGDTRRX_EL0.
   b. Write address [63:32] to DBGDTRTX_EL0.
   c. Write `MRS X0, DBGDTR_EL0` to EDITR. The PE executes this instruction.
   d. Set EDSCR.MA to 1.
2. Loop *n* times. From the external debug interface:
   a. Write to DBGDTRRX_EL0. The PE reads the word from DTRRX and stores it to memory. It increments X0 by 4.
3. Epilogue. From the external debug interface:
   a. Clear EDSCR.MA to 0.
   b. Read EDSCR to check for overruns or Data Aborts during download.

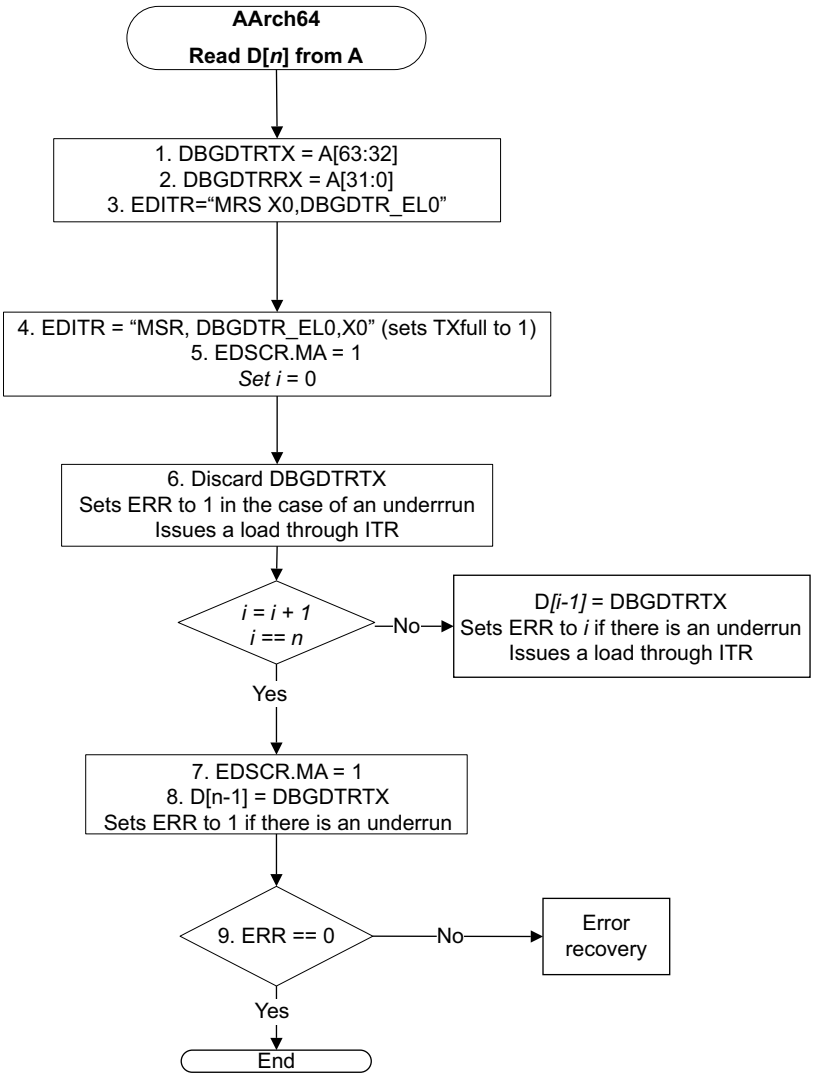



**Figure K9-2 Fast data upload in AArch64 (target to external host)**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

In Figure K9-2 on page K9-5603, the sequence for the fast code download is as follows:

1. Setup. From the external debug interface:
   a. Write address [31:0] to DBGDTRRX_EL0.
   b. Write address [63:32] to DBGDTRTX_EL0.
   c. Write `MRS X0, DBGDTR_EL0` to EDITR.
   d. Write `MSR DBGDTR_EL0, X0` to EDITR. This dummy operation ensures EDSCR.TXfull == 1.
   e. Set EDSCR.MA to 1.
   f. Read DBGDTRTX_EL0 and discard the value. The PE returns the previous DTR value, loads the first word, and writes it to DTR. It increments X0 by 4.
2. Loop *n*-1 times. From the external debug interface:
   a. Read DBGDTRTX_EL0. The PE returns the previous DTRTX value, loads a new word, and writes it to DTRTX. It increments X0 by 4.
3. Epilogue. From the external debug interface:
   a. Clear EDSCR.MA to 0.
   b. Read DBGDTRTX_EL0 for the $n^{th}$ value.
   c. Read EDSCR to check for underruns, overruns or Data Aborts during upload.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K10
# **Barrier Litmus Tests**

This appendix gives examples of the use of the barrier instructions provided by the ARMv8 architecture. It contains the following sections:


- *Introduction* on page K10-5606.
- *Load-Acquire, Store-Release and barriers* on page K10-5609.
- *Load-Acquire Exclusive, Store-Release Exclusive and barriers* on page K10-5616.
- *Using a mailbox to send an interrupt* on page K10-5621.
- *Cache and TLB maintenance instructions and barriers* on page K10-5622.
- *ARMv7 compatible approaches for ordering, using DMB and DSB barriers* on page K10-5633.


**Note**

This information is not part of the ARM architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# K10.1 Introduction

The exact rules for the insertion of barriers into code sequences is a very complicated subject, and this appendix describes many of the corner cases and behaviors that are possible in an implementation of the ARMv8 architecture.

This appendix is to help programmers, hardware design engineers, and validation engineers understand the need for the different kinds of barriers.

## K10.1.1 Overview of memory consistency

Early generations of microprocessors were relatively simple processing engines that executed each instruction in program order. In such processors, the effective behavior was that each instruction was executed in its entirety before a subsequent instruction started to be executed. This behavior is sometimes referred to as the *Sequential Execution Model* (SEM), and in this Manual it is described as *Simple sequential execution* of the program.

In later processor generations, the needs to increase processor performance, both in terms of the frequency of operation and the number of instructions executed each cycle, mean that such a simple form of execution is abandoned. Many techniques, such as pipelining, write buffering, caching, speculation, and out-of-order execution, are introduced to provide improved performance.

For general purpose PEs, such as ARM, these microarchitectural innovations are largely hidden from the programmer by a number of microarchitectural techniques. These techniques ensure that, within an individual PE, the behavior of the PE largely remains the same as the SEM. There are some exceptions to this where explicit synchronization is required. In the ARM architecture, these are limited to cases such as:

- Synchronization of changes to the instruction stream.
- Synchronization of changes to System registers.

In both these cases, the `ISB` instruction provides the necessary synchronization.

While the effect of ordering is largely hidden from the programmer within a single PE, the microarchitectural innovations have a profound impact on the ordering of memory accesses. Write buffering, speculation, and cache coherency protocols, in particular, can all mean that the order in which memory accesses occur, as seen by an external observer, differs significantly from the order of accesses that would appear in the SEM. This is usually invisible in a uniprocessor environment, but the effect becomes much more significant when multiple PEs are trying to communicate with memory. In reality, these effects are often only significant at particular synchronization boundaries between the different threads of execution.

The problems that arise from memory ordering considerations are sometimes described as the problem of *memory consistency*. Processor architectures have adopted one or more *memory consistency models*, or *memory models*, that describe the permitted limits of the memory re-ordering that can be performed by an implementation of the architecture. The comparison and categorization of these has generated significant research and comment in academic circles, and ARM recommends the *Memory Consistency Models for Shared Memory-Multiprocessors* paper as an excellent detailed treatment of this subject.

This appendix does not reproduce such a work, but instead concentrates on some cases that demonstrate the features of the weakly-ordered memory model of the ARM architecture from ARMv6. In particular, the examples show how the use of the `DMB` and `DSB` memory barrier instructions can provide the necessary safeguards to limit memory ordering effects at the required synchronization points.

## K10.1.2 Barrier operation definitions

The following reference, or provide, definitions of terms used in this appendix:

**DMB** See *Data Memory Barrier (DMB)* on page B2-87.

**DSB** See *Data Synchronization Barrier (DSB)* on page B2-88.

**ISB** See *Instruction Synchronization Barrier (ISB)* on page B2-87.

**Observer, Completion**

See *Observability and completion* on page B2-83.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Program order**

The order of instructions as they appear in an assembly language program. This appendix does not attempt to describe or define the legal transformations from a program written in a higher level programming language, such as C or C++, into the machine language that can then be disassembled to give an equivalent assembly language program. Such transformations are a function of the semantics of the higher level language and the capabilities and options on the compiler.

### K10.1.3 Conventions

Many of the examples are written in a stylized extension to ARM assembler, to avoid confusing the examples with unnecessary code sequences.

**AArch32**

The construct `WAIT([Rx]==1)` describes the following sequence:

```
loop
    LDR R12, [Rx]
    CMP R12, #1
    BNE loop
```

Also, the construct `WAIT_ACQ([Rx]==1)` describes the following sequence:

```
   loop
    LDA R12, [Rx]       ; load acquire ensures it is ordered before subsequent loads/stores
    CMP R12, #1
    BNE loop
```

R12 is chosen as an arbitrary temporary register that is not in use. It is named to permit the generation of a false dependency to ensure ordering.

**AArch64**

The construct `WAIT([Xx]==1)` describes the following sequence:

```
loop
    LDR W12, [Xx]
    CMP W12, #1
    B.NE loop
```

Also, the construct `WAIT_ACQ([Xx]==1)` and describes the following sequence:

```
   loop
    LDAR W12, [Xx]       ; load acquire ensures it is ordered before subsequent loads/stores
    CMP W12, #1
    B.NE loop
```

For each example, a code sequence is preceded by an identifier of the observer running it:

- P0, P1…P*x* refer to caching coherent PEs that implement the ARMv8 architecture, and are in the same shareability domain.
- E0, E1…E*x* refer to non-caching observers, that do not participate in the coherency protocol, but execute ARM instructions and have a weakly-ordered memory model. This does not preclude these observers being different objects, such as DMA engines or other system masters.

These observers are unsynchronized other than as required by the documented code sequence.

**Note**

Throughout this appendix, *ARM instruction* and *instruction* refer to instructions from the A64, A32, or T32 instruction set, provided by ARMv8 implementations.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Results are expressed in terms of `<agent>:<register>`, such as `P0:R5`. The results can be described as:

**Permissible** This does not imply that the results expressed are required or are the only possible results. In most cases they are results that would not be possible under a sequentially consistent running of the code sequences on the agents involved. In general terms, this means that these results might be unexpected to anyone unfamiliar with memory consistency issues.

**Not permissible** Results that the architecture expressly forbids.

**Required** Results that the architecture expressly requires.

The examples omit the required shareability domain arguments of `DMB` and `DSB` instructions. The arguments are assumed to be selected appropriately for the shareability domains of the observers.

In AArch32 state, where the barrier function in the litmus test can be achieved by a `DMB ST`, that is a barrier to stores only, this is shown by the use of `DMB [ST]`. This indicates that the `ST` qualifier can be omitted without affecting the result of the test. In some implementations `DMB ST` is faster than `DMB`.

For AArch64 code, the shareability domain of the `DMB` or `DSB` must be included. This is shown in this manual using the notation `DMB <domain>` and `DSB <domain>` respectively.

Except where otherwise stated, other conventions are:

- All memory initializes to 0.
- R0 and W0 contain the value 1.
- R1 - R4 and W1 - W4 contain arbitrary independent addresses that initialize to the same value on all PEs. The addresses held in these registers are shareable and:
  - The addresses held in R1 and R2 are in Write-Back Cacheable Normal memory.
  - The address held in R3 is in Write-Through Cacheable Normal memory.
  - The address held in R4 is in Non-cacheable Normal memory.
- R5 - R8 and W5 - W8 contain:
  - When used with an `STR` instruction, `0x55`, `0x66`, `0x77`, and `0x88` respectively.
  - When used with an `LDR` instruction, the value 0.
- R11 and W11 contain a new instruction or new translation table entry, as appropriate, and R10 contains the virtual address and the ASID, for use in this change of translation table entry.
- Memory locations are Normal memory locations unless otherwise stated.

The examples use mnemonics for the cache maintenance and TLB maintenance instructions. The following tables describe the mnemonics:

- *Cache maintenance instructions, functional group* on page G4-4169.
- *TLB maintenance instructions, functional group* on page G4-4171.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K10.2 Load-Acquire, Store-Release and barriers

The Load-Acquire and Store-Release instructions are described in *Load-Acquire, Store-Release* on page B2-89.

The following sections show that most of the examples in sections *Simple ordering and barrier cases* on page K10-5633 and *Load-Exclusive, Store-Exclusive and barriers* on page K10-5639 can be achieved using the Load-Acquire and Store-Release instructions without the need for additional barriers.

### K10.2.1 Message passing

The following sections describe:

- *Resolving weakly-ordered message passing by using Acquire and Release*.
- *Resolving message passing by the use of Store-Release and address dependency* on page K10-5610.

#### Resolving weakly-ordered message passing by using Acquire and Release

The message passing problem described in *Weakly-ordered message passing problem* on page K10-5633 can be solved by the use of Load-Acquire and Store-Release instructions when accessing the communications flag:

**AArch32**

P1

```
STR R5, [R1]          ; sets new data
STL R0, [R2]          ; sends flag indicating data ready, which is ordered after the STR
```

P2

```
WAIT_ACQ([R2]==1)     ; waits on flag
LDR R5, [R1]
```

**AArch64**

P1

```
STR W5, [X1]          ; sets new data
STLR W0, [X2]         ; sends flag indicating data ready, which is ordered after the STR
```

P2

```
WAIT_ACQ([X2]==1)     ; waits on flag
LDR W5, [X1]
```

This ensures the observed order of both the reads and the writes allows transfer of data such that the result P2:R5==0x55 is guaranteed.

This approach also works with multiple observers, in a way that further observers use the same sequence as P2 uses:

**AArch32**

P3

```
WAIT_ACQ([R2]==1)     ; waits on flag
LDR R5, [R1]
```

**AArch64**

P3

```
WAIT_ACQ([X2]==1)     ; waits on flag
LDR W5, [X1]
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Resolving message passing by the use of Store-Release and address dependency

The lack of ordering of stores discussed in *Message passing with multiple observers* on page K10-5635 can be resolved by the use of Store-Release for the store of the valid flag by P1, even when the observers are using an address dependency:

**AArch32**

P1

```
STR R5, [R1]          ; sets new data
STL R0, [R2]          ; sends flag indicating data ready using a Store-Release
```

P2

```
WAIT([R2]==1)
AND R12, R12, #0      ; R12 is the destination of LDR in the WAIT macro
LDR R5, [R1, R12]     ; the load has an address dependency on R12
                      : and so is ordered after the flag has been seen
```

**AArch64**

P1

```
STR W5, [X1]          ; sets new data
STLR W0, [X2]         ; sends flag indicating data ready using a Store-Release
```

P2

```
WAIT([X2]==1)
AND W12, W12, WZR     ; W12 is the destination of LDR in the WAIT macro
LDR W5, [X1, X12]     ; the load has an address dependency on W12
                      : and so is ordered after the flag has been seen
```

This ensures the observed order of the writes allows transfer of data such that P2:R5 and P3:R5 contain the same value of `0x55`.

This approach also works with multiple observers, in a way that further observers use the same sequence as P2 uses:

**AArch32**

P3

```
WAIT([R2]==1)
AND R12, R12, #0      ; R12 is the destination of LDR in the WAIT macro
LDR R5, [R1, R12]     ; the load has an address dependency on R12
                      : and so is ordered after the flag has been seen
```

**AArch64**

P3

```
WAIT([X2]==1)
AND W12, W12, WZR     ; R12 is the destination of LDR in the WAIT macro
LDR W5, [X1, X12]     ; the load has an address dependency on W12
                      : and so is ordered after the flag has been seen
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### K10.2.2 Address dependency with object construction

When accessing an object-oriented data structure, the address dependency rule means that barriers are not required, even when initializing the object. A Store-Release can be used to ensure the order of the update of the base address:

**AArch32**

P1

```
STR R5, [R1, #offset] ; sets new data in a field
STL R1, [R2]          ; updates base address
```

P2

```
LDR R1, [R2]          ; reads base address
CMP R1, #0            ; checks if it is valid
BEQ null_trap
LDR R5, [R1, #offset] ; uses base address to read field
```

**AArch64**

P1

```
STR W5, [X1, #offset] ; sets new data in a field
STLR X1, [X2]         ; updates base address
```

P2

```
LDR X1, [X2]          ; reads base address
CMP X1, #0            ; check if it is valid
B.EQ null_trap
LDR W5, [X1, #offset] ; uses base address to read field
```

It is required that P2:R5==0x55 if the null_trap is not taken. This avoids P2 observing a partially constructed object from P1. Significantly, P2 does not need a barrier to ensure this behavior.

The read of the base address in P2 could be a Load-Acquire, but it is not necessary in this case.

### K10.2.3 Causal consistency issues with multiple observers

The cause consistent problem discussed in *Causal consistency issues with multiple observers* on page K10-5636 can be addressed by the use of a Store-Release, as this requires that the store is multicopy atomic in the case of a Load-Acquire. In addition, a Store-Release has an effect on the observation order of any stores observed by the observer executing the Store-Release.

The following sequences guarantee causal consistency:

- *Using multi-copy atomicity of the Store-Release when observed by Load-Acquire*.
- *Using ordering property of Store-Release on stores observed by the PE* on page K10-5612.

#### Using multi-copy atomicity of the Store-Release when observed by Load-Acquire

**AArch32**

P1

```
STL R0, [R2]          ; sets new data
                      ; this is made multi-copy atomic
```

P2

```
WAIT_ACQ([R2]==1)     ; waits to see new data from P1
STR R0, [R3]          ; sends flag
                      ; must be after the new data has been by P2,
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
                        ; as stores must not be speculative
                        ; this does not need to be a Store-Release,
                        ; though it could be a Store-Release
```

P3

```
    WAIT([R3]==1)       ; waits for P2 flag
                        ; this does not need to be a WAIT_ACQ, although
                        ; it could be a WAIT_ACQ (at which point the dependency is not needed)
    AND R12, R12, #0    ; dependency to ensure order (only needed for a WAIT, not WAIT_ACQ)
    ADD R12, R2, R12    ; creating a dependency
    LDA R0, [R12]       ; reads P1 data using a Load-Acquire
```

**AArch64**

P1

```
    STLR W0, [X2]       ; sets new data
                        ; this is made multi-copy atomic
```

P2

```
    WAIT_ACQ([X2]==1)   ; waits to see new data from P1
    STR W0, [X3]        ; sends flag
                        ; must be after the new data has been by P2 as stores
                        ; must not be speculative.
                        ; this does not need to be a Store-Release,
                        ; though it could be a Store-Release
```

P3

```
    WAIT([X3]==1)       ; waits for P2 flag
                        ; this does not need to be a WAIT_ACQ, though
                        ; it could be a WAIT_ACQ (at which point the dependency is not needed
    AND W12, W12, WZR   ; dependency to ensure order (only needed for a WAIT, not WAIT_ACQ)
    ADD X12, X2, X12    ; creating a dependency
    LDAR W0, [X12]      ; reads P1 data using a Load-Acquire
```

In this case, P3:R0 == 0 is not permissible. P3 is guaranteed to see the store from P1 if P2 has seen the store from P1 using a Load-Acquire.

### Using ordering property of Store-Release on stores observed by the PE

**AArch32**

P1

```
    STR R0, [R2]        ; sets new data
                        ; this does not have to be a Store-Release, though it
                        ; could be a Store-release
```

P2

```
    WAIT ([R2]==1)      ; waits to see new data from P1
                        ; this does not need to be a WAIT_ACQ, though it could be a WAIT_ACQ
    STL R0, [R3]        ; sends flag
                        ; this must be after the new data has been seen by P2
                        : as stores must not be speculative,
                        ; as a Store-Release, this orders the P1 store
```

P3

```
    WAIT([R3]==1)       ; waits for P2 flag
                        ; this does not need to be a WAIT_ACQ, although it could be a WAIT_ACQ
                        ; (at which point the dependency is not needed)
    AND R12, R12, #0    ; dependency to ensure order (only needed for a WAIT, not WAIT_ACQ)
    LDR R0, [R2, R12]   ; reads P1 data
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**AArch64**

P1

```
    STR W0, [X2]          ; sets new data
                          ; this does not have to be a Store-Release, though it
                          ; could be a Store-Release
```

P2

```
    WAIT ([X2]==1)        ; waits to see new data from P1
                          ; this does not need to be a WAIT_ACQ, though it could be a WAIT_ACQ
    STLR W0, [X3]         ; sends flag
                          ; this must be after the new data has been seen by P2
                          : as stores must not be speculative,
                          ; as a Store-Release, this orders the P1 store
```

P3

```
    WAIT([X3]==1)         ; waits for P2 flag
                          ; this does not need to be a WAIT_ACQ, though it could be a WAIT_ACQ
                          ; at which point the dependency is not needed
    AND W12, W12, WZR     ; dependency to ensure order
                          ; only needed for a WAIT, not WAIT_ACQ
    LDR W0, [X2, X12]     ; reads P1 data
```

In this case, P3:R0 == 0 is not permissible. P3 is guaranteed to see the store from P1 if P2 has seen the store from P1 before it does the Store-Release that is then seen by P3.

**Note**

The use of dependency by P3 could be replaced by a Load-Acquire.

### K10.2.4 Multiple observers of writes to multiple locations

The ARM weakly consistent memory model means that different observers can observe writes to different locations in different orders as was shown in *Multiple observers of writes to multiple locations* on page K10-5636, but the use of Load-Acquire and Store-Release can resolve this. In this case, the loads by P3 and P4 must be Load-Acquire in order to ensure the perceived multi-copy atomicity of the stores:

**AArch32**

P1

```
    STL R0, [R1]          ; sets new data
```

P2

```
    STL R0, [R2]          ; sets new data
```

P3

```
    LDA R10, [R2]         ; reads P2 data before P1
    LDA R9, [R1]          ;
    BIC R9, R10, R9       ; R9  <- R10 & ~R9
                          ; R9 contains 1 if read from [R2] is observed to be 1 and
                          ; read from [R1] is observed to be 0
```

P4

```
    LDA R9, [R1]
    LDA R10, [R2]
    BIC  R9, R9, R10      ; R9 <- R9 & ~R10
                          ; R9 contains 1 if read from [R2] is observed to be 0 and
                          ; read from [R1] is observed to be 1
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**AArch64**

P1

```
    STLR W0, [X1]         ; sets new data
```

P2

```
    STLR W0, [X2]         ; sets new data
```

P3

```
    LDAR W10, [X2]        ; reads P2 data before P1
    LDAR W9, [X1]         ;
    BIC W9, W10, W9       ; W9  <- W10 & ~W9
                          ; W9 contains 1 if read from [X2] is observed to be 1 and
                          ; read from [X1] is observed to be 0
```

P4

```
    LDAR W9, [X1]
    LDAR W10, [X2]
    BIC  W9, W9, W10      ; W9 <- W9 & ~W10
                          ; W9 contains 1 if read from [X2] is observed to be 0 and
                          ; read from [X1] is observed to be 1
```

In this case, the result P3:R9==1 and P4:R9==1 is not permissible, as the stores from P1 and P2 are multi-copy atomic when read by Load-Acquire.

Therefore, if P3 gets R10==1, then we know that the P3 load of R9 can only be observed after we know that P4 has also observed the P2 store to [R2]. Similarly, if the P4 load of R9 returns 1, and the P3 load of R9 returns 0, then the P3 load must have occurred before the P4 load.

Therefore, if the P3 load of R10 returns 1 and the P3 load of R9 returns 0, then we know that if the P4 load of R9 returns 1, it must have happened after P4 has observed the P2 store to [R2], so the P4 load of R10 must return 1.

This shows that, of the 4 possible values for {P3:R9, P4:R9}, the use of these instructions makes the result {1,1} impossible.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### K10.2.5 WFE and WFI and barriers

The Wait For Event and Wait For Interrupt instructions permit the PE to suspend execution and enter a low-power state. An explicit `DSB` barrier instruction is required if it is necessary to ensure memory accesses made before the `WFI` or `WFE` are visible to other observers, unless some other mechanism has ensured this visibility. Examples of other mechanism that would guarantee the required visibility are the `DMB` described in *Posting a store before polling for acknowledgement* on page K10-5638, or a dependency on a load.

The following example requires the `DSB` to ensure that the store is visible:

**AArch32**

P1

```
    STR R0, [R2]
    DSB
Loop
    WFI
    B Loop
```

**AArch64**

P1

```
    STR W0, [X2]
    DSB <domain>
Loop
    WFI
    B Loop
```

This requirement is unchanged in ARMv8 by the presence of Load-Acquire or Store-Release.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K10.3 Load-Acquire Exclusive, Store-Release Exclusive and barriers

The ARMv8 architecture adds the acquire and release semantics to Load-Exclusive and Store-Exclusive instructions, which allows them to gain ordering acquire and/or release semantics.

The Load-Exclusive instruction can be specified to have acquire semantics, and the Store-Exclusive instruction can be specified to have release semantics. These can be arbitrarily combined to allow the atomic update created by a successful Load-Exclusive and Store-Exclusive pair to have any of:

- No Ordering semantics (using `LDREX` and `STREX`).
- Acquire only semantics (using `LDAEX` and `STREX`).
- Release only semantics (using `LDREX` and `STLEX`).
- Sequentially consistent semantics (using `LDAEX` and `STLEX`).

In addition, the ARMv8 specification requires that the clearing of a global monitor will generate an event for the PE associated with the global monitor, which can simplify the use of `WFE`, by removing the need for a `DSB` barrier and `SEV` instruction.

### K10.3.1 Acquiring a lock

A common use of Load-Exclusive and Store-Exclusive instructions is to claim a lock to permit entry into a critical region. This is typically performed by testing a lock variable that indicates 0 for a free lock and some other value, commonly 1 or an identifier of the process holding the lock, for a taken lock.

For a critical region, the requirement on taking a lock is usually for acquire semantics, while the clearing of a lock requires release semantics:

**AArch32**

Px

```
    PLDW[R1]              ; preload into cache in unique state
Loop
    LDAEX R5, [R1]        ; read lock with acquire
    CMP R5, #0            ; check if 0
    STREXEQ R5, R0, [R1]  ; attempt to store new value
    CMPEQ R5, #0          ; test if store suceeded
    BNE Loop              ; retry if not

    ; loads and stores in the critical region can now be performed
```

**AArch64**

Px

```
    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop
    LDAXR W5, [X1]        ; read lock with acquire
    CBNZ W5, Loop         ; check if 0
    STXR W5, W0, [X1]     ; attempt to store new value
    CBNZ W5, Loop         ; test if store succeeded and retry if not

    ; loads and stores in the critical region can now be performed
```

The acquire associated with the load is sufficient to ensure the required ordering in a lock situation. The Store-Exclusive will fail (and so be retried) if there is a store to the location being monitored between the Load-Exclusive and the Store-Exclusive.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### K10.3.2 Releasing a lock

The converse operation of releasing a lock does not require the use of Load-Exclusive and Store-Exclusive instructions, because only a single observer is able to write to the lock. However, often it is necessary for any observer to observe any memory updates, or any values that are loaded into memory, before they observe the release of the lock. Therefore, the lock release needs release semantics:

**AArch32**

Px

```
    ; loads and stores in the critical region
    MOV R0, #0
    STL R0, [R1]          ; clear the lock with release semantics
```

**AArch64**

Px

```
    ; loads and stores in the critical region
    STLR WZR, [X1]        ; clear the lock with release semantics
```

### K10.3.3 Ticket locks

When a lock is free, in order to avoid a rush to get the lock by many PEs, the use of ticket locks is common in more advanced systems. When the use is requested, the ticket locks determine the order of the users of the critical sections, in order to avoid starvation that can occur with a simple contention based spin lock.

A ticket lock allocates each thread a ticket number when it first requests the lock, and then compares that number with the current number for the lock. If they are the same, then the critical section can be entered. Otherwise the thread waits until the current number is equal to the ticket number for that thread.

The reading of the current number of the lock needs acquire semantics for the lock to be acquired.

**Note**

The code in this section is little-endian code, as it views the combined current and next values as a single combined quantity. The addresses of the current and next ticket values need to be adjusted for a big-endian system.

This is shown in the implementation below:

**AArch32**

Px

```
    ; R1 holds two 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PLDW[R1]              ; preload into cache in unique state
Loop1
    LDAEX R5, [R1]        ; read current and next
    ADD R5, R5, #0x10000  ; increment the next number
    STREX R6, R5, [R1]    ; and update the value
    CMP R6, #0            ; did the exclusive pass
    BNE Loop1             ; retry if not
    CMP R5, R5, ROR #16   ; is the current ticket ours
    BEQ block_start
Loop2
    LDAH R6, [R1]         ; read current value
    CMP R6, R5, LSR #16   ; compare it with our allocated ticket
    BNE Loop2             ; retry (spin) if it is not the same
block_start
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**AArch64**

Px

```
    ; X1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop1
    LDAXR W5, [X1]        ; read current and next
    ADD W5, W5, #0x10000  ; increment the next number
    STXR W6, W5, [X1]     ; and update the value
    CBNZ W6, Loop1        ; did the exclusive pass - retry if not

    AND W6, W5, #0xFFFF
    CMP W6, W5, LSR #16   ; is the current ticket ours
    B.EQ block_start
Loop2
    LDARH W6, [X1]        ; read current value
    CMP W6, W5, LSR #16   ; compare it with the our allocated ticket
    B.NE Loop2            ; retry (spin) if it isn't the same
block_start
```

Releasing the ticket lock simply involves incrementing the current ticket number, that is still assumed to be in R3, and doing a Store-Release:

**AArch32**

```
    ADD R6, R6, #1
    STLH R6, [R1]
```

**AArch64**

```
    ADD W6, W6, #1
    STLRH W6, [X1]
```

### K10.3.4 Use of Wait For Event (WFE) and Send Event (SEV) with locks

The ARMv8 architecture can use the Wait For Event mechanism to minimise the energy cost of polling variables by putting the PE into a low power state, suspending execution, until an asynchronous exception or an explicit event is seen by that PE. In ARMv8, the event can be generated as a result of clearing the global monitor, so removing the need for a `DSB` barrier or an explicit send event message.

This can be used with simple locks or with ticket locks.

#### Simple lock

The following is an example of lock acquire code using `WFE`:

**AArch32**

Px

```
    PLDW[R1]              ; preload into cache in unique state
Loop
    LDAEX R5, [R1]        ; read lock with acquire
    CMP R5, #0            ; check if 0
    WFENE                 ; sleep if the lock is held
    STREXEQ R5, R0, [R1]  ; attempt to store new value
    CMPEQ R5, #0          ; test if store succeeded
    BNE Loop              ; retry if not
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**AArch64**

Px

```
    SEVL                ; invalidates the WFE on the first loop iteration
    PRFM PSTL1KEEP, [X1]  ; allocate into cache in unique state
Loop
    WFE
    LDAXR W5, [X1]        ; read lock with acquire
    CBNZ W5, Loop         ; check if 0
    STXR W5, W0, [X1]     ; attempt to store new value
    CBNZ W5, Loop         ; test if store succeeded and retry if not

    ; loads and stores in the critical region can now be performed
```

And the following is an example of lock release code:

**AArch32**

Px

```
    ; loads and stores in the critical region
    MOV R0, #0
    STL R0, [R1]          ; clear the lock
```

**AArch64**

Px

```
    ; loads and stores in the critical region
    STLR WZR, [X1]        ; clear the lock
```

## Ticket lock

In the Ticket lock case, the Load-Exclusive instruction can be used to move the monitor into the exclusive state for the express purpose of creating an event when the monitor changes state:

**AArch32**

Px

```
    ; R1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PLDW[R1]              ; preload into cache in unique state
Loop1
    LDAEX R5, [R1]        ; read current and next
    ADD R5, R5, #0x10000  ; increment the next number
    STREX R6, R5, [R1]    ; and update the value
    CMP R6, #0            ; did the exclusive pass
    BNE Loop              ; retry if not
    CMP R5, R5, ROR #16   ; is the current ticket ours
    BEQ block_start
    SEVL
Loop2
    WFE                   ; wait if there has not been a change to the count since last
                          ; read
    LDAEXH R6, [R1]       ; check the current count
    CMP R6, R5, LSR #16   ; check if it is equal
    BNE Loop2
block_start
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**AArch64**

Px

```
    ; X1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop1
    LDAXR W5, [X1]        ; read current and next
    ADD W5, W5, #0x10000  ; increment the next number
    STXR W6, W5, [X1]     ; and update the value
    CBNZ W6, Loop1        ; did the exclusive pass - retry if not

    AND W6, W5, 0xFFFF
    CMP W6, W5, LSR #16   ; is the current ticket ours
    B.EQ block_start
    SEVL
Loop2
    WFE
    LDAXRH W6, [X1]       ; read current value
    CMP W6, W5, LSR #16   ; compare it with our allocated ticket
    B.NE Loop2            ; retry (spin) if it is not the same
block_start
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K10.4 Using a mailbox to send an interrupt

In some message passing systems, it is common for one observer to update memory and then notify a second observer of the update by sending an interrupt, using a mailbox.

Although a memory access might be made to initiate the sending of the mailbox interrupt, a `DSB` instruction is required to ensure the completion of previous memory accesses.

Therefore, the following sequence is required to ensure that P2 observes the updated value:

**AArch32**

P1

```
STR R5, [R1]          ; message stored to shared memory location
DSB ST
STR R0, [R4]          ; R4 contains the address of a mailbox
```

P2

```
; interrupt service routine
LDR R5, [R1]
```

**AArch64**

P1

```
STR W5, [X1]          ; message stored to shared memory location
DSB ST
STR W0, [X4]          ; R4 contains the address of a mailbox
```

P2

```
; interrupt service routine
LDR W5, [X1]
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K10.5 Cache and TLB maintenance instructions and barriers

The following sections describe the use of barriers with cache and TLB maintenance instructions:

- *Data cache maintenance instructions*.
- *Instruction cache maintenance instructions* on page K10-5626.
- *TLB maintenance instructions and barriers* on page K10-5629.

### K10.5.1 Data cache maintenance instructions

The following sections describe the use of barriers with data cache maintenance instructions:

- *Message passing to non-caching observers*.
- *Multiprocessing message passing to non-caching observers* on page K10-5623.
- *Invalidating DMA buffers, non-functional example* on page K10-5624.
- *Invalidating DMA buffers, functional example with single PE* on page K10-5624.
- *Invalidating DMA buffers, functional example with multiple coherent PEs* on page K10-5625.

#### Message passing to non-caching observers

The ARMv8 architecture requires the use of `DMB` instructions to ensure the ordering of data cache maintenance instructions and their effects. The Load-Acquire and Store-Release instructions have no effect on cache maintenance instruction. This means the following message passing approaches can be used when communicating between caching observers and non-caching observers:

**AArch32**

P1

```
STR R5, [R1]          ; updates data (assumed to be in P1 cache)
DCCMVAC R1            ; cleans cache to point of coherency
DMB                   ; ensures effects of the clean will be observed before the
                      ; flag is set
STR R0, [R4]          ; sends flag to external agent (Non-cacheable location)
```

E1

```
WAIT_ACQ ([R4] == 1)  ; waits for the flag (with order)
LDR R5, [R1]          ; reads the data
```

**AArch64**

P1

```
STR W5, [X1]          ; updates data (assumed to be in P1 cache)
DC CVAC, X1           ; cleans cache to point of coherency
DMB ISH               ; ensures effects of the clean will be observed before the
                      ; flag is set
STR W0, [X4]          ; sends flag to external agent (Non-cacheable location)
```

E1

```
WAIT_ACQ ([X4] == 1)  ; waits for the flag (with order)
LDR W5, [X1]          ; reads the data
```

In this example, it is required that E1:R5==`0x55`.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Multiprocessing message passing to non-caching observers

The broadcast nature of the cache maintenance instructions combined with properties of barriers, means that the message passing principle for non-caching observers is:

**AArch32**

P1

```
STR R5, [R1]          ; updates data (assumed to be in P1 cache)
STL R0, [R2]          ; sends a flag for P2 (ordered by the store release)
```

P2

```
WAIT ([R2] == 1)      ; waits for P1 flag
DMB                   ; ensures cache clean is observed after P1 flag is observed
DCCMVAC R1            ; cleans cache to point of coherency - will clean P1 cache
DMB                   ; ensures effects of the clean will be observed before the
                      ; flag to E1 is set
STR R0, [R4]          ; sends flag to E1
```

E1

```
WAIT_ACQ ([R4] == 1)      ; waits for P2 flag (ordered)
LDR R5, [R1]              ; reads data
```

**AArch64**

P1

```
STR W5, [X1]          ; updates data (assumed to be in P1 cache)
STLR W0, [X2]         ; sends a flag for P2 (ordered)
```

P2

```
WAIT ([X2] == 1)      ; waits sfor P1 flag
DMB SY                ; ensure cache clean is observed after P1 flag is observed
DC CVAC, X1           ; cleans cache to point of coherency, will clean P1 cache
DMB SY                ; ensures effects of the clean will be observed before the
                      ; flag to E1 is set
STR W0, [X4]          ; sends flag to E1
```

E1

```
WAIT_ACQ ([X4] == 1)      ; waits for P2 flag
LDR W5, [X1]              ; reads data
```

In this example, it is required that E1:R5==`0x55`. The clean operation executed by P2 affects the data location in the P1 cache. The cast-out from the P1 cache is guaranteed to be observed before P2 updates [R4].

**Note**

The cache maintenance instructions are not ordered by the Load-Acquire and Store-Release instructions.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Invalidating DMA buffers, non-functional example

The basic scheme for communicating with an external observer that is a process that passes data in to a Cacheable memory region must take account of the architectural requirement that regions marked as Cacheable can be allocated into a cache at any time, for example as a result of speculation. The following example shows this possibility:

**AArch32**

P1

```
    DCIMVAC R1            ; ensures cache is not dirty. A clean operation could be used
                          ; but as the DMA will subsequently overwrite this region an
                          ; invalidate operation is sufficient and usually more efficient
    DMB                   ; ensures cache invalidation is observed before the next store
                          ; is observed
    STR R0, [R3]          ; sends flag to external agent
    WAIT_ACQ ([R4]==1)    ; waits for a different flag from an external agent
    LDR R5, [R1]
```

E1

```
    WAIT ([R3] == 1)      ; waits for flag
    STR R5, [R1]          ; stores new data
    STL R0, [R4]          ; sends a flag
```

**AArch64**

P1

```
    DC IVAC, X1           ; ensure cache is not dirty. A clean operation could be used
                          ; but as the DMA will subsequently overwrite this region an
                          ; invalidate operation is sufficient and usually more efficient
    DMB SY                ; ensures cache invalidation is observed before the next store
                          ; is observed
    STR W0, [X3]          ; sends flag to external agent
    WAIT_ACQ ([X4]==1)    ; waits for a different flag from an external agent
    LDR W5, [X1]
```

E1

```
    WAIT ([X3] == 1)      ; waits for flag
    STR W5, [X1]          ; stores new data
    STLR W0, [X4]         ; sends a flag
```

If a speculative access occurs, there is no guarantee that the cache line containing [R1] is not brought back into the cache after the cache invalidation, but before [R1] is written by E1. Therefore, the result P1:R5=0 is permissible.

### Invalidating DMA buffers, functional example with single PE

**AArch32**

P1

```
    DCIMVAC R1            ; ensures cache is not dirty. A clean operation could be used
                          ; but as the DMA will subsequently overwrite this region an
                          ; invalidate operation is sufficient and usually more efficient
    DMB                   ; ensures cache invalidation is observed before the next store
                          ; is observed
    STR R0, [R3]          ; sends flag to external agent
    WAIT ([R4]==1)        ; waits for a different flag from an external agent
    DMB                   ; ensures that cache invalidate is observed after the flag
                          ; from external agent is observed
    DCIMVAC R1            ; ensures cache discards stale copies before use
    LDR R5, [R1]
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

E1

```
WAIT ([R3] == 1)      ; waits for flag

STR R5, [R1]          ; stores new data

STL R0, [R4]          ; sends a flag
```

**AArch64**

P1

```
DC IVAC, X1           ; ensures cache is not dirty. A clean operation could be used
                      ; but as the DMA will subsequently overwrite this region an
                      ; invalidate operation is sufficient and usually more efficient
DMB SY                ; ensures cache invalidation is observed before the next store
                      ; is observed
STR W0, [X3]          ; sends flag to external agent
WAIT ([X4]==1)        ; waits for a different flag from an external agent
DMB SY                ; ensures that cache invalidate is observed after the flag
                      ; from external agent is observed
DC IVAC, X1           ; ensures cache discards stale copies before use
LDR W5, [X1]
```

E1

```
WAIT ([X3] == 1)      ; waits for flag
STR W5, [X1]          ; stores new data
STLR W0, [X4]         ; sends a flag
```

In this example, the result P1:R5 == `0x55` is required. Including a cache invalidation after the store by E1 to [R1] is observed ensures that the line is fetched from external memory after it has been updated.

### Invalidating DMA buffers, functional example with multiple coherent PEs

The broadcasting of cache maintenance instructions, and the use of `DMB` instructions to ensure their observability, means that the previous example extends naturally to a multiprocessor system. Typically this requires a transfer of ownership of the region that the external observer is updating.

**AArch32**

P0

```
(Use data from [R1], potentially using [R1] as scratch space)
STL R0, [R2]          ; signals release of [R1]
WAIT_ACQ ([R2] == 0)  ; waits for new value from DMA
LDR R5, [R1]
```

P1

```
WAIT ([R2] == 1)      ; waits for release of [R1] by P0
DCIMVAC R1            ; ensures caches are not dirty, an invalidate is sufficient
DMB
STR R0, [R3]          ; requests new data for [R1]
WAIT ([R4] == 1)      ; waits for new data
DMB
DCIMVAC R1            ; ensures caches discard stale copies before use
DMB
MOV R0, #0
STR R0, [R2]          ; signals availability of new [R1]
```

E1

```
WAIT ([R3] == 1)      ; waits for new data request
STR R5, [R1]          ; sends new [R1]
DMB [ST]
STR R0, [R4]          ; indicates that new data is available to P1
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**AArch64**

P0

```
(Use data from [X1], potentially using [X1] as scratch space)
STLR W0, [X2]         ; signals release of [X1]
WAIT_ACQ ([X2] == 0)  ; waits for new value from DMA
LDR W5, [X1]
```

P1

```
WAIT ([X2] == 1)      ; waits for release of [R1] by P0
DC IVAC, X1           ; ensures caches are not dirty, an invalidate is sufficient
DMB SY
STR W0, [X3]          ; requests new data for [R1]
WAIT ([X4] == 1)      ; waits for new data
DMB SY
DCIMVAC X1            ; ensures caches discard stale copies before use
DMB SY
STR WZR, [X2]         ; signals availability of new [R1]
```

E1

```
WAIT ([X3] == 1)      ; waits for new data request
STR W5, [X1]          ; sends new [R1]
STR W0, [X4]          ; indicates new data is available to P1
```

In this example, the result P0:R5 == `0x55` is required. The `DMB` issued by P1 after the first data cache invalidation ensures that effect of the cache invalidation on P0 is seen by E1 before the store by E1 to [R1]. The `DMB` issued by P1 after the second data cache invalidation ensures that its effects are seen before the store of 0 to the semaphore location in [R2].

## K10.5.2 Instruction cache maintenance instructions

The following sections describe the use of barriers with instruction cache maintenance instructions:

- *Ensuring the visibility of updates to instructions for a uniprocessor*.
- *Ensuring the visibility of updates to instructions for a multiprocessor* on page K10-5627.

### Ensuring the visibility of updates to instructions for a uniprocessor

On a single PE, the agent that causes instruction fetches, or instruction cache linefills, is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the instruction cache can rely only on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

Also, instruction cache maintenance instructions are only guaranteed to complete after the execution of a `DSB`, and an `ISB` is required to discard any instructions that might have been prefetched before the instruction cache invalidation completed. Therefore, on a uniprocessor, to ensure the visibility of an update to code and to branch to it, the following sequence is required:

**AArch32**

P1

```
STR R11, [R1]         ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB
ICIMVAU R1            ; ensures instruction cache/branch predictor discards stale data
BPIMVA R1
DSB                   ; ensures completion of the invalidation
ISB                   ; ensures instruction fetch path sees new instruction cache state
BX R1
```

In AArch64, the branch predictor maintenance is not required.

**AArch64**


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

P1

```
STR W11, [X1]         ; W11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to instruction cache
DSB ISH
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DSB ISH               ; ensures completion of the invalidation
ISB                   ; ensures instruction fetch path sees new instruction cache state
BR X1
```

**Note**

Where the changes to the instructions span multiple cache lines, then the data cache and instruction cache maintenance instructions can be duplicated to cover each of the lines to be cleaned and to be invalidated.

### Ensuring the visibility of updates to instructions for a multiprocessor

The ARMv8 architecture requires a PE that executes an instruction cache maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the cache maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

An `ISB` is not broadcast, and so does not affect other PEs. This means that any other PE must perform its own `ISB` synchronization after it knows that the update is visible, if it is necessary to ensure its synchronization with the update. The following example shows how this might be done:

**AArch32**

P1

```
STR R11, [R1]         ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB                   ; ensures completion of the clean on all PEs
ICIMVAU R1            ; ensures instruction cache discards stale data
BPIMVA R              ; ensures branch predictor discards stale data
DSB                   ; ensures completion of the instruction cache and branch predictor
                      ; invalidation on all PEs
STR R0, [R2]          ; sets flag to signal completion
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)      ; waits for flag signalling completion
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```

**AArch64**

P1

```
STR X11, [X1]         ; X11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to the instruction cache
DSB ISH               ; ensures completion of the clean on all PEs
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DSB ISH               ; ensures completion of the instruction cache/branch predictor
                      ; invalidation on all PEs
STR W0, [X2]          ; sets flag to signal completion
ISB                   ; synchronizes context on this PE
BR R1                 ; branches to new code
```

P2-P*x*

```
WAIT ([X2] == 1)      ; waits for flag signalling completion
ISB                   ; synchronizes context on this PE
BR X1                 ; branches to new code
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### *Nonfunctional approach*

The following sequence does not have the same effect, because a DSB is not required to complete the instruction cache maintenance instructions that other PEs issue:

**AArch32**

P1

```
STR R11, [R1]          ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1             ; clean to PoU makes the new instruction visible to the instruction cache
DSB                    ; ensures completion of the clean on all PEs
ICIMVAU R1             ; ensures instruction cache discards stale data
BPIMVA R1              : ensures branch predictor discards stale data
DMB                    ; ensures ordering of the store after the invalidation
                       ; DOES NOT guarantee completion of instruction cache/branch
                       ; predictor on other PEs
STR R0, [R2]           ; sets flag to signal completion
DSB                    ; ensures completion of the invalidation on all PEs
ISB                    ; synchronizes context on this PE
BX R1                  ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)       ; waits for flag signalling completion
DSB                    ; this DSB does not guarantee completion of P1
                       ; ICIMVAU/BPIMVA
ISB
BX R1
```

**AArch64**

P1

```
STR W11, [X1]          ; W11 contains a new instruction to be stored in program memory
DC CVAU, X1            ; clean to PoU makes the new instruction visible to instruction cache
DSB ISH                ; ensures completion of the clean on all PEs
IC IVAU, X1            ; ensures instruction cache/branch predictor discards stale data
DMB ISH                ; ensures ordering of the store after the invalidation
                       ; DOES NOT guarantee completion of instruction cache/branch
                       ; predictor on other PEs
STR W0, [X2]           ; sets flag to signal completion
DSB ISH                ; ensures completion of the invalidation on all PEs
ISB                    ; synchronizes context on this PE
BR X1                  ; branches to new code
```

P2-P*x*

```
WAIT ([X2] == 1)       ; waits for flag signalling completion
DSB ISH                ; this DSB does not guarantee completion of P1
                       ; ICIMVAU/BPIMVA
ISB
BR X1
```

In this example, P2…P*x* might not see the updated region of code at R1.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### K10.5.3 TLB maintenance instructions and barriers

The following sections describe the use of barriers with TLB maintenance instructions:

- *Ensuring the visibility of updates to translation tables for a uniprocessor*.
- *Ensuring the visibility of updates to translation tables for a multiprocessor* on page K10-5630.
- *Paging memory in and out* on page K10-5630.

#### Ensuring the visibility of updates to translation tables for a uniprocessor

On a single PE, the agent that causes translation table walks is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the TLB can only rely on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

The ARMv8 architecture requires that translation table walks look in the data or unified caches at L1, so such systems do not require data cache cleaning.

After the translation tables update, any old copies of entries that might be held in the TLBs must be invalidated. This operation is only guaranteed to affect all instructions, including instruction fetches and data accesses, after the execution of a `DSB` and an `ISB`. Therefore, the code for updating a translation table entry is:

**AArch32**

P1

```
STR R11, [R1]         ; updates the translation table entry
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVA R10
BPIALL
DSB                   ; ensures completion of the BP and TLB invalidation
ISB                   ; synchronises context on this PE
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using
; the old mapping have been completed
```

**AArch64**

P1

```
STR X11, [X1]         ; updates the translation table entry
DSB ISH               ; ensures visibility of the update to translation table walks
TLBI VAE1, X10        ; assumes we are in the EL1
DSB ISH               ; ensures completion of the TLB invalidation
ISB                   ; synchronise context on this PE
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using
; the old mapping have been completed
```

Importantly, by the end of this sequence, all accesses that used the old translation table mappings have been observed by all observers.

An example of this is where a translation table entry is marked as invalid. Such a system must provide a mechanism to ensure that any access to a region of memory being marked as invalid has completed before any action is taken as a result of the region being marked as invalid.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Ensuring the visibility of updates to translation tables for a multiprocessor

The same code sequence can be used in a multiprocessing system. The ARMv8 architecture requires a PE that executes a TLB maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the TLB maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

The completion of a `DSB` that completes a TLB maintenance instruction ensures that all accesses that used the old mapping have completed.

**AArch32**

P1

```
STR R11, [R1]         ; updates the translation table entry
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVAIS R10
BPIALLIS
DSB                   ; ensures completion of the BP and TLB invalidation
ISB                   ; Note ISB is not broadcast and must be executed locally
                      ; on other PEs
;  new translation table entry can be relied upon at this point and all accesses
;  generated by any observers affected by the broadcast TLBIMVAIS operation using
;  the old mapping have been completed
```

**AArch64**

P1

```
STR X11, [X1]         ; updates the translation table entry
DSB ISH               ; ensures visibility of the update to translation table walks
TLBI VAE1IS,  X10
DSB ISH               ; ensures completion of the TLB invalidation
ISB                   ; Note ISB is not broadcast and must be executed locally
                      ; on other PEs
;  new translation table entry can be relied upon at this point and all accesses
;  generated by any observers affected by the broadcast TLBIMVAIS operation using
;  the old mapping have been completed
```

The completion of the TLB maintenance instruction is guaranteed only by the execution of a `DSB` by the observer that performed the TLB maintenance instruction. The execution of a `DSB` by a different observer does not have this effect, even if the `DSB` is known to be executed after the TLB maintenance instruction is observed by that different observer.

### Paging memory in and out

In a multiprocessor system there is a requirement to ensure the visibility of translation table updates when paging regions of memory into RAM from a backing store. This might, or might not, also involve paging existing locations in memory from RAM to a backing store. In such situations, the operating system selects one or more pages of memory that might be in use but are suitable to discard, with or without copying to a backing store, depending on whether or not the region of memory is writable. Disabling the translation table mappings for a page, and ensuring the visibility of that update to the translation tables, prevents agents accessing the page.

For this reason, it is important that the `DSB` that is performed after the TLB invalidation ensures that no other updates to memory using those mappings are possible.

An example sequence for the paging out of an updated region of memory, and the subsequent paging in of memory, is as follows:

**AArch32**

P1

```
STR R11, [R1]         ; updates the translation table for the region being paged out
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVAIS R10         ; invalidates the old entry
DSB                   ; ensures completion of the invalidation on all PEs
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
ISB                  ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB                  ; ensures completion of the memory transfer (this could be part of
                     ; LoadMemoryFromBackingStore
ICIALLUIS            ; also invalidates the branch predictor
STR R9, [R1]         ; creates a new translation table entry with a new mapping
DSB                  ; ensures completion of the instruction cache
                     ; and branch predictor invalidation
                     ; AND ensures visibility of the new translation table mapping
ISB                  ; ensures synchronisation of this instruction stream
```

**AArch64**

P1

```
STR X11, [X1]        ; updates the translation table for the region being paged out
DSB ISH              ; ensures visibility of the update to translation table walks
TLBI VAE1IS, X10     ; invalidates the old entry
DSB ISH              ; ensures completion of the invalidation on all PEs
ISB                  ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB ISH              ; ensures completion of the memory transfer (this could be part of
                     ; LoadMemoryFromBackingStore
IC IALLUIS           ; also invalidates the branch predictor
STR X9, [X1]         ; creates a new translation table entry with a new mapping
DSB ISH              ; ensures completion of the instruction cache
                     ; and branch predictor invalidation
                     ; AND ensures visibility of the new translation table mapping
ISB                  ; ensures synchronisation of this instruction stream
```

This example assumes the memory copies are performed by an observer that is coherent with the caches of PE P1. This observer might be P1 itself, using a specific paging mapping. For clarity, the example omits the functional descriptions of `SaveMemoryPageToBackingStore` and `LoadMemoryFromBackingStore`. `LoadMemoryFromBackingStore` is required to ensure that the memory updates that it makes are visible to instruction fetches.

In this example, the use of ICIALLUIS in AArch32 and IC IALLUIS in AArch64 to invalidate the entire instruction cache is a simplification, that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using ICIMVAU and IC IVAU operations in AArch32 and AArch64 respectively. This invalidation must be done when the mapping used for the ICIMVAU and IC IVAU operations is valid but not executable.

### K10.5.4 Ordering of Memory-mapped device control with payloads

With a Memory-mapped peripheral, such as a DMA, which can also access memory for its own use, it is common to have control or status registers which are Memory-mapped. These registers need to be accessed in an ordered manner with respect to the data that the Memory-mapped peripheral is handling.

Two simple examples of this are:

- When a processing element is writing a buffer of data, and then writing to a control register in the DMA peripheral to start that peripheral to access the buffer of data.
- When a DMA peripheral has written to a buffer of data in memory, and the processing element is reading a status register to determine that the DMA transfer has completed, and then is reading the data.

For the case of the processing element writing a buffer of data, before starting the DMA peripheral, the ordering requirements between the stores to the data buffer and the stores to the Memory-mapped a to the DMA peripheral can be met by the insertion of a `DSB <domain>` instruction between these sets of accesses as this ensures the global observation of the stores before the DMA is started. this is shown by the following code:


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**AArch32**

P1

```
STR R5, [R2]          ; data written to the data buffer

DSB

STR R0, [R4]          ; R4 contains the address of the DMA control register
```

**AArch64**

P1

```
STR W5, [X2]          ; data written to the data buffer
DSB <domain>
STR W0, [X4]          ; X4 contains the address of the DMA control register
```

For the case of DMA peripheral writing the data buffer and then setting a status register when those stores are complete (and so globally observed) and then having this status register polled by the processing element before the processing element reads the data buffer, the processing element must insert a `DSB <domain>` between the load that reads the status register, and the read of the buffer. A DMB, or load-acquire, is not sufficient as this problem is not solely concerned with observation order, since the polling read is actually a read of a status register at a slave, not the polling a data value that has been written by an observer.

For this case, the code is therefore:

**AArch32**

P1

```
WAIT ([R4] == 1)      ; R4 contains the address of the status register,
                      ; and the value '1' indicates completion of the DMA transfer
DSB
LDR R5, [R2]          ; reads data from the data buffer
```

**AArch64**

P1

```
WAIT ([X4] == 1)      ; X4 contains the address of the status register,
                      ; and the value '1' indicates completion of the DMA transfer
DSB <domain>
LDR W5, [X2]          ; reads data from the data buffer
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K10.6 ARMv7 compatible approaches for ordering, using DMB and DSB barriers

The following sections describe the ARMv7 compatible approaches for ordering, using DMB and DSB barriers:

- *Simple ordering and barrier cases*.
- *Load-Exclusive, Store-Exclusive and barriers* on page K10-5639.
- *Using a mailbox to send an interrupt* on page K10-5641.
- *Cache and TLB maintenance instructions and barriers* on page K10-5641.

### K10.6.1 Simple ordering and barrier cases

ARM implements a weakly consistent memory model for Normal memory. In general terms, this means that the order of memory accesses observed by other observers might not be the order that appears in the program, for either loads or stores.

This section includes examples of this.

#### Simple weakly consistent ordering example

P1

```
STR R5, [R1]
LDR R6, [R2]
```

P2

```
STR R6, [R2]
LDR R5, [R1]
```

In the absence of barriers, the result of P1: R6=0, P2: R5=0 is permissible.

#### Message passing

The following sections describe:

- *Weakly-ordered message passing problem*.
- *Message passing with multiple observers* on page K10-5635.

##### *Weakly-ordered message passing problem*

P1

```
STR R5, [R1]            ; sets new data
STR R0, [R2]            ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1)           ; waits on flag
LDR R5, [R1]            ; reads new data
```

In the absence of barriers, an end result of P2: R5=0 is permissible.

##### *Resolving by the addition of barriers*

The addition of barriers, to ensure the observed order of the reads and the writes, ensures that data is transferred so that the result P2:R5==0x55 is guaranteed, as follows:

P1

```
STR R5, [R1]            ; sets new data
DMB [ST]                ; ensures all observers observe data before the flag
STR R0, [R2]            ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1)           ; waits on flag
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
DMB                     ; ensures that the load of data is after the flag has been observed
LDR R5, [R1]
```

### *Resolving by the use of barriers and address dependency*

There is a rule within the ARM architecture that:

- Where the value returned by a read is used for computation of the virtual address of a subsequent read or write, then these two memory accesses are observed in program order.

  Where the value returned by a read is used for computation of the virtual address of a subsequent read or write, this is called an *address dependency*. An address dependency exists even if the value returned by the first read has no effect on the virtual address. This might occur if the value returned is masked off before it is used, or if it confirms a predicted address value that it might have changed.

  This restriction applies only when the data value returned by a read is used as a data value to calculate the address of a subsequent read or write. It does not apply if the data value returned by a read determines the condition flags values, and the values of the flags are used for condition code evaluation to determine the address of a subsequent read, either through conditional execution or the evaluation of a branch. This is called a *control dependency*.

  Where both a control and address dependency exist, the ordering behavior is consistent with the address dependency.

Table K10-1 shows examples of address dependencies, control dependencies, and an address and control dependency.

**Table K10-1 Dependency examples**

| **Address dependency** | | **Control dependency** | | **Address and control dependency**[a] |
|---|---|---|---|---|
| **(a)** | **(b)** | **(c)** | **(d)** | **(e)** |
| `LDR r1, [r0]` | `LDR r1, [r0]` | `LDR r1, [r0]` | `LDR r1, [r0]` | `LDR r1, [r0]` |
| `LDR r2, [r1]` | `AND r1, r1, #0` | `CMP r1, #55` | `CMP r1, #55` | `CMP r1, #0` |
| | `LDR r2, [r3, r1]` | `LDRNE r2, [r3]` | `MOVNE r4, #22` | `LDRNE r2, [r1]` |
| | | | `LDR r2, [r3, r4]` | |

a. The address dependency takes priority.

This means that the data transfer example of *Weakly-ordered message passing problem* on page K10-5633 can also be satisfied as shown in the following example:

P1

```
STR R5, [R1]            ; sets new data
DMB [ST]                ; ensures all observers observe data before the flag
STR R0, [R2]            ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1)
AND R12, R12, #0        ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12]       ; the load has an address dependency on R12
                        ; and so is ordered after the flag has been seen
```

The load of R5 by P2 is ordered with respect to the load from [R2] because there is an address dependency using R12. P1 uses a `DMB` to ensure that P2 does not observe the write of [R2] before the write of [R1].


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

#### Message passing with multiple observers

Where the ordering of Normal memory accesses is not resolved by the use of barriers or dependencies, then different observers might observe the accesses in a different order, as shown in the following example:

P1

```
STR R5, [R1]            ; sets new data
STR R0, [R2]            ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1)
AND R12, R12, #0        ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12]       ; the load has an address dependency on R12
                        ; and so is ordered after the flag has been seen
```

P3

```
WAIT([R2]==1)
AND R12, R12, #0        ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12]       ; the load is address depndent on R12
                        ; and so is ordered after the flag has been seen
```

In this case, it is permissible for P2:R5 and P3:R5 to contain different values, because there is no order guaranteed between the two stores performed by P1.

#### Resolving by the addition of barriers

The addition of a barrier by P1, as shown in the following example, ensures the observed order of the writes, transferring data so that P2:R5 and P3:R5 both contain the value `0x55`:

P1

```
STR R5, [R1]            ; sets new data
DMB [ST]                ; ensures all observers observe data before the flag
STR R0, [R2]            ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1)
AND R12, R12, #0        ; R12 is the destination of LDR in WAIT macro
LDR R5, [R1, R12]       ; the load has an address dependency on R12
                        ; and so is ordered after the flag has been seen
```

P3

```
WAIT([R2]==1)
AND R12, R12, #0        ; R12 is the destination of LDR in WAIT macro
LDR R5, [R1, R12]       ; the load has an address dependency on R12
                        ; and so is ordered after the flag has been seen
```

### Address dependency with object construction

When accessing an object-oriented data structure, the address dependency rule means that barriers are not required, even when initializing the object:

P1

```
STR R5, [R1, #offset]   ; sets new data in a field
DMB [ST]                ; ensures all observers observe data before base address is updated
STR R1, [R2]            ; updates base address
```

P2

```
LDR R1, [R2]            ; reads for base address
CMP R1, #0              ; checks if it is valid
BEQ null_trap
LDR R5, [R1, #offset]   ; uses base address to read field
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

If the null_trap is not taken, it is required that P2:R5==0x55. This avoids P2 observing a partially constructed object from P1. Significantly, P2 does not require a barrier to ensure this behavior.

P1 requires a barrier to ensure the observed order of the writes by P1. In general, the impact of requiring a barrier during the construction phase is much less than the impact of requiring a barrier for every read access.

### Causal consistency issues with multiple observers

The fact that different observers can observe memory accesses in different orders extends, in the absence of barriers, to behaviors that do not fit naturally expected causal properties, as the following example shows:

P1

```
    STR R0, [R2]            ; sets new data
```

P2

```
    WAIT([R2]==1)           ; waits to see new data from P1
    STR R0, [R3]            ; sends flag, must be after the new data has been by P2 as stores,
                            ; must not be speculative
```

P3

```
    WAIT([R3]==1)           ; waits for P2 flag
    AND R12, R12, #0        ; dependency to ensure order
    LDR R0, [R2, R12]       ; reads P1's data
```

In this example, P3:R0==0 is permissible. P3 is not guaranteed to seethe stores from P1 and P2 in any particular order. This applies despite the fact that the store from P2 can only happen after P2 has observed the store from P1.

This example shows that the ARM memory order model for Normal memory does not conform to c*ausal consistency*. This means that the apparently transitive causal relationship between two variables is not guaranteed to be transitive.

The following example shows the insertion of a barrier by P2 to create causal consistency:

P1

```
    STR R0, [R2]            ; sets new data
```

P2

```
    WAIT([R2]==1)           ; waits to see new data from P1
    DMB                     ; ensures P1 data is observed by all observers before any following store
    STR R0, [R3]            ; sends flag
```

P3

```
    WAIT([R3]==1)           ; waits for P2 flag
    AND R12, R12, #0        ; dependency to ensure order
    LDR R0, [R2, R12]       ; reads P1 data
```

This creates causal consistency because a DMB is required to order all accesses that the executing PE observed before the DMB, not only those it issued, before any of the accesses that follow the DMB.

### Multiple observers of writes to multiple locations

The ARM weakly consistent memory model means that different observers can observe writes to different locations in different orders, as the following example shows:

P1

```
    STR R0, [R1]            ; sets new data
```

P2

```
    STR R0, [R2]            ; sets new data
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

P3

```
    LDR R10, [R2]            ; reads P2 data before P1 data
    LDR R9, [R1]             ;
    BIC R9, R10, R9          ; R9 <- R10 && ~R9
                             ; R9 contains 1 iff read from [R2] is observed to be 1 and
                             ; read from [R1] is observed to be 0.
```

P4

```
    LDR R9, [R1]
    LDR R10, [R2]
    BIC  R9, R9, R10         ; R9 <- R9 && ~R10
                             ; R9 contains 1 iff read from [R2] is observed to be 0 and
                             ; read from [R1] is observed to be 1.
```

In this example, the result P3:R9==1 and P4:R9==1 is permissible. This means that P3 and P4 observed the stores from P1 and P2 in different orders.

The following example shows the use of `DMB` instructions to ensure sequential consistency:

P1

```
    STR R0, [R1]             ; sets new data
```

P2

```
    STR R0, [R2]             ; sets new data
```

P3

```
    LDR R10, [R2]            ; reads P2 data before P1 data
    DMB
    LDR R9, [R1]
    BIC R9, R10, R9          ; R9 <- R10 && ~R9
                             ; R9 contains 1 iff read from [R2] is observed to be 1 and
                             ; read from [R1] is observed to be 0.
```

P4

```
    LDR R9, [R1]             ; reads P1 data before P2 data
    DMB
    LDR R10, [R2]
    BIC  R9, R9, R10         ; R9 <- R9 && ~R10
                             ; R9 contains 1 iff read from [R2] is observed to be 0 and
                             ; read from [R1] is observed to be 1.
```

In this example:

- The `DMB` executed by P3 ensures that, if the P3 load from [R2] observes the P2 store to [R2], then all observers observe the P2 store to [R2] before they observe the P3 load from [R1].
- The `DMB` executed by P4 ensures that, if the P4 load from [R1] observes the P1 store to [R1], then all observers observe the P1 store to [R1 before they observe the P4 load from [R2].

If the P3 load from [R1] returns 0, then it has not observed the P1 store to [R1]. Also, if the P3 load of [R2] returns 1, then all observers must have observed the P2 store to [R2] before they observed the P1 store to [R1]. This means that P4 cannot observe the P1 store to [R1] without also observing the P2 store to [R2].

Alternatively, if the P4 load from [R2] returns 0, then it has not observed the P2 store to [R2]. If, also, the P4 load of [R1] returns 1, then all observers must have observed the P1 store to [R1] before they observed the P2 store to [R2]. This means that P3 cannot observe the P2 store to [R2] without also observing the P1 store to [R1].

This shows that, of the four possible results for {P3:R9, P4:R9}, the insertion of these barriers makes the result {1, 1} impossible.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Posting a store before polling for acknowledgement

In the case where an observer stores to a location, and then polls for an acknowledge from a different observer, the weak ordering of the memory model can lead to a deadlock, as the following example shows:

P1

```
STR R0, [R2]
WAIT ([R3]==1)
```

P2

```
WAIT ([R2]==1)
STR R0, [R3]
```

In ARMv7 implementations that do not include the Multiprocessing Extensions, then this can deadlock because P2 might not observe the store by P1 in finite time. For ARMv7 implementations with the Multiprocessing Extensions and for ARMv8, this is not an issue as all stores must be observed by all observers within their shareability domain in finite time.

The addition of a `DMB` instruction prevents this deadlock in ARMv7 implementations that do not include the Multiprocessing Extensions:

P1

```
STR R0, [R2]
DMB
WAIT ([R3]==1)
```

P2

```
WAIT ([R2]==1)
STR R0, [R3]
```

The `DMB` executed by P1 ensures that P2 observes the store by P1 before it observes the load by P1. This ensures a timely completion.

The following example is a variant of the previous example, where the two observers poll the same memory location:

P1

```
STR R0, [R2]
WAIT ([R2]==2)
```

P2

```
WAIT ([R2]==1)
LDR R0, [R2]
ADD R0, R0, #1
STR R0, [R2]
```

In this example, the same deadlock can occur in ARMv7 implementations that do not include the Multiprocessing Extensions, because the architecture permits P1 to read the result of its own store to [R2] early, and continue doing so for an indefinite amount of time. The addition of a `DMB` instruction prevents this deadlock:

P1

```
STR R0, [R2]
DMB
WAIT ([R2]==2)
```

P2

```
WAIT ([R2]==1)
LDR R0, [R2]
ADD R0, R0, #1
STR R0, [R2]
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### WFE and WFI and barriers

The Wait For Event and Wait For Interrupt instructions permit the PE to suspend execution and enter a low-power state. A DSB barrier instruction is required if it is necessary to ensure that memory accesses made before the WFI or WFE are visible to other observers, unless some other mechanism has ensured this visibility. Examples of other mechanism that would guarantee the required visibility are the DMB described in *Posting a store before polling for acknowledgement* on page K10-5638, or a dependency on a load.

The following example requires the DSB to ensure that the store is visible:

P1

```
    STR R0, [R2]
    DSB
Loop
    WFI
    B Loop
```

However, if the example in *Posting a store before polling for acknowledgement* on page K10-5638 is extended to include a WFE, there is no risk of a deadlock. The extended example is:

P1

```
    STR R0, [R2]
    DMB
Loop
    LDR R12, [R3]
    CMP R12, #1
    WFENE
    BNE Loop
```

P2

```
    WAIT ([R2]==1)
    STR R0, [R3]
    DSB
    SEV
```

In this example:

- The DMB by P1 ensures that P2 observes the store by P1 before it observes the load by P1.
- The dependency of the WFE on the result of the load by P1 means that this load must complete before P1 executes the WFE.

For more information about SEV, see *Use of Wait For Event (WFE) and Send Event (SEV) with locks* on page K10-5640.

## K10.6.2 Load-Exclusive, Store-Exclusive and barriers

The Load-Exclusive and Store-Exclusive instructions, described in *Synchronization and semaphores* on page B2-107, are predictable only with Normal memory. These instructions do not have any implicit barrier functionality. Therefore, any use of these instructions to implement locks of any type requires the addition of explicit barriers.

### Acquiring a lock

A common use of Load-Exclusive and Store-Exclusive instructions is to claim a lock to permit entry into a critical region. This is typically performed by testing a lock variable that indicates 0 for a free lock and some other value, commonly 1 or an identifier of the process holding the lock, for a taken lock.

The lack of implicit barriers in the Load-Exclusive and Store-Exclusive instructions means that the mechanism requires a DMB instruction between acquiring a lock and making the first access to the critical region, to ensure that all observers observe the successful claim of the lock before they observe any subsequent loads or stores to the region. This example shows Px acquiring a lock:


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

P*x*

```
Loop
    LDREX R5, [R1]           ; reads lock
    CMP R5, #0               ; checks if 0
    STREXEQ R5, R0, [R1]     ; attempts to store new value
    CMPEQ R5, #0             ; tests if store succeeded
    BNE Loop                 ; retries if not
    DMB                      ; ensures that all subsequent accesses are observed after the
                             ; gaining of the lock is observed
    ; loads and stores in the critical region can now be performed
```

### Releasing a lock

The converse operation of releasing a lock does not require the use of Load-Exclusive and Store-Exclusive instructions, because only a single observer is able to write to the lock. However, often it is necessary for any observer to observe any memory updates, or any values that are loaded into memory, before they observe the release of the lock. Therefore, a `DMB` usually precedes the lock release, as the following example shows.

P*x*

```
    ; loads and stores in the critical region
    MOV R0, #0
    DMB                      ; ensures all previous accesses are observed before the lock is cleared
    STR R0, [R1]             ; clears the lock
```

### Use of Wait For Event (WFE) and Send Event (SEV) with locks

The ARMv8 architecture includes Wait For Event and Send Event instructions, that can be executed to reduce the required number of iterations of a lock-acquire loop, or *spinlock*, to reduce power. The basic mechanism involves an observer that is in a spinlock executing a `WFE` instruction that suspends execution on that observer until an asynchronous exception or an explicit event, sent by some other observer using the `SEV` instruction, is seen by the suspended observer. An observer that holds the lock executes an `SEV` instruction to send an event after it has released the lock.

The Event signal is a non-memory communication, and therefore the memory update that releases the lock must be observable by all observers before the `SEV` instruction is executed and the event is sent. This requires the use of `DSB` instruction, rather than `DMB`.

Therefore, the following is an example of lock acquire code using `WFE`:

P*x*

```
Loop
    LDREX R5, [R1]           ; reads lock
    CMP R5, #0               ; checks if 0
    WFENE                    ; sleeps if the lock is held
    STREXEQ R5, R0, [R1]     ; attempts to store new value
    CMPEQ R5, #0             ; tests if store succeeded
    BNE Loop                 ; retries if not
    DMB                      ; ensures that all subsequent accesses are observed after the
                             ; gaining of the lock is observed
    ; loads and stores in the critical region can now be performed
```

And the following is an example of lock release code using `SEV`:

P*x*

```
    ; loads and stores in the critical region
    MOV R0, #0
    DMB                      ; ensures all previous accesses are observed before the lock is cleared
    STR R0, [R1]             ; clears the lock
    DSB                      ; ensures completion of the store that cleared the lock before
                             ; sending the event
    SEV
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### K10.6.3 Using a mailbox to send an interrupt

In some message passing systems, it is common for one observer to update memory and then notify a second observer of the update by sending an interrupt, using a mailbox.

Although a memory access might be made to initiate the sending of the mailbox interrupt, a DSB instruction is required to ensure the completion of previous memory accesses.

Therefore, the following sequence is required to ensure that P2 observes the updated value:

P1

```
STR R5, [R1]            ; message stored to shared memory location
DSB [ST]
STR R1, [R4]            ; R4 contains the address of a mailbox
```

P2

```
; interrupt service routine
LDR R5, [R1]
```

**Note**

The DSB executed by P1 ensures global observation of the store to [R1].The interrupt timing ensures that the code executed by P2 is executed after the global observation of the update to [R1], and therefore must see this update. In some implementations, this might be implemented by requiring that interrupts flush non-coherent buffers that hold speculatively loaded data.

### K10.6.4 Cache and TLB maintenance instructions and barriers

The following sections describe the use of barriers with cache and TLB maintenance instructions:

- *Data cache maintenance instructions*.
- *Instruction cache maintenance instructions* on page K10-5643.
- *TLB maintenance instructions and barriers* on page K10-5645.

#### Data cache maintenance instructions

The following sections describe the use of barriers with data cache maintenance instructions:

- *Message passing to non-caching observers*.
- *Multiprocessing message passing to non-caching observers* on page K10-5642.
- *Invalidating DMA buffers, non-functional example* on page K10-5642.
- *Invalidating DMA buffers, functional example with single PE* on page K10-5642.
- *Invalidating DMA buffers, functional example with multiple coherent PEs* on page K10-5643.

##### *Message passing to non-caching observers*

The ARMv8 architecture requires the use of DMB instructions to ensure the ordering of data cache maintenance instructions and their effects. This means the following message passing approaches can be used when communicating between caching observers and non-caching observers:

P1

```
STR R5, [R1]            ; updates data (assumed to be in P1's cache)
DCCMVAC R1              ; cleans cache to point of coherency
DMB                     ; ensures effects of the clean will be observed before the flag is set
STR R0, [R4]            ; sends flag to external agent (Non-cacheable location)
```

E1

```
WAIT ([R4] == 1)        ; waits for the flag
DMB                     ; ensures that flag has been seen before reading data
LDR R5, [R1]            ; reads the data
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In this example, it is required that E1:R5==0x55.

### *Multiprocessing message passing to non-caching observers*

The broadcast nature of the cache maintenance instructions in ARMv8, and in ARMv7 implementations that include the Multiprocessing Extensions, combined with properties of barriers, means that the message passing principle for non-caching observers is:

P1

```
STR R5, [R1]            ; updates data (assumed to be in P1's cache)
DMB [ST]                ; ensures new data is observed before the flag to P2 is set
STR R0, [R2]            ; sends flag to P2
```

P2

```
WAIT ([R2] == 1)        ; waits for flag from P1
DMB                     ; ensures cache clean is observed after P1 flag is observed
DCCMVAC R1              ; cleans cache to point of coherency - this cleans the cache of P1
DMB                     ; ensures effects of the clean are observed before the flag to E1 is set
STR R0, [R4]            ; sends flag to E1
```

E1

```
WAIT ([R4] == 1)        ; waits for flag from P2
DMB                     ; ensures that flag has been observed before reading the data
LDR R5, [R1]            ; reads the data
```

In this example, it is required that E1:R5==0x55. The clean operation executed by P2 affects the data location in the P1 cache. The cast-out from the P1 cache is guaranteed to be observed before P2 updates [R4].

### *Invalidating DMA buffers, non-functional example*

The basic scheme for communicating with an external observer that is a process that passes data in to a Cacheable memory region must take account of the architectural requirement that regions marked as Cacheable can be allocated into a cache at any time, for example as a result of speculation. The following example shows this possibility:

P1

```
DCIMVAC R1              ; ensures caches are not dirty. A clean operation could be
                        ; used but the DMA overwrites this region so an invalidate operation
                        ; is sufficient and usually more efficient
DMB                     ; ensures cache invalidation is observed before the next store is observed
STR R0, [R3]            ; sends flag to external agent
WAIT ([R4]==1)          ; waits for a different flag from an external agent
DMB                     ; observes flag from external agent before reading new data. However [R1]
                        ; could have been brought into cache earlier
LDR R5, [R1]
```

E1

```
WAIT ([R3] == 1)        ; waits for flag
STR R5, [R1]            ; stores new data
DMB
STR R0, [R4]            ; sends a flag
```

If a speculative access occurs, there is no guarantee that the cache line containing [R1] is not brought back into the cache after the cache invalidation, but before [R1] is written by E1. Therefore, the result P1:R5=0 is permissible.

### *Invalidating DMA buffers, functional example with single PE*

P1

```
DCIMVAC R1              ; ensures cache is not dirty. A clean operation could be
                        ; used but the DMA overwrites this region so an invalidate operation
                        ; is sufficient and usually more efficient
DMB                     ; ensures cache invalidation is observed before the next store is observed
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
    STR R0, [R3]            ; sends flag to external agent
    WAIT ([R4]==1)          ; waits for a different flag from an external agent
    DMB                     ; ensures that cache invalidate is observed after the flag
                            ; from external agent is observed
    DCIMVAC R1              ; ensures cache discards stale copies before use
    LDR R5, [R1]
```

E1

```
    WAIT ([R3] == 1)        ; waits for flag
    STR R5, [R1]            ; stores new data
    DMB [ST]
    STR R0, [R4]            ; sends a flag
```

In this example, the result P1:R5 == `0x55` is required. Including a cache invalidation after the store by E1 to [R1] is observed ensures that the line is fetched from external memory after it has been updated.

#### *Invalidating DMA buffers, functional example with multiple coherent PEs*

The broadcasting of cache maintenance instructions, and the use of `DMB` instructions to ensure their observability, means that the previous example extends naturally to a multiprocessor system. Typically this requires a transfer of ownership of the region that the external observer is updating.

P0

```
    (Use data from [R1], potentially using [R1] as scratch space)
    DMB
    STR R0, [R2]            ; signals release of [R1]
    WAIT ([R2] == 0)        ; waits for new value from DMA
    DMB
    LDR R5, [R1]
```

P1

```
    WAIT ([R2] == 1)        ; waits for release of [R1] by P0
    DCIMVAC R1              ; ensures caches are not dirty, invalidate is sufficient
    DMB
    STR R0, [R3]            ; requests new data for [R1]
    WAIT ([R4] == 1)        ; waits for new data
    DMB
    DCIMVAC R1              ; ensures caches discard stale copies before use
    DMB
    MOV R0, #0
    STR R0, [R2]            ; signals availability of new [R1]
```

E1

```
    WAIT ([R3] == 1)        ; waits for new data request
    STR R5, [R1]            ; sends new [R1]
    DMB [ST]
    STR R0, [R4]            ; indicates new data available to P1
```

In this example, the result P0:R5 == `0x55` is required. The `DMB` issued by P1 after the first data cache invalidation ensures that effect of the cache invalidation on P0 is seen by E1 before the store by E1 to [R1]. The `DMB` issued by P1 after the second data cache invalidation ensures that its effects are seen before the store of 0 to the semaphore location in [R2].

### Instruction cache maintenance instructions

The following sections describe the use of barriers with instruction cache maintenance instructions:

- *Ensuring the visibility of updates to instructions for a uniprocessor* on page K10-5644.
- *Ensuring the visibility of updates to instructions for a multiprocessor* on page K10-5644.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Ensuring the visibility of updates to instructions for a uniprocessor

On a single PE, the agent that causes instruction fetches, or instruction cache linefills, is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the instruction cache can rely only on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

Also, instruction cache maintenance instructions are only guaranteed to complete after the execution of a `DSB`, and an `ISB` is required to discard any instructions that might have been prefetched before the instruction cache invalidation completed. Therefore, on a uniprocessor, to ensure the visibility of an update to code and to branch to it, the following sequence is required:

P1

```
STR R11, [R1]           ; R11 contains a new instruction to store in program memory
DCCMVAU R1              ; clean to PoU makes new instructions visible to instruction cache
DSB
ICIMVAU R1              ; ensures instruction cache and branch predictor discard stale data
BPIMVA R1
DSB                     ; ensures completion of the invalidation
ISB                     ; ensures instruction fetch path observes new instruction cache state
BX R1
```

### Ensuring the visibility of updates to instructions for a multiprocessor

ARMv8, and an ARMv7 implementation that includes the Multiprocessing Extensions, requires a PE that executes an instruction cache maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the cache maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

An `ISB` is not broadcast, and so does not affect other PEs. This means that any other PE must perform its own `ISB` synchronization after it knows that the update is visible, if it is necessary to ensure its synchronization with the update. The following example shows how this might be done:

P1

```
STR R11, [R1]           ; R11 contains a new instruction to store in program memory
DCCMVAU R1              ; clean to PoU makes new instructions visible to instruction cache
DSB                     ; ensures completion of the clean on all processors
ICIMVAU R1              ; ensures instruction cache/branch predictor discards stale data
BPIMVA R1
DSB                     ; ensures completion of the instruction cache and branch predictor
                        ; invalidation on all PEs
STR R0, [R2]            ; sets flag to signal completion
ISB                     ; synchronizes context on this PE
BX R1                   ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)        ; waits for flag signaling completion
ISB                     ; synchronizes context on this processor
BX R1                   ; branches to new code
```

### Nonfunctional approach

The following sequence does not have the same effect, because a `DSB` is not required to complete the instruction cache maintenance instructions that other PEs issue:

P1

```
STR R11, [R1]           ; R11 contains a new instruction to store in program memory
DCCMVAU R1              ; clean to PoU makes new instructions visible to instruction cache
DSB                     ; ensure completion of the clean on all PEs
ICIMVAU R1              ; ensure instruction cache/branch predictor discards stale data
BPIMVA R1
DMB                     ; ensure ordering of the store after the invalidation
                        ; DOES NOT guarantee completion of instruction cache/branch
                        ; predictor on other PEs
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
STR R0, [R2]            ; sets flag to signal completion
DSB                     ; ensures completion of the invalidation on all PEs
ISB                     ; synchronizes context on this PE
BX R1                   ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)        ; waits for flag signaling completion
DSB                     ; this DSB does not guarantee completion of P1's ICIMVAU/BPIMVA
ISB
BX R1
```

In this example, P2…P*x* might not see the updated region of code at R1.

## TLB maintenance instructions and barriers

The following sections describe the use of barriers with TLB maintenance instructions:

- *Ensuring the visibility of updates to translation tables for a uniprocessor*.
- *Ensuring the visibility of updates to translation tables for a multiprocessor*.
- *Paging memory in and out* on page K10-5646.

### *Ensuring the visibility of updates to translation tables for a uniprocessor*

On a single PE, the agent that causes translation table walks is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the TLB can only rely on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

In the ARMv8 architecture, and in an ARMv7 implementation that includes the Multiprocessing Extensions, translation table walks must look in the data or unified caches at L1, so such systems do not require data cache cleaning.

After the translation tables update, any old copies of entries that might be held in the TLBs must be invalidated. This operation is only guaranteed to affect all instructions, including instruction fetches and data accesses, after the execution of a `DSB` and an `ISB`. Therefore, the code for updating a translation table entry is:

P1

```
STR R11, [R1]           ; updates the translation table entry
DSB                     ; ensures visibility of the update to translation table walks
TLBIMVA R10
BPIALL
DSB                     ; ensures completion of the BP and TLB invalidation
ISB                     ; synchronizes context on this PE
;
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using the old mapping have been completed
```

Importantly, by the end of this sequence, all accesses that used the old translation table mappings have been observed by all observers.

An example of this is where a translation table entry is marked as invalid. Such a system must provide a mechanism to ensure that any access to a region of memory being marked as invalid has completed before any action is taken as a result of the region being marked as invalid.

### *Ensuring the visibility of updates to translation tables for a multiprocessor*

The same code sequence can be used in a multiprocessing system. In the ARMv8 architecture, and in an ARMv7 implementation that includes the Multiprocessing Extensions, a PE that executes a TLB maintenance instruction must execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the TLB maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

The completion of a `DSB` that completes a TLB maintenance instruction ensures that all accesses that used the old mapping have completed.

P1


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

```
STR R11, [R1]           ; updates the translation table entry
DSB                     ; ensures visibility of the update to translation table walks
TLBIMVAIS R10
BPIALLIS
DSB                     ; ensures completion of the BP and TLB invalidation
ISB                     ; Note ISB is not broadcast and must be executed locally on other PEs
;
; new translation table entry can be relied upon at this point and all accesses generated by any
; observers affected by the broadcast TLBIMVAIS operation using the old mapping have completed
```

The completion of the TLB maintenance instruction is guaranteed only by the execution of a DSB by the observer that performed the TLB maintenance instruction. The execution of a DSB by a different observer does not have this effect, even if the DSB is known to be executed after the TLB maintenance instruction is observed by that different observer.

### *Paging memory in and out*

In a multiprocessor system there is a requirement to ensure the visibility of translation table updates when paging regions of memory into RAM from a backing store. This might, or might not, also involve paging existing locations in memory from RAM to a backing store. In such situations, the operating system selects one or more pages of memory that might be in use but are suitable to discard, with or without copying to a backing store, depending on whether or not the region of memory is writable. Disabling the translation table mappings for a page, and ensuring the visibility of that update to the translation tables, prevents agents accessing the page.

For this reason, it is important that the DSB that is performed after the TLB invalidation ensures that no other updates to memory using those mappings are possible.

An example sequence for the paging out of an updated region of memory, and the subsequent paging in of memory, is as follows:

P1

```
STR R11, [R1]           ; updates the translation table for the region being paged out
DSB                     ; ensures visibility of the update to translation table walks
TLBIMVAIS R10           ; invalidates the old entry
DSB                     ; ensures completion of the invalidation on all processors
ISB                     ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB                     ; ensures completion of the memory transfer (this could be part of
                        ; LoadMemoryFromBackingStore
ICIALLUIS               ; also invalidates the branch predictor
STR R9, [R1]            ; creates a new translation table entry with a new mapping
DSB                     ; ensures completion of instruction cache and branch predictor invalidation
                        ; and ensures visibility of the new translation table mapping
ISB                     ; ensures synchronization of this instruction stream
```

This example assumes the memory copies are performed by an observer that is coherent with the caches of PE P1. This observer might be P1 itself, using a specific paging mapping. For clarity, the example omits the functional descriptions of SaveMemoryPageToBackingStore and LoadMemoryFromBackingStore. LoadMemoryFromBackingStore is required to ensure that the memory updates that it makes are visible to instruction fetches.

In this example, the use of ICIALLUIS to invalidate the entire instruction cache is a simplification, that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using ICIMVAU operations. This invalidation must be done when the mapping used for the ICIMVAU operations is valid but not executable.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

# Appendix K11
# ARM Pseudocode Definition

This appendix provides a definition of the pseudocode that is used in this manual, and defines some *helper* procedures and functions that are used by pseudocode. It contains the following sections:


- *About the ARM pseudocode* on page K11-5648.
- *Pseudocode for instruction descriptions* on page K11-5649.
- *Data types* on page K11-5651.
- *Operators* on page K11-5656.
- *Statements and control structures* on page K11-5662.
- *Built-in functions* on page K11-5668.
- *Miscellaneous helper procedures and functions* on page K11-5671.
- *ARM pseudocode definition index* on page K11-5674.


**Note**

This appendix is not a formal language definition for the pseudocode. It is a guide to help understand the use of ARM pseudocode. This appendix is not complete. Changes are planned for future releases.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K11.1 About the ARM pseudocode

The ARM pseudocode provides precise descriptions of some areas of the ARM architecture. This includes description of the decoding and operation of all valid instructions. *Pseudocode for instruction descriptions* on page K11-6021 gives general information about this instruction pseudocode, including its limitations.

The following sections describe the ARM pseudocode in detail:

- *Data types* on page K11-6023.
- *Operators* on page K11-6028.
- *Statements and control structures* on page K11-6034.

*Built-in functions* on page K11-6040 and *Miscellaneous helper procedures and functions* on page K11-6043 describe some built-in functions and pseudocode helper functions that are used by the pseudocode functions that are described elsewhere in this manual. *ARM pseudocode definition index* on page K11-6046 contains the indexes to the pseudocode.

### K11.1.1 General limitations of ARM pseudocode

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `UNDEFINED`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

- Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
- No subsequent behavior indicated by the pseudocode occurs.

For more information, see *Special Statements* on page K11-6038.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## K11.2 Pseudocode for instruction descriptions

Each instruction description includes pseudocode that provides a precise description of what the instruction does, subject to the limitations described in *General limitations of ARM pseudocode* on page K11-6020 and *Limitations of the instruction pseudocode* on page K11-6022.

In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction. *Instruction encoding diagrams and instruction pseudocode* gives more information about the pseudocode provided for each instruction.

### K11.2.1 Instruction encoding diagrams and instruction pseudocode

Instruction descriptions in this manual contain:

- An Encoding section, containing one or more encoding diagrams, each followed by some encoding-specific pseudocode that translates the fields of the encoding into inputs for the common pseudocode of the instruction, and picks out any encoding-specific special cases.
- An Operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the `EncodingSpecificOperations()` function, either at its start or only after a condition code check performed by `if ConditionPassed() then`.

An encoding diagram specifies each bit of the instruction as one of the following:

- An obligatory 0 or 1, represented in the diagram as 0 or 1. If this bit does not have this value, the encoding corresponds to a different instruction.
- A *should be* 0 or 1, represented in the diagram as (0) or (1). If this bit does not have this value, the instruction is UNPREDICTABLE. For more information, see *SBZ or SBO fields in instructions* on page K1-5798.
- A named single bit or a bit in a named multi-bit field. The `cond` field in bits[31:28] of many A32/T32 instructions has some special rules associated with it.

An encoding diagram matches an instruction if all obligatory bits are identical in the encoding diagram and the instruction, and one of the following is true:

- The encoding diagram is not for an A32/T32 instruction.
- The encoding diagram is for an A32/T32 instruction that does not have a `cond` field in bits[31:28].
- The encoding diagram is for an A32/T32 instruction that has a `cond` field in bits[31:28], and bits[31:28] of the instruction are not `0b1111`.

In the context of the instruction pseudocode, the execution model for an instruction is:

1. Find all encoding diagrams that match the instruction. It is possible that no encoding diagram matches. In that case, abandon this execution model and consult the relevant instruction set chapter instead to find out how the instruction is to be treated. The bit pattern of such an instruction is usually reserved and UNDEFINED, though there are some other possibilities. For example, unallocated hint instructions are documented as being reserved and executed as NOPs.
2. If the operation pseudocode for the matching encoding diagrams starts with a condition code check, perform that check. If the condition code check fails, abandon this execution model and treat the instruction as a NOP. If there are multiple matching encoding diagrams, either all or none of their corresponding pieces of common pseudocode start with a condition code check.
3. Perform the encoding-specific pseudocode for each of the matching encoding diagrams independently and in parallel. Each such piece of encoding-specific pseudocode starts with a bitstring variable for each named bit or multi-bit field in its corresponding encoding diagram, named the same as the bit or multi-bit field and initialized with the values of the corresponding bit or bits from the bit pattern of the instruction.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

In a few cases, the encoding diagram contains more than one bit or field with same name. In these cases, the values of the different instances of those bits or fields must be identical. The encoding-specific pseudocode contains a special case using the `Consistent()` function to specify what happens if they are not identical. `Consistent()` returns `TRUE` if all instruction bits or fields with the same name as its argument have the same value, and `FALSE` otherwise.

If there are multiple matching encoding diagrams, all but one of the corresponding pieces of pseudocode must contain a special case that indicates that it does not apply. Discard the results of all such pieces of pseudocode and their corresponding encoding diagrams.

There is now one remaining piece of pseudocode and its corresponding encoding diagram left to consider. This pseudocode might also contain a special case, most commonly one indicating that it is UNPREDICTABLE. If so, abandon this execution model and treat the instruction according to the special case.

4. Check the *should be* bits of the encoding diagram against the corresponding bits of the bit pattern of the instruction. If any of them do not match, abandon this execution model and treat the instruction as UNPREDICTABLE.

5. Perform the rest of the operation pseudocode for the instruction description that contains the encoding diagram. That pseudocode starts with all variables set to the values they were left with by the encoding-specific pseudocode.

The `ConditionPassed()` call in the common pseudocode, if present, performs step 2, and the `EncodingSpecificOperations()` call performs steps 3 and 4.

### K11.2.2 Limitations of the instruction pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses see *Ordering requirements* on page E2-2267.
- Pseudocode does not describe the exact rules when an UNDEFINED instruction fails its condition code check. In such cases, the `UNDEFINED` pseudocode statement lies inside the `if ConditionPassed() then …` structure, either directly or in the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a NOP. *Conditional execution of undefined instructions* on page G1-3686 describes the exact rules.
- Pseudocode does not describe the exact ordering requirements when a single floating-point instruction generates more than one floating-point exception and one or more of those floating-point exceptions is trapped. *Combinations of floating-point exceptions* on page E1-2221 describes the exact rules.

  **Note**

  There is no limitation in the case where all the floating-point exceptions are untrapped, because the pseudocode specifies the same behavior as the cross-referenced section.

- An exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function such as `Abort()`, or implicitly, for example if an interrupt is taken during execution of an `LDM` instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs. To determine that, see the descriptions of the exceptions in *Handling exceptions that are taken to an Exception level using AArch32* on page G1-3647.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K11.3 Data types

This section describes:


- *General data type rules*.
- *Bitstrings*.
- *Integers* on page K11-6024.
- *Reals* on page K11-6024.
- *Booleans* on page K11-6025.
- *Enumerations* on page K11-6025.
- *Tuples* on page K11-6025.
- *Structs* on page K11-6026.
- *Arrays* on page K11-6027.


### K11.3.1 General data type rules

ARM architecture pseudocode is a strongly typed language. Every literal and variable is of one of the following types:

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- Tuple.
- Struct.
- Array.

The type of a literal is determined by its syntax. A variable can be assigned to without an explicit declaration. The variable implicitly has the type of the assigned value. For example, the following assignments implicitly declare the variables `x`, `y` and `z` to have types integer, bitstring of length 1, and Boolean, respectively.

```
x = 1;
y = '1';
z = TRUE;
```

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. The following example declares explicitly that a variable named `count` is an integer.

```
integer count;
```

This is most often done in function definitions for the arguments and the result of the function.

The remaining subsections describe each data type in more detail.

### K11.3.2 Bitstrings

This section describes the bitstring data type.

#### Syntax

| | |
|---|---|
| `bits(N)` | The type name of a bitstring of length `N`. |
| `bit` | A synonym of `bits(1)`. |

#### Description

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 1.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

Bitstring constants literals are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants literals of type `bit` are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

The bits in a bitstring are numbered from left to right *N*-1 to 0. This numbering is used when accessing the bitstring using bitslices. In conversions to and from integers, bit *N*-1 is the MSByte and bit 0 is the LSByte. This order matches the order in which bitstrings derived from encoding diagrams are printed.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length `N` is bit (*N*–1) and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings that are derived from encoding diagrams, this order matches the way that they are printed.

Bitstrings are the only concrete data type in pseudocode, corresponding directly to the contents values that are manipulated in registers, memory locations, and instructions. All other data types are abstract.

### K11.3.3 Integers

This section describes the data type for integer numbers.

#### Syntax

`integer` The type name for the integer data type.

#### Description

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as integers.

Integer literals are normally written in decimal form, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal form, such as `0x55` or `0x80000000`. Hexadecimal integer literals are treated as positive unless they have a preceding minus sign. For example, `0x80000000` is the integer $+2^{31}$. If $-2^{31}$ needs to be written in hexadecimal, it must be written as `-0x80000000`.

### K11.3.4 Reals

This section describes the data type for real numbers.

#### Syntax

`real` The type name for the real data type.

#### Description

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable and the pseudocode provides functions to interpret those bitstrings as reals.

Real constants literals are written in decimal form with a decimal point. This means `0` is an integer constant literal, but `0.0` is a real constant literal.

### K11.3.5 Booleans

This section describes the Boolean data type.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Syntax

`boolean` The type name for the Boolean data type.

`TRUE` The two values a Boolean variable can take.

### Description

A Boolean is a logical `TRUE` or `FALSE` value.

**Note**

This is not the same type as `bit`, which is a bitstring of length 1. A Boolean can only take on one of two values: `TRUE` or `FALSE`.

## K11.3.6 Enumerations

This section describes the enumeration data type.

### Syntax and examples

`enumeration` Keyword to defined a new enumeration type.

```
enumeration Example {Example_One, Example_Two, Example_Three};
```

A definition of a new enumeration called `Example`, which can take on the values `Example_One`, `Example_Two`, `Example_Three`.

### Description

An enumeration is a defined set of named values.

An enumeration must contain at least one named value. A named value must not be shared between enumerations.

Enumerations must be defined explicitly, although a variable of an enumeration type can be declared implicitly by assigning one of the named values to it. By convention, each named value starts with the name of the enumeration followed by an underscore. The name of the enumeration is its *type name*, or *type*, and the named values are its possible *values*.

## K11.3.7 Structures

This section describes the structure data type.

### Syntax and examples

`type` The keyword used to declare the structure data type.

```
type ShiftSpec is (bits(2) shift, integer amount)
```

An example definition for a new structure called `ShiftSpec` that contains an bitstring member called `shift` and a integer member named `amount`. Structure definitions must not be terminated with a semicolon.

```
ShiftSpec abc;
```

A declaration of a variable named `abc` of type `ShiftSpec`.

```
abc.shift
```

Syntax to refer to the individual members within the structure variable.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Description

A structure is a compound data type composed of one or more data items. The data items can be of different data types. This can include compound data types. The data items of a structure are called its members and are named.

In the syntax section, the example defines a structure called `ShiftSpec` with two members. The first is a bitstring of length 2 named `shift` and the second is an integer named `amount`. After declaring a variable of that type named `abc`, the members of this structure are referred to as `abc.shift` and `abc.amount`.

Every definition of a structure creates a different type, even if the number and type of their members are identical. For example:

```
type ShiftSpec1 is (bits(2) shift, integer amount)
type ShiftSpec2 is (bits(2) shift, integer amount)
```

`ShiftSpec1` and `ShiftSpec2` are two different types despite having identical definitions. This means that the value in a variable of type `ShiftSpec1` cannot be assigned to variable of type `ShiftSpec2`.

## K11.3.8 Tuples

This section describes the tuple data type.

### Examples

```
(bits(32) shifter_result, bit shifter_carry_out)
```

An example of the tuple syntax.

```
(shift_t, shift_n) = ('00', 0);
```

An example of assigning values to a tuple.

### Description

A tuple is an ordered set of data items, separated by commas and enclosed in parentheses. The items can be of different types and a tuple must contain at least one data item.

Tuples are often used as the return type for functions that return multiple results. For example, in the syntax section, the example tuple is the return type of the function `Shift_C()`, which performs a standard A32/T32 shift or rotation. Its return type is a tuple containing two data items, with the first of type `bits(32)` and the second of type `bit`.

Each tuple is a separate compound data type. The compound data type is represented as a comma-separated list of ordered data types between parentheses. This means that the example tuple at the start of this section is of type `(bits(32), bit)`. The general principle that types can be implied by an assignment extends to implying the type of the elements in the tuple. For example, in the syntax section, the example assignment implicitly declares:

- `shift_t` to be of type `bits(2)`.
- `shift_n` to be of type `integer`.
- `(shift_t, shift_n)` to be a tuple of type `(bits(2), integer)`.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### K11.3.9 Arrays

This section describes the array data type.

#### Syntax

`array` The type name for the array data type.

```
array data_type array_name[A..B];
```

Declaration of an array of type `data_type`, which may be compound data type. It is named `array_name` and is indexed with an integer range from `A` to `B`.

#### Description

An array is an ordered set of fixed size containing items of a single data type. This can include compound data types. Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then `..`, then the upper inclusive end of the range.

For example:

The following example declares an array of 31 bitstrings of length 64, indexed from 0 to 30.

```
array bits(64) _R[0..30];
```

Arrays are always explicitly declared, and there is no notation for a constant literal array. Arrays always contain at least one element data item, because:

- Enumerations always contain at least one symbolic constant named value.
- Integer ranges always contain at least one integer.

An array declared with an enumeration type as the index must be accessed using enumeration values of that enumeration type. An array declared with an integer range type as the index must be accessed using integer values from that inclusive range. Accessing such an array with an integer value outside of the range is a coding error.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as `R[i]`, `MemU[address, size]` or `Elem[vector, i, size]`. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing. See *Function and procedure calls* on page K11-5662.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# K11.4 Operators

This section describes:

- *Relational operators*.
- *Boolean operators*
- *Arithmetic operators* on page K11-6029

## K11.4.1 Relational operators

The following operations yield results of type `boolean`.

### Equality and non-equality

If two variables `x` and `y` are of the same type, their values can be tested for equality by using the expression `x == y` and for non-equality by using the expression `x != y`. In both cases, the result is of type `boolean`.

Both `x` and `y` must be of type `bits(N)`, `real` , `enumeration`, `boolean`, or `integer`. Named values from an `enumeration` can only be compared if they are both from the same `enumeration`. An exception is that a bitstring can be tested for equality with an integer to allow a `d==15` test.

A special form of comparison is defined with a bitstring literal that can contain bit values `'0'` , `'1'`, and `'x'`. Any bit with value `'x'` is ignored in determining the result of the comparison. For example, if `opcode` is a 4-bit bitstring, the expression `opcode == '1x0x'` matches the values `'1000'`, `'1100'`, `'1001'`, and `'1101'`. This is known as a bitmask.

**Note**

This special form is permitted in the implied equality comparisons in the `when` parts of `case … of …` structures.

### Comparisons

If `x` and `y` are integers or reals, then `x < y`, `x <= y`, `x > y`, and `x >= y` are less than, less than or equal, greater than, and greater than or equal comparisons between them, producing Boolean results.

### Set membership with `IN`

`<expression> IN {<set>}` produces `TRUE` if `<expression>` is a member of `<set>`. Otherwise, it is `FALSE`. `<set>` must be a list of expressions separated by commas.

## K11.4.2 Boolean operators

If `x` is a Boolean expression, then `!x` is its logical inverse.

If `x` and `y` are Boolean expressions, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

**Note**

This is known as short circuit evaluation.

If `x` and `y` are `booleans`, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

**Note**

If `x` and `y` are `booleans` or Boolean expressions, then the result of `x != y` is the same as the result of exclusive-ORing `x` and `y` together. The operator `EOR` only accepts bitstring arguments.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### K11.4.3 Bitstring operators

The following operations can be applied only to bitstrings.

#### Logical operations on bitstrings

If x is a bitstring, NOT(x) is the bitstring of the same length obtained by logically inverting every bit of x.

If x and y are bitstrings of the same length, x AND y, x OR y, and x EOR y are the bitstrings of that same length obtained by logically ANDing, logically ORing, and exclusive-ORing corresponding bits of x and y together.

#### Bitstring concatenation and slicing

If x and y are bitstrings of lengths N and M respectively, then x:y is the bitstring of length N+M constructed by concatenating x and y in left-to-right order.

The bitstring slicing operator addresses specific bits in a bitstring. This can be used to create a new bitstring from extracted bits or to set the value of specific bits. Its syntax is x<integer_list>, where x is the integer or bitstring being sliced, and <integer_list> is a comma-separated list of integers enclosed in angle brackets. The length of the resulting bitstring is equal to the number of integers in <integer_list>. In x<integer_list>, each of the integers in <integer_list> must be:

- >= 0.
- < Len(x) if x is a bitstring.

The definition of x<integer_list> depends on whether integer_list contains more than one integer:

- If integer_list contains more than one integer, x<i, j, k,…, n> is defined to be the concatenation:

  x<i> : x<j> : x<k> : … : x<n>.

- If integer_list consists of just one integer i, x<i> is defined to be:
  - If x is a bitstring, '0' if bit i of x is a zero and '1' if bit i of x is a one.
  - If x is an integer, and y is the unique integer in the range 0 to 2^(i+1)-1 that is congruent to x modulo 2^(i+1). Then x<i> is '0' if y < 2^i and '1' if y >= 2^i.

    Loosely, this definition treats an integer as equivalent to a sufficiently long two's complement representation of it as a bitstring.

The notation for a range expression is i:j with i >= j is shorthand for the integers in order from i down to j, with both end values included. For example, instr<31:28> represents instr<31, 30, 29, 28>.

x<integer_list> is assignable provided x is an assignable bitstring and no integer appears more than once in <integer_list>. In particular, x<i> is assignable if x is an assignable bitstring and 0 <= i < Len(x).

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the APSR shows its bit<31> as N. In such cases, the syntax APSR.N is used as a more readable synonym for APSR<31> as named bits can be referred to with the same syntax as referring to members of a struct. A comma-separated list of named bits enclosed in angle brackets following the register name allows multiple bits to be addressed simultaneously. For example, APSR.<N, C, Q> is synonymous with APSR <31, 29, 27>.

### K11.4.4 Arithmetic operators

Most pseudocode arithmetic is performed on integer or real values, with operands obtained by conversions from bitstrings and results converted back to bitstrings . As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

#### Unary plus and minus

If x is an integer or real, then +x is x unchanged, -x is x with its sign reversed. Both are of the same type as x.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Addition and subtraction

If x and y are integers or reals, x+y and x-y are their sum and difference. Both are of type `integer` if x and y are both of type `integer`, and `real` otherwise.

There are two cases where the types of x and y can be different. A bitstring and an integer can be added together to allow the operation `PC + 4`. An integer can be subtracted from a bitstring to allow the operation `PC - 2`.

If x and y are bitstrings of the same length N, so that `N = Len(x) = Len(y)`, then x+y and x-y are the least significant *N* bits of the results of converting x and y to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>
x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

If x is a bitstring of length N and y is an integer, x+y and x-y are the bitstrings of length N defined by `x+y = x + y<N-1:0>` and `x-y = x - y<N-1:0>`. Similarly, if x is an integer and y is a bitstring of length M, x+y and x-y are the bitstrings of length M defined by `x+y = x<M-1:0> + y` and `x-y = x<M-1:0> - y`.

### Multiplication

If x and y are integers or reals, then `x * y` is the product of x and y. It is of type `integer` if x and y are both of type `integer`, and `real` otherwise.

### Division and modulo

If x and y are reals, then x/y is the result of dividing x by y, and is always of type `real`.

If x and y are integers, then `x DIV y` and `x MOD y` are defined by:

```
x DIV y = RoundDown(x/y)
x MOD y = x - y * (x DIV y)
```

It is a pseudocode error to use any of x/y, `x MOD y`, or `x DIV y` in any context where y can be zero.

### Scaling

If x and n are of type `integer`, then:

- `x << n = RoundDown(x * 2^n)`.
- `x >> n = RoundDown(x * 2^(-n))`.

### Raising to a power

If x is an integer or a real and n is an integer then x^n is the result of raising x to the power of n, and:

- If x is of type `integer` then x^n is of type `integer`.
- If x is of type `real` then x^n is of type `real`.

## K11.4.5 The assignment operator

The assignment operator is the = character, which assigns the value of the right-hand side to the left-hand side. An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

This following subsection defines valid expression syntax.

### General expression syntax

An expression is one of the following:

- A literal.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

- A variable, optionally preceded by a data type name to declare its type.
- The word `UNKNOWN` preceded by a data type name to declare its type.
- The result of applying a language-defined operator to other expressions.
- The result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register defined in an ARM architecture specification defines a correspondingly named pseudocode bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as RAZ/WI, then the corresponding bit of its variable reads as `'0'` and ignore writes.

An expression like `bits(32) UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not:

- Return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE or `CONSTRAINED UNPREDICTABLE` and do not return UNKNOWN values,
- Be promoted as providing any useful information to software.

**Note**

UNKNOWN values are similar to the definition of UNPREDICTABLE, but do not indicate that the entire architectural state becomes unspecified.

Only the following expressions are assignable. This means that these are the only expressions that can be placed on the left-hand side of an assignment.

- Variables.
- The results of applying some operators to other expressions.

  The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. For example, those circumstances might require that one or more of the expressions the operator operates on is an assignable expression.
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

**Note**

If the right-hand side in an assignment is a function returning a tuple, an item in the assignment destination can be written as - to indicate that the corresponding item of the assigned tuple value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

The expression on the right-hand side itself can be a tuple. For example:

```
(x, y) = (function_1(), function_2());
```

Every expression has a data type.

- For a literal, this data type is determined by the syntax of the literal.
- For a variable, there are the following possible sources for the data type
  - — An optional preceding data type name.
  - — A data type the variable was given earlier in the pseudocode by recursive application of this rule.
  - — A data type the variable is being given by assignment, either by direct assignment to the variable, or by assignment to a list of which the variable is a member.

  It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

- For a language-defined operator, the definition of the operator determines the data type.
- For a function, the definition of the function determines the data type.

### K11.4.6 Precedence rules

The precedence rules for expressions are:

1. Literals, variables and function invocations are evaluated with higher priority than any operators using their results, but see *Boolean operators* on page K11-6028.
2. Operators on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but need not be if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j` and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

### K11.4.7 Conditional expressions

If `x` and `y` are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

### K11.4.8 Operator polymorphism

Operators in pseudocode can be polymorphic, with different functionality when applied to different data types. Each resulting form of an operator has a different prototype definition. For example, the operator + has forms that act on various combinations of integers, reals and bitstrings.

Table K11-2 summarizes the operand types valid for each unary operator and the result type. Table K11-2 summarizes the operand types valid for each binary operator and the result type.

**Table K11-1 Result and operand types permitted for unary operators**

| Operator | Operand Type | Result Type |
|---|---|---|
| - | integer | integer |
| | real | real |
| NOT | bits(N) | bits(N) |
| ! | boolean | boolean |

**Table K11-2 Result and operand types permitted for binary operators**

| Operator | First operand type | Second operand type | Result type |
|---|---|---|---|
| == | bits(N) | integer | boolean |
| | | bits(N) | |
| | integer | integer | |
| | real | real | |
| | enumeration | enumeration | |
| | boolean | boolean | |
| != | bits(N) | bits(N) | boolean |
| | integer | integer | |
| | real | real | |
| <, > | integer | integer | boolean |
| <= , >= | real | real | |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K11-2 Result and operand types permitted for binary operators (continued)**

| Operator | First operand type | Second operand type | Result type |
|---|---|---|---|
| +, - | integer | integer | integer |
| | real | real | real |
| | bits(N) | bits(N) | bits(N) |
| | | integer | |
| <<, >> | integer | integer | integer |
| * | integer | integer | integer |
| | real | real | real |
| | bits(N) | bits(N) | bits(N) |
| / | real | real | real |
| DIV | integer | integer | integer |
| MOD | integer | integer | integer |
| | bits(N) | integer | |
| &&, \|\| | boolean | boolean | boolean |
| AND, OR, EOR | bits(N) | bits(N) | bits(N) |
| ^ | integer | integer | integer |
| | real | integer | real |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K11.5 Statements and control structures

This section describes the statements and program structures available in the pseudocode.

### K11.5.1 Statements and Indentation

A simple statement is either an assignment, a function call, or a procedure call. Each statement must be terminated with a semicolon.

Indentation normally indicates the structure in compound statements. The statements contained in structures such as `if … then … else …` or procedure and function definitions are indented more deeply than the statement structure itself. The end of a compound statement structure and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level. Standard indentation uses four spaces for each level of indent.

### K11.5.2 Function and procedure calls

This section describes how functions and procedures are defined and called in the pseudocode.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Procedure and function definitions

A procedure definition has the form:

```
<procedure name>(<argument prototypes>)
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

where `<argument prototypes>` consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

**Note**

This first definition line is not terminated by a semicolon. This distinguishes it from a procedure call.

A function definition is similar, but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

**Note**

A function or procedure name can include a "`.`". This is a convention used for functions that have similar but different behaviors in AArch32 and AArch64 states.

Array-like functions are similar, but are written with square brackets and have two forms. These two forms exist because reading from and writing to an array element require different functions. They are frequently used in memory operations. An array-like function definition with a return type is equivalent to reading from an array. For example:

```
<return type> <function name>[<argument prototypes>]
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

Its related function definition with no return type is equivalent to writing to an array. For example:

```
<function name>[<argument prototypes>] = <value prototype>
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

The value prototype determines what data type can be written to the array. The two related functions must share the same name, but the value prototype and return type can be different.

### Procedure calls

A procedure call has the form:

```
<procedure_name>(<arguments>);
```

### Return statements

A procedure return has the form:

```
return;
```

A function return has the form:


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
return <expression>;
```

where `<expression>` is of the type declared in the function prototype line.

### K11.5.3 Conditional control structures

This section describes how conditional control structures are used in the pseudocode.

#### if … then … else …

In addition to being a ternary operator, a multi-line `if … then … else …` structure can act as a control structure and has the form:

```
if <boolean_expression> then
    <statement 1>;
    <statement 2>;
    …
    <statement n>;

elsif <boolean_expression> then
    <statement a>;
    <statement b>;
    …
    <statement z>;
else
    <statement A>;
    <statement B>;
    …
    <statement Z>;
```

The block of lines consisting of `elsif` and its indented statements is optional, and multiple `elsif` blocks can be used.

The block of lines consisting of `else` and its indented statements is optional.

Abbreviated one-line forms can be used when the `then` part, and in the `else` part if it is present, contain only simple statements such as:

```
if <boolean_expression> then <statement 1>;
if <boolean_expression> then <statement 1>; else <statement A>;
if <boolean_expression> then <statement 1>; <statement 2>; else <statement A>;
```

**Note**

In these forms, `<statement 1>`, `<statement 2>` and `<statement A>` must be terminated by semicolons. This and the fact that the `else` part is optional distinguish its use as a control structure from its use as a ternary operator.

#### case … of …

A `case … of …` structure has the form:

```
case <expression> of
    when <literal values1>
        <statement 1>;
        <statement 2>;
        …
        <statement n>;

    when <literal values2>
        <statement 1>;
        <statement 2>;
        …
        <statement n>;

    … more "when" groups if required …
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
otherwise
    <statement A>;
    <statement B>;
    …
    <statement Z>;
```

In this structure, <literal values1> and <literal values2> consist of literal values of the same type as <expression>, separated by commas. There can be additional when groups in the structure. Abbreviated one line forms of when and otherwise parts can be used when they contain only simple statements.

If <expression> has a bitstring type, the literal values can also include bitstring literals containing 'x' bits, known as bitmasks. For details see *Equality and non-equality testing* on page K11-6028.

### K11.5.4 Loop control structures

This section describes the three loop control structures used in the pseudocode.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### repeat … until …

A `repeat … until …` structure has the form:

```
repeat
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
until <boolean_expression>;
```

It executes the statement block at least once, and the loop repeats until `<boolean expression>` evaluates to `TRUE`. Variables explicitly declared inside the loop body have scope local to that loop and may not be accessed outside the loop body.

### while … do

A `while … do` structure has the form:

```
while <boolean_expression> do
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

It begins executing the statement block only if the Boolean expression is true. The loop then runs until the expression is false.

### for …

A `for …` structure has the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
```

The `<assignable_expression>` is initialized to `<integer_expr1>` and compared to `<integer_expr2>`. If `<integer_expr1>` is less than `<integer_expr2>`, the loop body is executed and the `<assignable_expression>` incremented by one. This repeats until `<assignable expression>` is more than or equal to `<integer_expr2>`.

There is an alternate form:

```
for <assignable_expression> = <integer_expr1> downto <integer_expr2>
```

where `<integer_expr1>` is decremented after the loop body executes and continues until `<assignable expression>` is less than or equal than `<integer_expr2>`.

## K11.5.5 Special statements

This section describes statements with particular architecturally-defined behaviors.

### UNDEFINED

This subsection describes the statement:

```
UNDEFINED;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that the Undefined Instruction exception is taken.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is UNPREDICTABLE.

### SEE…

This subsection describes the statement:

```
SEE <reference>;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The `<reference>` indicates where that other pseudocode can be found.

It usually refers to another instruction, but can also refer to another encoding or note of the same instruction.

### IMPLEMENTATION_DEFINED

This subsection describes the statement:

```
IMPLEMENTATION_DEFINED {"<text>"};
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is IMPLEMENTATION DEFINED. An optional `<text>` field can give more information.

## K11.5.6 Comments

The pseudocode supports two styles of comments:

- // starts a comment that is terminated by the end of the line.
- /* starts a comment that is terminated by */.

/**/ statements may not be nested, and the first */ ends the comment.

**Note**

Comment lines do not require a terminating semicolon.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K11.6 Built-in functions

This section describes:

- *Bitstring manipulation functions*
- *Arithmetic functions* on page K11-6042

### K11.6.1 Bitstring manipulation functions

The following bitstring manipulation functions are defined:

#### Bitstring length and most significant bit

If x is a bitstring:

- The bitstring length function `Len(x)` returns the length of x as an integer.

#### Bitstring concatenation and replication

If x is a bitstring and n is an integer with `n >= 0`:

- `Replicate(x, n)` is the bitstring of length `n*Len(x)` consisting of n copies of x concatenated together.
- `Zeros(n)` = `Replicate('0', n)`.
- `Ones(n)` = `Replicate('1', n)`.

#### Bitstring count

If x is a bitstring, `BitCount(x)` is an integer result equal to the number of bits of x that are ones.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### Testing a bitstring for being all zero or all ones

If x is a bitstring:

- IsZero(x) produces TRUE if all of the bits of x are zeros and FALSE if any of them are ones
- IsZeroBit(x) produces '1' if all of the bits of x are zeros and '0' if any of them are ones.

IsOnes(x) and IsOnesBit(x) work in the corresponding ways. This means:

```
IsZero(x)    = (BitCount(x) == 0)
IsOnes(x)    = (BitCount(x) == Len(x))
IsZeroBit(x) = if IsZero(x) then '1' else '0'
IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

### Lowest and highest set bits of a bitstring

If x is a bitstring, and N = Len(x):

- LowestSetBit(x) is the minimum bit number of any of the bits of x that are ones. If all of its bits are zeros, LowestSetBit(x) = N.
- HighestSetBit(x) is the maximum bit number of any of the bits of x that are ones. If all of its bits are zeros, HighestSetBit(x) = -1.
- CountLeadingZeroBits(x) is the number of zero bits at the left end of x, in the range 0 to *N*. This means:

  CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x).
- CountLeadingSignBits(x) is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to *N*-1. This means:

  CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>).

### Zero-extension and sign-extension of bitstrings

If x is a bitstring and i is an integer, then ZeroExtend(x, i) is x extended to a length of i bits, by adding sufficient zero bits to its left. That is, if i == Len(x), then ZeroExtend(x, i) = x, and if i > Len(x), then:

```
ZeroExtend(x, i) = Replicate('0', i-Len(x)) : x
```

If x is a bitstring and i is an integer, then SignExtend(x, i) is x extended to a length of i bits, by adding sufficient copies of its leftmost bit to its left. That is, if i == Len(x), then SignExtend(x, i) = x, and if i > Len(x), then:

```
SignExtend(x, i) = Replicate(TopBit(x), i-Len(x)) : x
```

It is a pseudocode error to use either ZeroExtend(x, i) or SignExtend(x, i) in a context where it is possible that i < Len(x).

### Converting bitstrings to integers

If x is a bitstring, SInt() is the integer whose two's complement representation is x.

UInt() is the integer whose unsigned representation is x.

Int(x, unsigned) returns either SInt(x) or UInt(x) depending on the value of its second argument.

## K11.6.2 Arithmetic functions

This section defines built-in arithmetic functions.

### Absolute value

If x is either of type real or integer, Abs(x) returns the absolute value of x. The result is the same type as x.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### Rounding and aligning

If x is a real:

- `RoundDown(x)` produces the largest integer `n` such that `n <= x`.
- `RoundUp(x)` produces the smallest integer `n` such that `n >= x`.
- `RoundTowardsZero(x)` produces:
  - `RoundDown(x)` if `x > 0.0`.
  - `0` if `x == 0.0`.
  - `RoundUp(x)` if `x < 0.0`.

If x and y are both of type `integer`, `Align(x, y) = y * (x DIV y)`, and is of type `integer`.

If x is of type `bitstring` and y is of type `integer`, `Align(x, y) = (Align(UInt(x), y))<Len(x)-1:0>`, and is a `bitstring` of the same length as x.

It is a pseudocode error to use either form of `Align(x, y)` in any context where y can be 0. In practice, `Align(x, y)` is only used with y a constant power of two, and the bitstring form used with `y = 2^n` has the effect of producing its argument with its n low-order bits forced to zero.

### Maximum and minimum

If x and y are integers or reals, then `Max(x, y)` and `Min(x, y)` are their maximum and minimum respectively. x and y must both be of type integer or of type real. The function returns a value of the same type as its operands.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K11.7 Miscellaneous helper procedures and functions

The functions described in this section are not part of the pseudocode specification. They are miscellaneous *helper* procedures and functions used by pseudocode that are not described elsewhere in this manual. Each has a brief description and a pseudocode prototype, except that the prototype is omitted where it is identical to the section title.

### K11.7.1 EndOfInstruction()

This procedure terminates processing of the current instruction.

```
EndOfInstruction();
```

### K11.7.2 GenerateAlignmentException()

This procedure generates the appropriate exception for an alignment error.

In this manual, `GenerateAlignmentException()` generates a Data Abort exception.

### K11.7.3 Hint_Debug()

This procedure supplies a hint to the debug system.

```
Hint_Debug(bits(4) option);
```

### K11.7.4 Hint_PreloadData()

This procedure performs a *preload data* hint.

```
Hint_PreloadData(bits(32) address);
```

### K11.7.5 Hint_PreloadDataForWrite()

This procedure performs a *preload data* hint with a probability that the use will be for a write.

```
Hint_PreloadDataForWrite(bits(32) address);
```

### K11.7.6 Hint_PreloadInstr()

This procedure performs a *preload instructions* hint.

```
Hint_PreloadInstr(bits(32) address);
```

### K11.7.7 Hint_Yield()

This procedure performs a *Yield* hint.

```
Hint_Yield();
```

### K11.7.8 IntegerZeroDivideTrappingEnabled()

This function returns `TRUE` if the trapping of divisions by zero in the integer division instructions `SDIV` and `UDIV` is enabled, and `FALSE` otherwise.

The ARMv8-A `SDIV` and `UDIV` implementation does not support trapping of integer division by zero and therefore this function always returns FALSE.

```
boolean IntegerZeroDivideTrappingEnabled();
```


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K11.7.9 IsExternalAbort()

This function returns TRUE if the abort currently being processed is an external abort and FALSE otherwise. It is used only in exception entry pseudocode.

```
boolean IsExternalAbort(Fault type)
    assert type != Fault_None;

boolean IsExternalAbort(FaultRecord fault);
```

### K11.7.10 IsAsyncAbort()

This function returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE otherwise. It is used only in exception entry pseudocode.

```
boolean IsAsyncAbort(Fault type)
    assert type != Fault_None;

boolean IsAsyncAbort(FaultRecord fault);
```

### K11.7.11 LSInstructionSyndrome()

This function returns the extended syndrome information for a fault reported in the HSR.

```
bits(11) LSInstructionSyndrome();
```

### K11.7.12 ProcessorID()

This function returns an integer that uniquely identifies the executing PE in the system.

```
integer ProcessorID();
```

### K11.7.13 RemapRegsHaveResetValues()

This function returns TRUE if the remap registers PRRR and NMRR have their IMPLEMENTATION DEFINED reset values, and FALSE otherwise.

```
boolean RemapRegsHaveResetValues();
```

### K11.7.14 ResetControlRegisters()

This function resets the System registers and memory-mapped control registers that have architecturally-defined reset values to those values. For more information about the affected registers see:

- *PE state on reset to AArch64 state* on page D1-1426.
- *PE state on reset into AArch32 state* on page G1-3712.

```
AArch64.ResetControlRegisters(boolean ResetIsCold)
AArch32.ResetControlRegisters(boolean ResetIsCold)
```

### K11.7.15 ThisInstr()

This function returns the bitstring encoding of the currently-executing instruction.

```
bits(32) ThisInstr();
```

**Note**

Currently, this function is used only on 32-bit instruction encodings.

### K11.7.16 ThisInstrLength()

This function returns the length, in bits, of the current instruction. This means it returns 32 or 16:


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

```
integer ThisInstrLength();
```


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K11.8 ARM pseudocode definition index

This section contains the following tables:

- Table K11-3 which contains the pseudocode data types.
- Table K11-4 which contains the pseudocode operators.
- Table K11-5 on page K11-5675 which contains the pseudocode keywords and control structures.
- Table K11-6 on page K11-5676 which contains the statements with special behaviors.

**Table K11-3 Index of pseudocode data types**

| Keyword | Meaning |
|---|---|
| `array` | Type name for the array type |
| `bit` | Keyword equivalent to `bits(1)` |
| `bits(N)` | Type name for the bitstring of length `N` data type |
| `boolean` | Type name for the Boolean data type |
| `enumeration` | Keyword to define a new enumeration type |
| `integer` | Type name for the integer data type |
| `real` | Type name for the real data type |
| `type` | Keyword to define a new structure |

**Table K11-4 Index of pseudocode operators**

| Operator | Meaning |
|---|---|
| `-` | Unary minus on integers or reals |
| | Subtraction of integers, reals and bitstrings |
| | Used in the left-hand side of an assignment or a tuple to discard the result |
| `+` | Unary plus on integers or reals |
| | Addition of integers, reals and bitstrings |
| `.` | Extract named member from a list |
| | Extract named bit or field from a register |
| `:` | Bitstring concatenation |
| | Integer range in bitstring extraction operator |
| `!` | Boolean NOT |
| `!=` | Comparison for inequality |
| `(…)` | Around arguments of procedure or function |
| `[…]` | Around array index |
| | Around arguments of array-like function |
| `*` | Multiplication of integers, reals, and bitstrings |
| `/` | Division of reals |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K11-4 Index of pseudocode operators (continued)**

| Operator | Meaning |
|---|---|
| `&&` | Boolean AND |
| `<` | *Less than* comparison of integers and reals |
| `<…>` | Slicing of specified bits of bitstring or integer |
| `<<` | Multiply integer by power of 2 |
| `<=` | *Less than or equal* comparison of integers and reals |
| `=` | Assignment operator |
| `==` | Comparison for equality |
| `>` | *Greater than* comparison of integers and reals |
| `>=` | *Greater than or equal* comparison of integers and reals |
| `>>` | Divide integer by power of 2 |
| `\|\|` | Boolean OR |
| `^` | Exponential operator |
| `AND` | Bitwise AND of bitstrings |
| `DIV` | Quotient from integer division |
| `EOR` | Bitwise EOR of bitstrings |
| `IN` | Tests membership of a certain expression in a set of values |
| `MOD` | Remainder from integer division |
| `NOT` | Bitwise inversion of bitstrings |
| `OR` | Bitwise OR of bitstrings |

**Table K11-5 Index of pseudocode keywords and control structures**

| Operator | Meaning |
|---|---|
| `/*…*/` | Comment delimiters |
| `//` | Introduces comment terminated by end of line |
| `case … of …` | Control structure for the |
| `FALSE` | One of two values a Boolean can take (other than `TRUE`) |
| `for … = …to …` | Loop control structure, counting up from the initial value to the upper limit |
| `for … = … downto …` | Loop control structure, counting down from the initial value to the lower limit |
| `if … then … else …` | Condition expression selecting between two values |
| `if … then … else …` | Conditional control structure |
| `otherwise` | Introduces default case in `case … of …` control structure |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K11-5 Index of pseudocode keywords and control structures (continued)**

| Operator | Meaning |
|---|---|
| `repeat … until …` | Loop control structure that runs at least once until the termination condition is satisfied |
| `return` | Procedure or function return |
| `TRUE` | One of two values a Boolean can take (other than `FALSE`) |
| `when` | Introduces specific case in `case … of …` control structure |
| `while … do …` | Loop control structure that runs until the termination condition is satisfied |

**Table K11-6 Index of special statements**

| Keyword | Meaning |
|---|---|
| `IMPLEMENTATION_DEFINED` | Describes IMPLEMENTATION DEFINED behavior |
| `SEE` | Points to other pseudocode to use instead |
| `UNDEFINED` | Cause Undefined Instruction exception |
| `UNKNOWN` | Unspecified value |
| `UNPREDICTABLE` | Unspecified behavior |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K12
# **Pseudocode Index**

This appendix provides indexes to pseudocode definitions and to the pseudocode functions. It contains the following sections:


- *Pseudocode operators and keywords* on page K12-5678.
- *Pseudocode index* on page K12-5679.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K12.1 Pseudocode operators and keywords

See *ARM pseudocode definition index* on page K11-5674 for a full list of operators, keywords, data types, and control structures.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K12.2 Pseudocode index

Table K12-1 indexes the definitions of the pseudocode functions.

**Table K12-1 Pseudocode functions and procedures**

| Function | See |
|---|---|
| _Dclone[] | *aarch32/functions/registers/_Dclone* on page J1-5320 |
| _Mem[] | *Basic memory access* on page D3-1731<br>*Basic memory access* on page G3-3980<br>*shared/functions/memory/_Mem* on page J1-5400 |
| _PC | *shared/functions/registers/_PC* on page J1-5401 |
| A32ExpandImm_C() | *Operation of modified immediate constants, A32 instructions* on page F2-2457<br>*aarch32/functions/common/A32ExpandImm_C* on page J1-5303 |
| A32ExpandImm() | *Operation of modified immediate constants, A32 instructions* on page F2-2457<br>*aarch32/functions/common/A32ExpandImm* on page J1-5303 |
| Abort() | *Abort exceptions* on page D3-1736<br>*Abort exceptions* on page G3-3985<br>*aarch64/exceptions/aborts/AArch64.Abort* on page J1-5221<br>*aarch32/exceptions/aborts/AArch32.Abort* on page J1-5282 |
| AbortSyndrome() | *aarch64/exceptions/aborts/AArch64.AbortSyndrome* on page J1-5221<br>*aarch32/exceptions/aborts/AArch32.AbortSyndrome* on page J1-5282 |
| Abs() | *Unary plus and minus* on page K11-5657<br>*shared/functions/common/Abs* on page J1-5368 |
| AccessFlagFault() | *Pseudocode description of the MMU faults* on page D4-1832<br>*aarch64/translation/faults/AArch64.AccessFlagFault* on page J1-5262<br>*aarch32/translation/faults/AArch32.AccessFlagFault* on page J1-5332 |
| AccType | *shared/functions/memory/AccType* on page J1-5396 |
| AddressDescriptor | *Memory data type definitions* on page D3-1730<br>*Memory data type definitions* on page G3-3979<br>*shared/functions/memory/AddressDescriptor* on page J1-5397 |
| AddressSizeFault() | *Pseudocode description of the MMU faults* on page D4-1832<br>*aarch64/translation/faults/AArch64.AddressSizeFault* on page J1-5262<br>*aarch32/translation/faults/AArch32.AddressSizeFault* on page J1-5332 |
| AddrTop() | *Address tagging in AArch64 state* on page D4-1742<br>*shared/functions/memory/AddrTop* on page J1-5397 |
| AddWithCarry() | *Pseudocode description of addition and subtraction* on page E1-2314<br>*shared/functions/integer/AddWithCarry* on page J1-5396 |
| AdvSIMDExpandImm() | *Advanced SIMD expand immediate pseudocode* on page F5-2564<br>*shared/functions/vector/AdvSIMDExpandImm* on page J1-5411 |
| AdvSIMDFPAccessTrap() | *aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap* on page J1-5231 |
| AESInvMixColumns() | *shared/functions/crypto/AESInvMixColumns* on page J1-5375 |
| AESInvShiftRows() | *shared/functions/crypto/AESInvShiftRows* on page J1-5375 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| `AESInvSubBytes()` | *shared/functions/crypto/AESInvSubBytes* on page J1-5375 |
| `AESMixColumns()` | *shared/functions/crypto/AESMixColumns* on page J1-5375 |
| `AESShiftRows()` | *shared/functions/crypto/AESShiftRows* on page J1-5375 |
| `AESSubBytes()` | *shared/functions/crypto/AESSubBytes* on page J1-5375 |
| `Align()` | *Rounding and aligning* on page K11-5670<br>*shared/functions/common/Align* on page J1-5368 |
| `AlignmentFault()` | *Pseudocode description of the MMU faults* on page D4-1832<br>*aarch64/translation/faults/AArch64.AlignmentFault* on page J1-5262<br>*aarch32/translation/faults/AArch32.AlignmentFault* on page J1-5333 |
| `Allocation` | *shared/functions/memory/Allocation* on page J1-5397 |
| `AllowExternalDebugAccess()` | *External access disabled* on page H8-4952<br>*shared/debug/authentication/AllowExternalDebugAccess* on page J1-5346 |
| `AllowExternalPMUAccess()` | *External access disabled* on page H8-4952<br>*shared/debug/authentication/AllowExternalPMUAccess* on page J1-5346 |
| `ALUExceptionReturn()` | *aarch32/functions/registers/ALUExceptionReturn* on page J1-5316 |
| `ALUWritePC()` | *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318<br>*aarch32/functions/registers/ALUWritePC* on page J1-5316 |
| `ArchVersion()` | *shared/functions/system/ArchVersion* on page J1-5402 |
| `ASR_C()` | *Pseudocode description of shift and rotate operations* on page E1-2313<br>*shared/functions/common/ASR_C* on page J1-5368 |
| `ASR()` | *Pseudocode description of shift and rotate operations* on page E1-2313<br>*shared/functions/common/ASR* on page J1-5368 |
| `AsynchExternalAbort()` | *aarch64/translation/faults/AArch64.AsynchExternalAbort* on page J1-5263<br>*aarch32/translation/faults/AArch32.AsynchExternalAbort* on page J1-5333 |
| `BadMode()` | *aarch32/functions/system/BadMode* on page J1-5321 |
| `BankedRegisterAccessValid()` | *Pseudocode support for the Banked register transfer instructions* on page F6-3205<br>*aarch32/functions/system/BankedRegisterAccessValid* on page J1-5321 |
| `BFXPreferred()` | *aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred* on page J1-5252 |
| `BigEndian()` | *Mixed-endian support* on page D3-1706<br>*Mixed-endian support* on page G3-3950<br>*shared/functions/memory/BigEndian* on page J1-5397 |
| `BigEndianReverse()` | *Endianness in SIMD operations* on page B2-79<br>*shared/functions/memory/BigEndianReverse* on page J1-5397 |
| `BitCount()` | *Bitstring count* on page K11-5668<br>*shared/functions/common/BitCount* on page J1-5369 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| BitReverse() | *shared/functions/crc/BitReverse* on page J1-5374 |
| BranchTo() | *shared/functions/registers/BranchTo* on page J1-5400 |
| BranchType | *shared/functions/registers/BranchType* on page J1-5400 |
| BranchWritePC() | *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318<br>*aarch32/functions/registers/BranchWritePC* on page J1-5316 |
| BreakpointException() | *Pseudocode description of debug exceptions* on page D2-1648<br>*Pseudocode description of Breakpoint exceptions taken from AArch64 state* on page D2-1666<br>*aarch64/exceptions/debug/AArch64.BreakpointException* on page J1-5225 |
| BreakpointMatch() | *Pseudocode description of Breakpoint exceptions taken from AArch64 state* on page D2-1666<br>*Pseudocode description of Breakpoint exceptions taken from AArch32 state* on page G2-3918<br>*aarch64/debug/breakpoint/AArch64.BreakpointMatch* on page J1-5214<br>*aarch32/debug/breakpoint/AArch32.BreakpointMatch* on page J1-5274 |
| BreakpointValueMatch() | *Pseudocode description of Breakpoint exceptions taken from AArch64 state* on page D2-1666<br>*Pseudocode description of Breakpoint exceptions taken from AArch32 state* on page G2-3918<br>*aarch64/debug/breakpoint/AArch64.BreakpointValueMatch* on page J1-5215<br>*aarch32/debug/breakpoint/AArch32.BreakpointValueMatch* on page J1-5275 |
| BXWritePC() | *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318<br>*aarch32/functions/registers/BXWritePC* on page J1-5316 |
| Cacheability | *shared/functions/memory/Cacheability* on page J1-5398 |
| CallHypervisor() | *Pseudocode details of system calls* on page F2-2462<br>*aarch64/exceptions/syscalls/AArch64.CallHypervisor* on page J1-5229<br>*aarch32/exceptions/syscalls/AArch32.CallHypervisor* on page J1-5291 |
| CallSecureMonitor() | *aarch64/exceptions/syscalls/AArch64.CallSecureMonitor* on page J1-5229 |
| CallSupervisor() | *Pseudocode details of system calls* on page F2-2462<br>*aarch64/exceptions/syscalls/AArch64.CallSupervisor* on page J1-5230<br>*aarch32/exceptions/syscalls/AArch32.CallSupervisor* on page J1-5291 |
| CheckAdvSIMDEnabled() | *aarch32/functions/float/CheckAdvSIMDEnabled* on page J1-5309 |
| CheckAdvSIMDOrFPEnabled() | *aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEnabled* on page J1-5295 |
| CheckAdvSIMDOrVFPEnabled() | *aarch32/functions/float/CheckAdvSIMDOrVFPEnabled* on page J1-5309 |
| CheckAdvSIMDFPSystemRegisterTraps | *aarch64/functions/system/AArch64.CheckAdvSIMDFPSystemRegisterTraps* on page J1-5248 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| `CheckAlignment()` | *Unaligned memory access* on page D3-1732<br>*aarch64/functions/memory/AArch64.CheckAlignment* on page J1-5240<br>*aarch32/functions/memory/AArch32.CheckAlignment* on page J1-5311 |
| `CheckBreakpoint()` | *Pseudocode description of Breakpoint exceptions taken from AArch64 state* on page D2-1666<br>*Pseudocode description of Breakpoint exceptions taken from AArch32 state* on page G2-3918<br>*aarch64/translation/debug/AArch64.CheckBreakpoint* on page J1-5261<br>*aarch32/translation/debug/AArch32.CheckBreakpoint* on page J1-5330 |
| `CheckCoprocInstr()` | *aarch64/exceptions/traps/AArch64.CheckCoprocInstr* on page J1-5233 |
| `CheckCoprocInstrTraps()` | *aarch64/exceptions/traps/AArch64.CheckCoprocInstrTraps* on page J1-5234<br>*aarch32/functions/coproc/AArch32.CheckCoprocInstrTraps* on page J1-5305 |
| `CheckCP15InstrCoarseTraps()` | *aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps* on page J1-5232<br>*aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps* on page J1-5305 |
| `CheckCryptoEnabled32()` | *aarch32/functions/float/CheckCryptoEnabled32* on page J1-5310 |
| `CheckCryptoEnabled64()` | *aarch64/instrs/vector/crypto/enabled/CheckCryptoEnabled64* on page J1-5256 |
| `CheckDebug()` | *aarch64/translation/debug/AArch64.CheckDebug* on page J1-5261<br>*aarch32/translation/debug/AArch32.CheckDebug* on page J1-5331 |
| `CheckDomain()` | *Domain checking* on page G4-4185<br>*aarch32/translation/checks/AArch32.CheckDomain* on page J1-5329 |
| `CheckExceptionCatch()` | *Pseudocode description of Exception Catch debug events* on page H3-4877<br>*shared/debug/haltingevents/CheckExceptionCatch* on page J1-5355 |
| `CheckForDCCInterrupts()` | *Pseudocode description of the operation of the DCC and ITR registers* on page H4-4903<br>*shared/debug/dccanditr/CheckForDCCInterrupts* on page J1-5348 |
| `CheckForPMUOverflow()` | *Pseudocode description of overflow interrupt requests* on page D5-1862<br>*aarch64/debug/pmu/AArch64.CheckForPMUOverflow* on page J1-5217<br>*aarch32/debug/pmu/AArch32.CheckForPMUOverflow* on page J1-5277 |
| `CheckForSMCTrap()` | *aarch64/exceptions/traps/AArch64.CheckForSMCTrap* on page J1-5234<br>*aarch32/exceptions/traps/AArch32.CheckForSMCTrap* on page J1-5296 |
| `CheckForWFxTrap()` | *aarch64/exceptions/traps/AArch64.CheckForWFxTrap* on page J1-5235<br>*aarch32/exceptions/traps/AArch32.CheckForWFxTrap* on page J1-5296 |
| `CheckFPAdvSIMDEnabled()` | *aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled* on page J1-5234 |
| `CheckFPAdvSIMDEnabled64()` | *aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64* on page J1-5237 |
| `CheckFPAdvSIMDTrap()` | *aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap* on page J1-5234<br>*aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap* on page J1-5295 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| CheckHaltingStep() | *Pseudocode description of Halting Step debug events* on page H3-4873<br>*shared/debug/haltingevents/CheckHaltingStep* on page J1-5355 |
| CheckIllegalState() | *aarch64/exceptions/traps/AArch64.CheckIllegalState* on page J1-5235<br>*aarch32/exceptions/traps/AArch32.CheckIllegalState* on page J1-5297 |
| CheckITEnabled() | *aarch32/exceptions/traps/AArch32.CheckITEnabled* on page J1-5296 |
| CheckOSUnlockCatch() | *Pseudocode description of OS Unlock Catch debug event* on page H3-4879<br>*shared/debug/haltingevents/CheckOSUnlockCatch* on page J1-5356 |
| CheckPCAlignment() | *aarch64/exceptions/aborts/AArch64.CheckPCAlignment* on page J1-5222<br>*aarch32/exceptions/aborts/AArch32.CheckPCAlignment* on page J1-5283 |
| CheckPendingOSUnlockCatch() | *Pseudocode description of OS Unlock Catch debug event* on page H3-4879<br>*shared/debug/haltingevents/CheckPendingOSUnlockCatch* on page J1-5356 |
| CheckPendingResetCatch() | *Pseudocode description of Reset Catch debug event* on page H3-4880<br>*shared/debug/haltingevents/CheckPendingResetCatch* on page J1-5356 |
| CheckPermission() | *Access permission checking* on page D3-1735<br>*Access permission checking* on page G3-3984<br>*Stage 2 translation table walk* on page G4-4194<br>*aarch64/translation/checks/AArch64.CheckPermission* on page J1-5260<br>*aarch32/translation/checks/AArch32.CheckPermission* on page J1-5329 |
| CheckResetCatch() | *Pseudocode description of Reset Catch debug event* on page H3-4880<br>*shared/debug/haltingevents/CheckResetCatch* on page J1-5356 |
| CheckS2Permission() | *Support functions* on page D4-1793<br>*Stage 2 translation table walk* on page G4-4194<br>*aarch64/translation/checks/AArch64.CheckS2Permission* on page J1-5260<br>*aarch32/translation/checks/AArch32.CheckS2Permission* on page J1-5330 |
| CheckSETENDEnabled() | *aarch32/exceptions/traps/AArch32.CheckSETENDEnabled* on page J1-5297 |
| CheckSoftwareAccessToDebugRegisters() | *Pseudocode description of Software Access debug event* on page H3-4881<br>*shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters* on page J1-5356 |
| CheckSoftwareStep() | *Pseudocode description of Software Step exceptions* on page D2-1699<br>*shared/debug/softwarestep/CheckSoftwareStep* on page J1-5358 |
| CheckSPAlignment() | *aarch64/functions/memory/CheckSPAlignment* on page J1-5241 |
| CheckSystemAccess() | *aarch64/functions/system/CheckSystemAccess* on page J1-5249 |
| CheckSystemRegisterTraps() | *aarch64/functions/system/AArch64.CheckSystemRegisterTraps* on page J1-5249 |
| CheckUnallocatedSystemAccess | *aarch64/functions/system/AArch64.CheckUnallocatedSystemAccess* on page J1-5249 |
| CheckVectorCatch() | *Pseudocode description of Vector Catch exceptions* on page G2-3943<br>*aarch32/translation/debug/AArch32.CheckVectorCatch* on page J1-5331 |
| CheckVFPEnabled() | *aarch32/functions/float/CheckVFPEnabled* on page J1-5310 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| `CheckWatchpoint()` | *Pseudocode description of Watchpoint exceptions taken from AArch64 state* on page D2-1682<br>*Pseudocode description of Watchpoint exceptions taken from AArch32 state* on page G2-3934<br>*aarch64/translation/debug/AArch64.CheckWatchpoint* on page J1-5262<br>*aarch32/translation/debug/AArch32.CheckWatchpoint* on page J1-5332 |
| `ClearEventRegister()` | *shared/functions/system/ClearEventRegister* on page J1-5402 |
| `ClearExclusiveByAddress()` | *Exclusive monitors operations* on page D3-1733<br>*Exclusive monitors operations* on page G3-3982<br>*shared/functions/exclusive/ClearExclusiveByAddress* on page J1-5376 |
| `ClearExclusiveLocal()` | *Exclusive monitors operations* on page D3-1733<br>*Exclusive monitors operations* on page G3-3982<br>*shared/functions/exclusive/ClearExclusiveLocal* on page J1-5376 |
| `ClearExclusiveMonitors()` | *shared/functions/exclusive/ClearExclusiveMonitors* on page J1-5376 |
| `ClearStickyErrors()` | *Pseudocode description of clearing the error flag* on page H4-4895<br>*shared/debug/ClearStickyErrors/ClearStickyErrors* on page J1-5345 |
| `CombineS1S2AttrHints()` | *shared/translation/attrs/CombineS1S2AttrHints* on page J1-5414 |
| `CombineS1S2Desc()` | *Stage 2 translation table walk* on page G4-4194<br>*shared/translation/attrs/CombineS1S2Desc* on page J1-5414 |
| `CombineS1S2Device()` | *shared/translation/attrs/CombineS1S2Device* on page J1-5415 |
| `CompareOp` | *aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp* on page J1-5256 |
| `ConditionHolds()` | *Pseudocode description of conditional execution* on page F2-2448<br>*shared/functions/system/ConditionHolds* on page J1-5402 |
| `ConditionPassed()` | *Pseudocode description of conditional execution* on page F2-2448<br>*aarch32/functions/system/ConditionPassed* on page J1-5322 |
| `Constraint` | *shared/functions/unpredictable/Constraint* on page J1-5411 |
| `ConstrainUnpredictable()` | *shared/functions/unpredictable/ConstrainUnpredictable* on page J1-5411 |
| `ConstrainUnpredictableBits()` | *shared/functions/unpredictable/ConstrainUnpredictableBits* on page J1-5411 |
| `ConstrainUnpredictableBool()` | *shared/functions/unpredictable/ConstrainUnpredictableBool* on page J1-5411 |
| `ConstrainUnpredictableInteger()` | *shared/functions/unpredictable/ConstrainUnpredictableInteger* on page J1-5411 |
| `Coproc_CanWriteAPSR()` | *aarch32/functions/coproc/Coproc_CanWriteAPSR* on page J1-5306 |
| `Coproc_CheckInstr()` | *aarch32/functions/coproc/Coproc_CheckInstr* on page J1-5306 |
| `Coproc_DoneLoading()` | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/Coproc_DoneLoading* on page J1-5307 |
| `Coproc_DoneStoring()` | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/Coproc_DoneStoring* on page J1-5307 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| Coproc_GetOneWord() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/Coproc_GetOneWord* on page J1-5307 |
| Coproc_GetTwoWords() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/Coproc_GetTwoWords* on page J1-5308 |
| Coproc_GetWordToStore() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/Coproc_GetWordToStore* on page J1-5308 |
| Coproc_InternalOperation() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/Coproc_InternalOperation* on page J1-5308 |
| Coproc_SendLoadedWord() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/Coproc_SendLoadedWord* on page J1-5308 |
| Coproc_SendOneWord() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/Coproc_SendOneWord* on page J1-5308 |
| Coproc_SendTwoWords() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/Coproc_SendTwoWords* on page J1-5308 |
| CountEvents() | *aarch64/debug/pmu/AArch64.CountEvents* on page J1-5217<br>*aarch32/debug/pmu/AArch32.CountEvents* on page J1-5278 |
| CountLeadingSignBits() | *Lowest and highest set bits of a bitstring* on page K11-5669<br>*shared/functions/common/CountLeadingSignBits* on page J1-5369 |
| CountLeadingZeroBits() | *Lowest and highest set bits of a bitstring* on page K11-5669<br>*shared/functions/common/CountLeadingZeroBits* on page J1-5369 |
| CountOp | *aarch64/instrs/countop/CountOp* on page J1-5251 |
| CP14DebugInstrDecode() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/CP14DebugInstrDecode* on page J1-5306 |
| CP14JazelleInstrDecode() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/CP14JazelleInstrDecode* on page J1-5306 |
| CP14TraceInstrDecode() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/CP14TraceInstrDecode* on page J1-5306 |
| CP15InstrDecode() | *Pseudocode description of coprocessor operations* on page F2-2461<br>*aarch32/functions/coproc/CP15InstrDecode* on page J1-5306 |
| CPRegTrap() | *aarch64/exceptions/traps/AArch64.CPRegTrap* on page J1-5231<br>*aarch32/exceptions/traps/AArch32.CPRegTrap* on page J1-5293 |
| CPRegTrapSyndrome() | *aarch64/exceptions/traps/AArch64.CPRegTrapSyndrome* on page J1-5231<br>*aarch32/exceptions/traps/AArch32.CPRegTrapSyndrome* on page J1-5294 |
| CPSRWriteByInstr() | *aarch32/functions/system/CPSRWriteByInstr* on page J1-5322 |
| CreateFaultRecord() | *aarch64/functions/aborts/AArch64.CreateFaultRecord* on page J1-5238<br>*aarch32/functions/aborts/AArch32.CreateFaultRecord* on page J1-5301 |
| CreatePCSample() | *Pseudocode description of Sample-based Profiling* on page H7-4940<br>*shared/debug/samplebasedprofiling/CreatePCSample* on page J1-5357 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| CrossTriggerIn | *shared/debug/cti/CrossTrigger* on page J1-5348 |
| CrossTriggerOut | *shared/debug/cti/CrossTrigger* on page J1-5348 |
| CTI_SetEventLevel() | *shared/debug/cti/CTI_SetEventLevel* on page J1-5348 |
| CTI_SignalEvent() | *shared/debug/cti/CTI_SignalEvent* on page J1-5348 |
| CurrentCond() | *Pseudocode description of conditional execution* on page F2-2448<br>*aarch32/functions/system/CurrentCond* on page J1-5322 |
| CurrentInstrSet() | *Pseudocode description of PSTATE PE state fields* on page E1-2322<br>*shared/functions/system/CurrentInstrSet* on page J1-5402 |
| CurrentPL() | *shared/functions/system/CurrentPL* on page J1-5402 |
| D[] | *Pseudocode description of the SIMD and Floating-point register file* on page E1-2327<br>*aarch32/functions/registers/D* on page J1-5316 |
| DataAbort() | *Abort exceptions* on page D3-1736<br>*aarch64/exceptions/aborts/AArch64.DataAbort* on page J1-5222 |
| DataMemoryBarrier() | *shared/functions/memory/DataMemoryBarrier* on page J1-5398 |
| DataSynchronizationBarrier() | *shared/functions/memory/DataSynchronizationBarrier* on page J1-5398 |
| DBGDTR_EL0[] | *Pseudocode description of the operation of the DCC and ITR registers* on page H4-4903<br>*shared/debug/dccanditr/DBGDTR_EL0* on page J1-5350 |
| DBGDTRRX_EL0[] | *Pseudocode description of the operation of the DCC and ITR registers* on page H4-4903<br>*shared/debug/dccanditr/DBGDTRRX_EL0* on page J1-5348 |
| DBGDTRTX_EL0[] | *Pseudocode description of the operation of the DCC and ITR registers* on page H4-4903<br>*shared/debug/dccanditr/DBGDTRTX_EL0* on page J1-5349 |
| DCPSInstruction() | *DCPS* on page H2-4847<br>*shared/debug/halting/DCPSInstruction* on page J1-5351 |
| Debug_authentication() | *shared/debug/authentication/Debug_authentication* on page J1-5347 |
| DebugException | *aarch32/exceptions/debug/DebugException* on page J1-5288 |
| DebugExceptionReturnSS() | *Pseudocode description of Software Step exceptions* on page D2-1699<br>*shared/debug/softwarestep/DebugExceptionReturnSS* on page J1-5359 |
| DebugFault() | *Pseudocode description of debug exceptions* on page D2-1648<br>*Pseudocode description of debug exceptions* on page G2-3894<br>*aarch64/translation/faults/AArch64.DebugFault* on page J1-5263<br>*aarch32/translation/faults/AArch32.DebugFault* on page J1-5333 |
| DebugHalt() | *Pseudocode description of entering Debug state* on page H2-4830<br>*shared/debug/halting/DebugHalt* on page J1-5352 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| DebugTarget() | *Pseudocode description of routing debug exceptions* on page D2-1642<br>*Pseudocode description of routing debug exceptions* on page G2-3888<br>*shared/debug/DebugTarget/DebugTarget* on page J1-5345 |
| DebugTargetFrom() | *Pseudocode description of routing debug exceptions* on page D2-1642<br>*Pseudocode description of routing debug exceptions* on page G2-3888<br>*shared/debug/DebugTarget/DebugTargetFrom* on page J1-5345 |
| DecodeBitMasks() | *aarch64/instrs/integer/bitmasks/DecodeBitMasks* on page J1-5253 |
| DecodeImmShift() | *Pseudocode description of instruction-specified shifts and rotates* on page F2-2450<br>*aarch32/functions/common/DecodeImmShift* on page J1-5303 |
| DecodeRegExtend() | *aarch64/instrs/extendreg/DecodeRegExtend* on page J1-5251 |
| DecodeRegShift() | *Pseudocode description of instruction-specified shifts and rotates* on page F2-2450<br>*aarch32/functions/common/DecodeRegShift* on page J1-5303 |
| DecodeShift() | *aarch64/instrs/integer/shiftreg/DecodeShift* on page J1-5254 |
| DefaultTEXDecode() | *aarch32/translation/attrs/AArch32.DefaultTEXDecode* on page J1-5325 |
| DeviceType | *Memory data type definitions* on page D3-1730<br>*Memory data type definitions* on page G3-3979<br>*shared/functions/memory/DeviceType* on page J1-5398 |
| Din[] | *Pseudocode description of the SIMD and Floating-point register file* on page E1-2327<br>*aarch32/functions/registers/Din* on page J1-5317 |
| DisableITRAndResumeInstructionPrefetch() | *shared/debug/halting/DisableITRAndResumeInstructionPrefetch* on page J1-5353 |
| DomainFault() | *aarch32/translation/faults/AArch32.DomainFault* on page J1-5333 |
| DomainValid() | *aarch32/functions/aborts/AArch32.DomainValid* on page J1-5301 |
| DoubleLockStatus() | *Debug behavior when the OS Double Lock is locked* on page H6-4932<br>*shared/debug/DoubleLockStatus/DoubleLockStatus* on page J1-5346 |
| DRPSInstruction() | *DRPS* on page H2-4849<br>*shared/debug/halting/DRPSInstruction* on page J1-5352 |
| DTRRX | *Pseudocode description of the operation of the DCC and ITR registers* on page H4-4903<br>*shared/debug/dccanditr/DTR* on page J1-5351 |
| DTRTX | *Pseudocode description of the operation of the DCC and ITR registers* on page H4-4903<br>*shared/debug/dccanditr/DTR* on page J1-5351 |
| EDITR[] | *Pseudocode description of the operation of the DCC and ITR registers* on page H4-4903<br>*shared/debug/dccanditr/EDITR* on page J1-5351 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| EDPCSRlo[] | *Pseudocode description of Sample-based Profiling* on page H7-4940<br>*shared/debug/samplebasedprofiling/EDPCSRlo* on page J1-5358 |
| EL0 | *shared/functions/system/EL0* on page J1-5402 |
| Elem[] | *Pseudocode description of Advanced SIMD vectors* on page E1-2328<br>*shared/functions/common/Elem* on page J1-5369 |
| ELFromM32() | *shared/functions/system/ELFromM32* on page J1-5403 |
| ELFromSPSR() | *shared/functions/system/ELFromSPSR* on page J1-5403 |
| ELR[] | *aarch64/functions/sysregisters/ELR* on page J1-5246 |
| ELStateUsingAArch32() | *shared/functions/system/ELStateUsingAArch32* on page J1-5403 |
| ELStateUsingAArch32K() | *shared/functions/system/ELStateUsingAArch32K* on page J1-5404 |
| ELUsingAArch32() | *shared/functions/system/ELUsingAArch32* on page J1-5404 |
| ELUsingAArch32K() | *shared/functions/system/ELUsingAArch32K* on page J1-5404 |
| EncodeLDFSC() | *shared/functions/aborts/EncodeLDFSC* on page J1-5366 |
| EncodeSDFSC() | *aarch32/functions/aborts/EncodeSDFSC* on page J1-5302 |
| EndOfInstruction() | *EndOfInstruction()* on page K11-5671<br>*shared/functions/system/EndOfInstruction* on page J1-5404 |
| EnterHypMode() | *aarch32/exceptions/takeexception/AArch32.EnterHypMode* on page J1-5292 |
| EnterHypModeInDebugState() | *Pseudocode description of taking exceptions in Debug state* on page H2-4852<br>*aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState* on page J1-5279 |
| EnterMode() | *aarch32/exceptions/takeexception/AArch32.EnterMode* on page J1-5293 |
| EnterModeInDebugState() | *Pseudocode description of taking exceptions in Debug state* on page H2-4852<br>*aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState* on page J1-5279 |
| EnterMonitorMode() | *aarch32/exceptions/takeexception/AArch32.EnterMonitorMode* on page J1-5293 |
| EnterMonitorModeInDebugState() | *Pseudocode description of taking exceptions in Debug state* on page H2-4852<br>*aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState* on page J1-5280 |
| ESR[] | *aarch64/functions/sysregisters/ESR* on page J1-5246 |
| ESRType[] | *aarch64/functions/sysregisters/ESRType* on page J1-5247 |
| EventRegistered() | *shared/functions/system/EventRegistered* on page J1-5405 |
| EventRegisterSet() | *shared/functions/system/EventRegisterSet* on page J1-5404 |
| Exception | *shared/exceptions/exceptions/Exception* on page J1-5360 |
| ExceptionClass() | *aarch64/exceptions/exceptions/AArch64.ExceptionClass* on page J1-5227<br>*aarch32/exceptions/exceptions/AArch32.ExceptionClass* on page J1-5288 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| ExceptionRecord | *shared/exceptions/exceptions/ExceptionRecord* on page J1-5360 |
| ExceptionReturn() | *aarch64/instrs/branch/eret/AArch64.ExceptionReturn* on page J1-5250<br>*aarch32/functions/system/AArch32.ExceptionReturn* on page J1-5320 |
| ExceptionSyndrome() | *shared/exceptions/exceptions/ExceptionSyndrome* on page J1-5360 |
| ExclusiveMonitorsPass() | *Exclusive monitors operations* on page D3-1733<br>*Exclusive monitors operations* on page G3-3982<br>*aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass* on page J1-5238<br>*aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass* on page J1-5308 |
| ExclusiveMonitorsStatus() | *Exclusive monitors operations* on page D3-1733<br>*Exclusive monitors operations* on page G3-3982<br>*shared/functions/exclusive/ExclusiveMonitorsStatus* on page J1-5376 |
| ExcVectorBase() | *aarch32/exceptions/exceptions/ExcVectorBase* on page J1-5290 |
| ExecuteA64() | *shared/debug/halting/ExecuteA64* on page J1-5353 |
| ExecuteT32() | *shared/debug/halting/ExecuteT32* on page J1-5353 |
| ExitDebugState() | *Exiting Debug state* on page H2-4858<br>*shared/debug/halting/ExitDebugState* on page J1-5353 |
| Extend() | *shared/functions/common/Extend* on page J1-5370 |
| ExtendReg() | *aarch64/instrs/extendreg/ExtendReg* on page J1-5251 |
| ExtendType | *aarch64/instrs/extendreg/ExtendType* on page J1-5252 |
| ExternalDebugRequest() | *Pseudocode description of External Debug Request debug events* on page H3-4878<br>*shared/debug/haltingevents/ExternalDebugRequest* on page J1-5356 |
| ExternalInvasiveDebugEnabled() | *Pseudocode description of External Invasive Debug Enabled* on page K2-5521<br>*shared/debug/authentication/ExternalInvasiveDebugEnabled* on page J1-5347 |
| ExternalNoninvasiveDebugEnabled() | *Pseudocode description of External Non-invasive Debug Enabled* on page K2-5521<br>*shared/debug/authentication/ExternalNoninvasiveDebugEnabled* on page J1-5347 |
| ExternalSecureInvasiveDebugEnabled() | *Pseudocode description of External Secure Invasive Debug Enabled* on page K2-5521<br>*shared/debug/authentication/ExternalSecureInvasiveDebugEnabled* on page J1-5347 |
| ExternalSecureNoninvasiveDebugEnabled() | *Pseudocode description of External Non-invasive Debug Enabled* on page K2-5521<br>*shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled* on page J1-5347 |
| FAR[] | *aarch64/functions/sysregisters/FAR* on page J1-5247 |
| Fault | *shared/functions/memory/Fault* on page J1-5398 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| FaultRecord | *shared/functions/memory/FaultRecord* on page J1-5398 |
| FaultStatusLD() | *aarch32/functions/aborts/AArch32.FaultStatusLD* on page J1-5301 |
| FaultStatusSD() | *aarch32/functions/aborts/AArch32.FaultStatusSD* on page J1-5302 |
| FaultSyndrome() | *shared/functions/aborts/FaultSyndrome* on page J1-5366 |
| FindWatchpoint() | *shared/debug/FindWatchpoint/FindWatchpoint* on page J1-5346 |
| FirstStageTranslate() | *Stage 1 translation* on page D4-1785<br>*aarch64/translation/translation/AArch64.FirstStageTranslate* on page J1-5264<br>*aarch32/translation/translation/AArch32.FirstStageTranslate* on page J1-5334 |
| FixedToFP() | *Floating-point conversions* on page E1-2350<br>*shared/functions/float/fixedtofp/FixedToFP* on page J1-5377 |
| FPAbs() | *Floating-point negation and absolute value* on page E1-2334<br>*shared/functions/float/fpabs/FPAbs* on page J1-5377 |
| FPAdd() | *Floating-point addition and subtraction* on page E1-2342<br>*shared/functions/float/fpadd/FPAdd* on page J1-5377 |
| FPCompare() | *Floating-point comparisons* on page E1-2340<br>*shared/functions/float/fpcompare/FPCompare* on page J1-5378 |
| FPCompareEQ() | *Floating-point comparisons* on page E1-2340<br>*shared/functions/float/fpcompareeq/FPCompareEQ* on page J1-5378 |
| FPCompareGE() | *Floating-point comparisons* on page E1-2340<br>*shared/functions/float/fpcomparege/FPCompareGE* on page J1-5379 |
| FPCompareGT() | *Floating-point comparisons* on page E1-2340<br>*shared/functions/float/fpcomparegt/FPCompareGT* on page J1-5379 |
| FPConvert() | *Floating-point conversions* on page E1-2350<br>*shared/functions/float/fpconvert/FPConvert* on page J1-5379 |
| FPConvertNaN() | *shared/functions/float/fpconvertnan/FPConvertNaN* on page J1-5380 |
| FPConvOp | *aarch64/instrs/float/convert/fpconvop/FPConvOp* on page J1-5252 |
| FPCRType | *shared/functions/float/fpcrtype/FPCRType* on page J1-5380 |
| FPDecodeRM() | *shared/functions/float/fpdecoderm/FPDecodeRM* on page J1-5380 |
| FPDecodeRounding() | *shared/functions/float/fpdecoderounding/FPDecodeRounding* on page J1-5381 |
| FPDefaultNaN() | *shared/functions/float/fpdefaultnan/FPDefaultNaN* on page J1-5381 |
| FPDiv() | *Floating-point multiplication and division* on page E1-2343<br>*shared/functions/float/fpdiv/FPDiv* on page J1-5381 |
| FPExc | *shared/functions/float/fpexc/FPExc* on page J1-5381 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| FPHalvedSub() | *Floating-point reciprocal square root estimate and step* on page E1-2347<br>*aarch32/functions/float/FPHalvedSub* on page J1-5310 |
| FPInfinity() | *Generation of specific floating-point values* on page E1-2333<br>*shared/functions/float/fpinfinity/FPInfinity* on page J1-5381 |
| FPMax() | *Floating-point maximum and minimum* on page E1-2341<br>*shared/functions/float/fpmax/FPMax* on page J1-5382 |
| FPMaxMinOp | *aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp* on page J1-5252 |
| FPMaxNormal() | *Generation of specific floating-point values* on page E1-2333<br>*shared/functions/float/fpmaxnormal/FPMaxNormal* on page J1-5382 |
| FPMaxNum() | *shared/functions/float/fpmaxnum/FPMaxNum* on page J1-5382 |
| FPMin() | *Floating-point maximum and minimum* on page E1-2341<br>*shared/functions/float/fpmin/FPMin* on page J1-5383 |
| FPMinNum() | *shared/functions/float/fpminnum/FPMinNum* on page J1-5383 |
| FPMul() | *Floating-point multiplication and division* on page E1-2343<br>*shared/functions/float/fpmul/FPMul* on page J1-5383 |
| FPMulAdd() | *Floating-point fused multiply-add* on page E1-2343<br>*shared/functions/float/fpmuladd/FPMulAdd* on page J1-5384 |
| FPMulX() | *shared/functions/float/fpmulx/FPMulX* on page J1-5385 |
| FPNeg() | *Floating-point negation and absolute value* on page E1-2334<br>*shared/functions/float/fpneg/FPNeg* on page J1-5385 |
| FPOnePointFive() | *shared/functions/float/fponepointfive/FPOnePointFive* on page J1-5385 |
| FPProcessException() | *Floating-point exception and NaN handling* on page E1-2336<br>*shared/functions/float/fpprocessexception/FPProcessException* on page J1-5385 |
| FPProcessNaN() | *Floating-point exception and NaN handling* on page E1-2336<br>*shared/functions/float/fpprocessnan/FPProcessNaN* on page J1-5386 |
| FPProcessNaNs() | *Floating-point exception and NaN handling* on page E1-2336<br>*shared/functions/float/fpprocessnans/FPProcessNaNs* on page J1-5386 |
| FPProcessNaNs3() | *Floating-point exception and NaN handling* on page E1-2336<br>*shared/functions/float/fpprocessnans3/FPProcessNaNs3* on page J1-5386 |
| FPRecipEstimate() | *Floating-point reciprocal estimate and step* on page E1-2344<br>*shared/functions/float/fprecipestimate/FPRecipEstimate* on page J1-5387 |
| FPRecipStep() | *Floating-point reciprocal estimate and step* on page E1-2344<br>*aarch32/functions/float/FPRecipStep* on page J1-5311 |
| FPRecipStepFused() | *aarch64/functions/fusedrstep/FPRecipStepFused* on page J1-5240 |
| FPRecpX() | *shared/functions/float/fprecpx/FPRecpX* on page J1-5388 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| FPRound() | *Floating-point rounding* on page E1-2338<br>*shared/functions/float/fpround/FPRound* on page J1-5389 |
| FPRounding | *shared/functions/float/fprounding/FPRounding* on page J1-5390 |
| FPRoundingMode() | *shared/functions/float/fproundingmode/FPRoundingMode* on page J1-5390 |
| FPRoundInt() | *shared/functions/float/fproundint/FPRoundInt* on page J1-5391 |
| FPRSqrtEstimate() | *Floating-point reciprocal square root estimate and step* on page E1-2347<br>*shared/functions/float/fprsqrtestimate/FPRSqrtEstimate* on page J1-5391 |
| FPRSqrtStep() | *Floating-point reciprocal square root estimate and step* on page E1-2347<br>*aarch32/functions/float/FPRSqrtStep* on page J1-5310 |
| FPRSqrtStepFused() | *aarch64/functions/fusedrstep/FPRSqrtStepFused* on page J1-5239 |
| FPSqrt() | *Floating-point square root* on page E1-2347<br>*shared/functions/float/fpsqrt/FPSqrt* on page J1-5392 |
| FPSub() | *Floating-point addition and subtraction* on page E1-2342<br>*shared/functions/float/fpsub/FPSub* on page J1-5393 |
| FPThree() | *Generation of specific floating-point values* on page E1-2333<br>*shared/functions/float/fpthree/FPThree* on page J1-5393 |
| FPToFixed() | *Floating-point conversions* on page E1-2350<br>*shared/functions/float/fptofixed/FPToFixed* on page J1-5393 |
| FPTrappedException() | *aarch64/exceptions/ieeefp/AArch64.FPTrappedException* on page J1-5229<br>*aarch32/exceptions/ieeefp/AArch32.FPTrappedException* on page J1-5290 |
| FPTwo() | *Generation of specific floating-point values* on page E1-2333<br>*shared/functions/float/fptwo/FPTwo* on page J1-5394 |
| FPType | *shared/functions/float/fptype/FPType* on page J1-5394 |
| FPUnaryOp | *aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp* on page J1-5252 |
| FPUnpack() | *Floating-point value unpacking* on page E1-2335<br>*shared/functions/float/fpunpack/FPUnpack* on page J1-5394 |
| FPZero() | *Generation of specific floating-point values* on page E1-2333<br>*shared/functions/float/fpzero/FPZero* on page J1-5396 |
| FullAddress | *Memory data type definitions* on page D3-1730<br>*Memory data type definitions* on page G3-3979<br>*shared/functions/memory/FullAddress* on page J1-5398 |
| FullTranslate() | *Performing the full address translation* on page D4-1784<br>*Address translation* on page G4-4183<br>*aarch64/translation/translation/AArch64.FullTranslate* on page J1-5265<br>*aarch32/translation/translation/AArch32.FullTranslate* on page J1-5335 |
| GeneralExceptionsToAArch64() | *aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64* on page J1-5289 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| GenerateAlignmentException() | *GenerateAlignmentException()* on page K11-5671<br>*aarch32/functions/memory/GenerateAlignmentException* on page J1-5312 |
| GenerateDebugExceptions() | *Pseudocode description of enabling debug exceptions* on page D2-1644<br>*Pseudocode description of enabling debug exceptions* on page G2-3890<br>*aarch64/debug/enables/AArch64.GenerateDebugExceptions* on page J1-5217<br>*aarch32/debug/enables/AArch32.GenerateDebugExceptions* on page J1-5277 |
| GenerateDebugExceptionsFrom() | *Pseudocode description of enabling debug exceptions* on page D2-1644<br>*Pseudocode description of enabling debug exceptions* on page G2-3890<br>*aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom* on page J1-5217<br>*aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom* on page J1-5277 |
| GenerateIntegerZeroDivide() | *aarch32/functions/system/GenerateIntegerZeroDivide* on page J1-5323 |
| GetPSRFromPSTATE() | *shared/functions/system/GetPSRFromPSTATE* on page J1-5405 |
| Halt() | *Pseudocode description of entering Debug state* on page H2-4830<br>*shared/debug/halting/Halt* on page J1-5353 |
| Halted() | *Pseudocode description of Halting on debug events* on page H2-4827<br>*shared/debug/halting/Halted* on page J1-5354 |
| HaltingAllowed() | *Pseudocode description of Halting on debug events* on page H2-4827<br>*shared/debug/halting/HaltingAllowed* on page J1-5354 |
| HaltingStep_DidNotStep() | *shared/debug/haltingevents/HaltingStep_DidNotStep* on page J1-5357 |
| HaltingStep_SteppedEX() | *shared/debug/haltingevents/HaltingStep_SteppedEX* on page J1-5357 |
| HaltOnBreakpointOrWatchpoint() | *Pseudocode description of Halting on debug events* on page H2-4827<br>*shared/debug/halting/HaltOnBreakpointOrWatchpoint* on page J1-5354 |
| HasArchVersion() | *shared/functions/system/HasArchVersion* on page J1-5405 |
| HaveAArch32EL() | *shared/functions/system/HaveAArch32EL* on page J1-5405 |
| HaveAnyAArch32() | *shared/functions/system/HaveAnyAArch32* on page J1-5405 |
| HaveCRCExt() | *shared/functions/crc/HaveCRCExt* on page J1-5374 |
| HaveCryptoExt() | *shared/functions/crypto/HaveCryptoExt* on page J1-5375 |
| HaveEL() | *shared/functions/system/HaveEL* on page J1-5406 |
| HighestEL() | *shared/functions/system/HighestEL* on page J1-5406 |
| HighestELUsingAArch32() | *shared/functions/system/HighestELUsingAArch32* on page J1-5406 |
| HighestSetBit() | *Lowest and highest set bits of a bitstring* on page K11-5669<br>*shared/functions/common/HighestSetBit* on page J1-5370 |
| Hint_Branch() | *shared/functions/registers/Hint_Branch* on page J1-5400 |
| Hint_Debug() | *Hint_Debug()* on page K11-5671<br>*shared/functions/system/Hint_Debug* on page J1-5406 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| Hint_Prefetch() | *shared/functions/memory/Hint_Prefetch* on page J1-5398 |
| Hint_PreloadData() | *Hint_PreloadData()* on page K11-5671<br>*aarch32/functions/memory/Hint_PreloadData* on page J1-5312 |
| Hint_PreloadDataForWrite() | *Hint_PreloadDataForWrite()* on page K11-5671<br>*aarch32/functions/memory/Hint_PreloadDataForWrite* on page J1-5312 |
| Hint_PreloadInstr() | *Hint_PreloadInstr()* on page K11-5671<br>*aarch32/functions/memory/Hint_PreloadInstr* on page J1-5312 |
| Hint_Yield() | *Hint_Yield()* on page K11-5671<br>*shared/functions/system/Hint_Yield* on page J1-5406 |
| IllegalExceptionReturn() | *shared/functions/system/IllegalExceptionReturn* on page J1-5406 |
| ImmediateOp | *aarch64/instrs/vector/logical/immediateop/ImmediateOp* on page J1-5256 |
| InITBlock() | *Pseudocode description of PSTATE PE state fields* on page E1-2322<br>*aarch32/functions/system/InITBlock* on page J1-5323 |
| InstrSet | *shared/functions/system/InstrSet* on page J1-5407 |
| InstructionAbort() | *Abort exceptions* on page D3-1736<br>*aarch64/exceptions/aborts/AArch64.InstructionAbort* on page J1-5222 |
| InstructionDevice() | *aarch64/translation/attrs/AArch64.InstructionDevice* on page J1-5258<br>*aarch32/translation/attrs/AArch32.InstructionDevice* on page J1-5326 |
| InstructionSynchronizationBarrier() | *shared/functions/system/InstructionSynchronizationBarrier* on page J1-5407 |
| Int() | *shared/functions/common/Int* on page J1-5370 |
| IntegerZeroDivideTrappingEnabled() | *IntegerZeroDivideTrappingEnabled()* on page K11-5671<br>*aarch32/functions/system/IntegerZeroDivideTrappingEnabled* on page J1-5323 |
| InterruptID | *shared/debug/interrupts/InterruptID* on page J1-5357 |
| InterruptPending() | *shared/functions/system/InterruptPending* on page J1-5407 |
| IPAValid() | *shared/functions/aborts/IPAValid* on page J1-5367 |
| IsAsyncAbort() | *IsAsyncAbort()* on page K11-5672<br>*shared/functions/aborts/IsAsyncAbort* on page J1-5367 |
| IsDebugException() | *shared/functions/aborts/IsDebugException* on page J1-5367 |
| IsExclusiveGlobal() | *Exclusive monitors operations* on page D3-1733<br>*Exclusive monitors operations* on page G3-3982<br>*shared/functions/exclusive/IsExclusiveGlobal* on page J1-5377 |
| IsExclusiveLocal() | *Exclusive monitors operations* on page D3-1733<br>*Exclusive monitors operations* on page G3-3982<br>*shared/functions/exclusive/IsExclusiveLocal* on page J1-5377 |
| IsExclusiveVA() | *aarch64/functions/exclusive/AArch64.IsExclusiveVA* on page J1-5239<br>*aarch32/functions/exclusive/AArch32.IsExclusiveVA* on page J1-5309 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| IsExternalAbort() | *IsExternalAbort()* on page K11-5672<br>*shared/functions/aborts/IsExternalAbort* on page J1-5367 |
| IsFault() | *shared/functions/aborts/IsFault* on page J1-5367 |
| IsOnes() | *Testing a bitstring for being all zero or all ones* on page K11-5669<br>*shared/functions/common/IsOnes* on page J1-5370 |
| IsSecondStage() | *shared/functions/aborts/IsSecondStage* on page J1-5368 |
| IsSecure() | *shared/functions/system/IsSecure* on page J1-5407 |
| IsSecureBelowEL3() | *shared/functions/system/IsSecureBelowEL3* on page J1-5407 |
| IsZero() | *Testing a bitstring for being all zero or all ones* on page K11-5669<br>*shared/functions/common/IsZero* on page J1-5370 |
| IsZeroBit() | *Testing a bitstring for being all zero or all ones* on page K11-5669<br>*shared/functions/common/IsZeroBit* on page J1-5370 |
| ITAdvance() | *Pseudocode description of PSTATE PE state fields* on page E1-2322<br>*aarch32/functions/system/AArch32.ITAdvance* on page J1-5321 |
| LastInITBlock() | *Pseudocode description of PSTATE PE state fields* on page E1-2322<br>*aarch32/functions/system/LastInITBlock* on page J1-5323 |
| LoadWritePC() | *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318<br>*aarch32/functions/registers/LoadWritePC* on page J1-5317 |
| LogicalOp | *aarch64/instrs/logicalop/LogicalOp* on page J1-5254 |
| LongConvertAttrsHints() | *shared/translation/attrs/LongConvertAttrsHints* on page J1-5415 |
| LookUpRIndex() | *aarch32/functions/registers/LookUpRIndex* on page J1-5317 |
| LowestSetBit() | *Lowest and highest set bits of a bitstring* on page K11-5669<br>*shared/functions/common/LowestSetBit* on page J1-5371 |
| LR | *aarch32/functions/registers/LR* on page J1-5317 |
| LR_mon | *aarch32/functions/registers/Monitor_mode_registers* on page J1-5318 |
| LSInstructionSyndrome() | *LSInstructionSyndrome()* on page K11-5672<br>*shared/functions/aborts/LSInstructionSyndrome* on page J1-5368 |
| LSL_C() | *Pseudocode description of shift and rotate operations* on page E1-2313<br>*shared/functions/common/LSL_C* on page J1-5371 |
| LSL() | *Pseudocode description of shift and rotate operations* on page E1-2313<br>*shared/functions/common/LSL* on page J1-5370 |
| LSR_C() | *Pseudocode description of shift and rotate operations* on page E1-2313<br>*shared/functions/common/LSR_C* on page J1-5371 |
| LSR() | *Pseudocode description of shift and rotate operations* on page E1-2313<br>*shared/functions/common/LSR* on page J1-5371 |
| MAIR[] | *aarch64/functions/sysregisters/MAIR* on page J1-5247 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| `MAIRType[]` | *aarch64/functions/sysregisters/MAIRType* on page J1-5248 |
| `MarkExclusiveGlobal()` | *Exclusive monitors operations* on page D3-1733<br>*Exclusive monitors operations* on page G3-3982<br>*shared/functions/exclusive/MarkExclusiveGlobal* on page J1-5377 |
| `MarkExclusiveLocal()` | *Exclusive monitors operations* on page D3-1733<br>*Exclusive monitors operations* on page G3-3982<br>*shared/functions/exclusive/MarkExclusiveLocal* on page J1-5377 |
| `MarkExclusiveVA()` | *aarch64/functions/exclusive/AArch64.MarkExclusiveVA* on page J1-5239<br>*aarch32/functions/exclusive/AArch32.MarkExclusiveVA* on page J1-5309 |
| `Max()` | *Maximum and minimum* on page K11-5670<br>*shared/functions/common/Max* on page J1-5371 |
| `MaybeZeroRegisterUppers()` | *aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers* on page J1-5242 |
| `MBReqDomain` | *Memory barriers* on page D3-1738<br>*Memory barriers* on page G3-3987<br>*shared/functions/memory/MBReqDomain* on page J1-5399 |
| `MBReqTypes` | *Memory barriers* on page D3-1738<br>*Memory barriers* on page G3-3987<br>*shared/functions/memory/MBReqTypes* on page J1-5399 |
| `Mem_with_type[]` | *Unaligned memory access* on page G3-3981<br>*aarch32/functions/memory/Mem_with_type* on page J1-5314 |
| `Mem[]` | *Unaligned memory access* on page D3-1732<br>*aarch64/functions/memory/Mem* on page J1-5241<br>*aarch32/functions/memory/MemA* on page J1-5313 |
| `MemAttrDefaults()` | *shared/translation/attrs/MemAttrDefaults* on page J1-5415 |
| `MemAttrHints` | *Memory data type definitions* on page D3-1730<br>*Memory data type definitions* on page G3-3979<br>*shared/functions/memory/MemAttrHints* on page J1-5399 |
| `MemBarrierOp` | *aarch64/instrs/system/barriers/barrierop/MemBarrierOp* on page J1-5255 |
| `MemO[]` | *aarch32/functions/memory/MemO* on page J1-5313 |
| `MemOp` | *aarch64/instrs/memory/memop/MemOp* on page J1-5254 |
| `MemoryAttributes` | *Memory data type definitions* on page D3-1730<br>*Memory data type definitions* on page G3-3979<br>*shared/functions/memory/MemoryAttributes* on page J1-5399 |
| `MemSingle[]` | *Aligned memory access* on page D3-1731<br>*Aligned memory access* on page G3-3980<br>*aarch64/functions/memory/AArch64.MemSingle* on page J1-5240<br>*aarch32/functions/memory/AArch32.MemSingle* on page J1-5311 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| MemType | *Memory data type definitions* on page D3-1730<br>*Memory data type definitions* on page G3-3979<br>*shared/functions/memory/MemType* on page J1-5399 |
| MemU_unpriv[] | *aarch32/functions/memory/MemU_unpriv* on page J1-5313 |
| MemU[] | *aarch32/functions/memory/MemU* on page J1-5313 |
| Min() | *Maximum and minimum* on page K11-5670<br>*shared/functions/common/Min* on page J1-5372 |
| ModeBits | *shared/functions/system/Mode_Bits* on page J1-5408 |
| MonitorModeTrap() | *aarch64/exceptions/traps/AArch64.MonitorModeTrap* on page J1-5235 |
| MoveWideOp | *aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp* on page J1-5253 |
| MoveWidePreferred() | *aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred* on page J1-5253 |
| NextInstrAddr() | *shared/functions/registers/NextInstrAddr* on page J1-5400 |
| NoFault() | *Abort exceptions* on page D3-1736<br>*aarch64/translation/faults/AArch64.NoFault* on page J1-5263<br>*aarch32/translation/faults/AArch32.NoFault* on page J1-5334 |
| NOT() | *Logical operations on bitstrings* on page K11-5657<br>*shared/functions/common/NOT* on page J1-5372 |
| Ones() | *shared/functions/common/Ones* on page J1-5372 |
| PAMax() | *Basic memory access* on page D3-1731<br>*Basic memory access* on page G3-3980<br>*shared/translation/translation/PAMax* on page J1-5418 |
| PC | *aarch32/functions/registers/PC* on page J1-5318 |
| PC[] | *aarch64/functions/registers/PC* on page J1-5244 |
| PCAlignmentFault() | *aarch64/exceptions/aborts/AArch64.PCAlignmentFault* on page J1-5222 |
| PCSample() | *Pseudocode description of Sample-based Profiling* on page H7-4940<br>*shared/debug/samplebasedprofiling/PCSample* on page J1-5358 |
| PCStoreValue() | *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-2318<br>*aarch32/functions/registers/PCStoreValue* on page J1-5318 |
| PermissionFault() | *Pseudocode description of the MMU faults* on page D4-1832<br>*aarch64/translation/faults/AArch64.PermissionFault* on page J1-5263<br>*aarch32/translation/faults/AArch32.PermissionFault* on page J1-5334 |
| Permissions | *Memory data type definitions* on page D3-1730<br>*Memory data type definitions* on page G3-3979<br>*shared/functions/memory/Permissions* on page J1-5399 |
| PLOfEL() | *shared/functions/system/PLOfEL* on page J1-5408 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| `Poly32Mod2()` | *shared/functions/crc/Poly32Mod2* on page J1-5374 |
| `PolynomialMult()` | *shared/functions/vector/PolynomialMult* on page J1-5412 |
| `Prefetch()` | *aarch64/instrs/memory/prefetch/Prefetch* on page J1-5254 |
| `PrefetchHint` | *shared/functions/memory/PrefetchHint* on page J1-5399 |
| `PrivilegeLevel` | *shared/functions/system/PrivilegeLevel* on page J1-5408 |
| `ProcessorID()` | *ProcessorID()* on page K11-5672<br>*shared/functions/exclusive/ProcessorID* on page J1-5377 |
| `ProcState` | *shared/functions/system/ProcState* on page J1-5408 |
| `ProfilingProhibited()` | *aarch64/debug/pmu/AArch64.ProfilingProhibited* on page J1-5218<br>*aarch32/debug/pmu/AArch32.ProfilingProhibited* on page J1-5278 |
| `PSTATE` | *shared/functions/system/PSTATE* on page J1-5408 |
| `PSTATEField` | *aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField* on page J1-5255 |
| `Q[]` | *aarch32/functions/registers/Q* on page J1-5318 |
| `Qin[]` | *Pseudocode description of the SIMD and Floating-point register file* on page E1-2327<br>*aarch32/functions/registers/Qin* on page J1-5318 |
| `R[]` | *aarch32/functions/registers/R* on page J1-5318 |
| `RBankSelect()` | *aarch32/functions/registers/RBankSelect* on page J1-5319 |
| `Reduce()` | *aarch64/instrs/vector/reduce/reduceop/Reduce* on page J1-5256 |
| `ReduceOp` | *aarch64/instrs/vector/reduce/reduceop/ReduceOp* on page J1-5257 |
| `RemappedTEXDecode()` | *Memory access decode when TEX remap is enabled* on page G4-4199<br>*aarch32/translation/attrs/AArch32.RemappedTEXDecode* on page J1-5326 |
| `RemapRegsHaveResetValues()` | *RemapRegsHaveResetValues()* on page K11-5672<br>*aarch32/translation/walk/RemapRegsHaveResetValues* on page J1-5343 |
| `Replicate()` | *shared/functions/common/Replicate* on page J1-5373 |
| `ReportDataAbort()` | *aarch32/exceptions/aborts/AArch32.ReportDataAbort* on page J1-5283 |
| `ReportException()` | *aarch64/exceptions/exceptions/AArch64.ReportException* on page J1-5227 |
| `ReportHypEntry()` | *Reporting syndrome information* on page G4-4197<br>*aarch32/exceptions/exceptions/AArch32.ReportHypEntry* on page J1-5289 |
| `ReportPrefetchAbort()` | *aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort* on page J1-5284 |
| `ReservedValue()` | *shared/exceptions/traps/ReservedValue* on page J1-5361 |
| `ResetControlRegisters()` | *aarch64/exceptions/exceptions/AArch64.ResetControlRegisters* on page J1-5228<br>*aarch32/exceptions/exceptions/AArch32.ResetControlRegisters* on page J1-5289 |
| `ResetExternalDebugRegisters()` | *shared/functions/registers/ResetExternalDebugRegisters* on page J1-5400 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| ResetGeneralRegisters() | *aarch64/functions/registers/AArch64.ResetGeneralRegisters* on page J1-5243<br>*aarch32/functions/registers/AArch32.ResetGeneralRegisters* on page J1-5315 |
| ResetSIMDFPRegisters() | *aarch64/functions/registers/AArch64.ResetSIMDFPRegisters* on page J1-5243<br>*aarch32/functions/registers/AArch32.ResetSIMDFPRegisters* on page J1-5315 |
| ResetSpecialRegisters() | *aarch64/functions/registers/AArch64.ResetSpecialRegisters* on page J1-5243<br>*aarch32/functions/registers/AArch32.ResetSpecialRegisters* on page J1-5315 |
| ResetSystemRegisters() | *aarch64/functions/registers/AArch64.ResetSystemRegisters* on page J1-5244<br>*aarch32/functions/registers/AArch32.ResetSystemRegisters* on page J1-5316 |
| Restarting() | *Pseudocode description of Halting on debug events* on page H2-4827<br>*shared/debug/halting/Restarting* on page J1-5354 |
| Rmode[] | *aarch32/functions/registers/Rmode* on page J1-5319 |
| ROL() | *shared/functions/crypto/ROL* on page J1-5375 |
| ROR_C() | *Pseudocode description of shift and rotate operations* on page E1-2313<br>*shared/functions/common/ROR_C* on page J1-5372 |
| ROR() | *Pseudocode description of shift and rotate operations* on page E1-2313<br>*shared/functions/common/ROR* on page J1-5372 |
| RoundDown() | *Rounding and aligning* on page K11-5670<br>*shared/functions/common/RoundDown* on page J1-5373 |
| RoundTowardsZero() | *Rounding and aligning* on page K11-5670<br>*shared/functions/common/RoundTowardsZero* on page J1-5373 |
| RoundUp() | *Rounding and aligning* on page K11-5670<br>*shared/functions/common/RoundUp* on page J1-5373 |
| RRX_C() | *Pseudocode description of shift and rotate operations* on page E1-2313<br>*aarch32/functions/common/RRX_C* on page J1-5304 |
| RRX() | *Pseudocode description of shift and rotate operations* on page E1-2313<br>*aarch32/functions/common/RRX* on page J1-5304 |
| RunHaltingStep() | *Pseudocode description of Halting Step debug events* on page H3-4873<br>*shared/debug/haltingevents/RunHaltingStep* on page J1-5357 |
| S[] | *Pseudocode description of the SIMD and Floating-point register file* on page E1-2327<br>*aarch32/functions/registers/S* on page J1-5320 |
| S1AttrDecode() | *Support functions* on page D4-1793<br>*Translation table walk using the Long-descriptor translation table format for stage 1* on page G4-4189<br>*aarch64/translation/attrs/AArch64.S1AttrDecode* on page J1-5258<br>*aarch32/translation/attrs/AArch32.S1AttrDecode* on page J1-5327 |
| S1TranslationRegime() | *shared/translation/translation/S1TranslationRegime* on page J1-5418 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| S2AttrDecode() | *Translation table walk using the Long-descriptor translation table format for stage 1* on page G4-4189<br>*shared/translation/attrs/S2AttrDecode* on page J1-5416 |
| S1CacheDisabled() | *shared/translation/attrs/S1CacheDisabled* on page J1-5416 |
| S2CacheDisabled() | *shared/translation/attrs/S2CacheDisabled* on page J1-5417 |
| S2ConvertAttrsHints() | *shared/translation/attrs/S2ConvertAttrsHints* on page J1-5417 |
| Sat() | *Pseudocode description of saturation* on page E1-2315<br>*aarch32/functions/v6simd/Sat* on page J1-5324 |
| SatQ() | *Pseudocode description of saturation* on page E1-2315<br>*shared/functions/vector/SatQ* on page J1-5412 |
| SCR_GEN | *shared/functions/system/SCR_GEN* on page J1-5409 |
| SCRType | *shared/functions/system/SCRType* on page J1-5409 |
| SCTLR[] | *aarch64/functions/sysregisters/SCTLR* on page J1-5248 |
| SCTRLType | *aarch64/functions/sysregisters/SCTLRType* on page J1-5248 |
| SecondStageTranslate() | *Stage 2 translation* on page D4-1787<br>*Stage 2 translation table walk* on page G4-4194<br>*aarch64/translation/translation/AArch64.SecondStageTranslate* on page J1-5265<br>*aarch32/translation/translation/AArch32.SecondStageTranslate* on page J1-5336 |
| SecondStageWalk() | *Stage 2 translation* on page D4-1787<br>*Stage 2 translation table walk* on page G4-4194<br>*aarch64/translation/translation/AArch64.SecondStageWalk* on page J1-5266<br>*aarch32/translation/translation/AArch32.SecondStageWalk* on page J1-5336 |
| SelectInstrSet() | *Pseudocode description of PSTATE PE state fields* on page E1-2322<br>*aarch32/functions/system/SelectInstrSet* on page J1-5324 |
| SelfHostedSecurePrivilegedInvasiveDebugEnabled() | *Pseudocode description of AArch32 Self-Hosted Secure Privileged Invasive Debug Enabled* on page K2-5521<br>*aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled* on page J1-5274 |
| SendEvent() | *shared/functions/system/SendEvent* on page J1-5409 |
| SetExclusiveMonitors() | *Exclusive monitors operations* on page D3-1733<br>*Exclusive monitors operations* on page G3-3982<br>*aarch64/functions/exclusive/AArch64.SetExclusiveMonitors* on page J1-5239<br>*aarch32/functions/exclusive/AArch32.SetExclusiveMonitors* on page J1-5309 |
| SetInterruptRequestLevel() | *shared/debug/interrupts/SetInterruptRequestLevel* on page J1-5357 |
| SetPSTATEFromPSR() | *shared/functions/system/SetPSTATEFromPSR* on page J1-5409 |
| SHA256hash() | *shared/functions/crypto/SHA256hash* on page J1-5375 |
| SHAchoose() | *shared/functions/crypto/SHAchoose* on page J1-5376 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| SHAhashSIGMA0() | *shared/functions/crypto/SHAhashSIGMA0* on page J1-5376 |
| SHAhashSIGMA1() | *shared/functions/crypto/SHAhashSIGMA1* on page J1-5376 |
| SHAmajority() | *shared/functions/crypto/SHAmajority* on page J1-5376 |
| SHAparity() | *shared/functions/crypto/SHAparity* on page J1-5376 |
| Shift_C() | *Pseudocode description of instruction-specified shifts and rotates* on page F2-2450<br>*aarch32/functions/common/Shift_C* on page J1-5304 |
| Shift() | *Pseudocode description of instruction-specified shifts and rotates* on page F2-2450<br>*aarch32/functions/common/Shift* on page J1-5304 |
| ShiftReg() | *aarch64/instrs/integer/shiftreg/ShiftReg* on page J1-5254 |
| ShiftType | *aarch64/instrs/integer/shiftreg/ShiftType* on page J1-5254 |
| ShortConvertAttrsHints() | *Translation table walk using the Short-descriptor translation table format for stage 1* on page G4-4186<br>*shared/translation/attrs/ShortConvertAttrsHints* on page J1-5417 |
| SignedSat() | *Pseudocode description of saturation* on page E1-2315<br>*aarch32/functions/v6simd/SignedSat* on page J1-5324 |
| SignedSatQ() | *Pseudocode description of saturation* on page E1-2315<br>*shared/functions/vector/SignedSatQ* on page J1-5412 |
| SignExtend() | *Zero-extension and sign-extension of bitstrings* on page K11-5669<br>*shared/functions/common/SignExtend* on page J1-5373 |
| SInt() | *shared/functions/common/SInt* on page J1-5373 |
| SoftwareBreakpoint() | *Pseudocode description of Software Breakpoint Instruction exceptions* on page D2-1651<br>*Pseudocode description of Software Breakpoint Instruction exceptions* on page G2-3897<br>*aarch64/exceptions/debug/AArch64.SoftwareBreakpoint* on page J1-5225<br>*aarch32/exceptions/debug/AArch32.SoftwareBreakpoint* on page J1-5288 |
| SoftwareStep_DidNotStep() | *shared/debug/softwarestep/SoftwareStep_DidNotStep* on page J1-5359 |
| SoftwareStep_SteppedEX() | *shared/debug/softwarestep/SoftwareStep_SteppedEX* on page J1-5360 |
| SoftwareStepException() | *Pseudocode description of debug exceptions* on page D2-1648<br>*aarch64/exceptions/debug/AArch64.SoftwareStepException* on page J1-5226 |
| SP | *aarch32/functions/registers/SP* on page J1-5320 |
| SP_mon | *aarch32/functions/registers/Monitor_mode_registers* on page J1-5318 |
| SP[] | *aarch64/functions/registers/SP* on page J1-5244 |
| SPAlignmentFault() | *aarch64/exceptions/aborts/AArch64.SPAlignmentFault* on page J1-5223 |
| SPSR[] | *shared/functions/sysregisters/SPSR* on page J1-5401 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| SPSRaccessValid() | *Pseudocode support for the Banked register transfer instructions* on page F6-3205<br>*aarch32/functions/system/SPSRaccessValid* on page J1-5323 |
| SPSRWriteByInstr() | *aarch32/functions/system/SPSRWriteByInstr* on page J1-5323 |
| SRType | *aarch32/functions/common/SRType* on page J1-5304 |
| SSAdvance() | *Pseudocode description of Software Step exceptions* on page D2-1699<br>*shared/debug/softwarestep/SSAdvance* on page J1-5359 |
| StandardFPSCRValue() | *Selection of ARM standard floating-point arithmetic* on page E1-2339<br>*aarch32/functions/float/StandardFPSCRValue* on page J1-5311 |
| StateMatch() | *Pseudocode description of Breakpoint exceptions taken from AArch64 state* on page D2-1666<br>*Pseudocode description of Breakpoint exceptions taken from AArch32 state* on page G2-3918<br>*aarch64/debug/breakpoint/AArch64.StateMatch* on page J1-5216<br>*aarch32/debug/breakpoint/AArch32.StateMatch* on page J1-5276 |
| StopInstructionPrefetchAndEnableITR() | *shared/debug/halting/StopInstructionPrefetchAndEnableITR* on page J1-5355 |
| SynchronizeContext() | *shared/functions/system/SynchronizeContext* on page J1-5410 |
| SysOp_R() | *aarch64/functions/system/SysOp_R* on page J1-5250 |
| SysOp_W() | *aarch64/functions/system/SysOp_W* on page J1-5250 |
| SysOp() | *aarch64/instrs/system/sysops/sysop/SysOp* on page J1-5255 |
| System_Get() | *aarch64/functions/system/System_Get* on page J1-5250 |
| System_Put() | *aarch64/functions/system/System_Put* on page J1-5250 |
| SystemHintOp | *aarch64/instrs/system/hints/syshintop/SystemHintOp* on page J1-5255 |
| SystemOp | *aarch64/instrs/system/sysops/sysop/SystemOp* on page J1-5256 |
| SystemRegisterTrap() | *aarch64/exceptions/traps/AArch64.SystemRegisterTrap* on page J1-5236 |
| T32ExpandImm_C() | *Operation of modified immediate constants, T32 instructions* on page F2-2460<br>*aarch32/functions/common/T32ExpandImm_C* on page J1-5305 |
| T32ExpandImm() | *Operation of modified immediate constants, T32 instructions* on page F2-2460<br>*aarch32/functions/common/T32ExpandImm* on page J1-5305 |
| TakeDataAbortException() | *aarch32/exceptions/aborts/AArch32.TakeDataAbortException* on page J1-5284 |
| TakeException() | *aarch64/exceptions/takeexception/AArch64.TakeException* on page J1-5230 |
| TakeExceptionInDebugState() | *Pseudocode description of taking exceptions in Debug state* on page H2-4852<br>*aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState* on page J1-5218 |
| TakeHVCException() | *aarch32/exceptions/syscalls/AArch32.TakeHVCException* on page J1-5291 |
| TakeHypTrapException() | *aarch32/exceptions/traps/AArch32.TakeHypTrapException* on page J1-5298 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| TakeMonitorTrapException() | *aarch32/exceptions/traps/AArch32.TakeMonitorTrapException* on page J1-5298 |
| TakePhysicalAsynchAbortException() | *aarch32/exceptions/asynch/AArch32.TakePhysicalAsynchAbortException* on page J1-5285 |
| TakePhysicalFIQException() | *aarch64/exceptions/asynch/AArch64.TakePhysicalFIQException* on page J1-5223<br>*aarch32/exceptions/asynch/AArch32.TakePhysicalFIQException* on page J1-5286 |
| TakePhysicalIRQException() | *aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException* on page J1-5223<br>*aarch32/exceptions/asynch/AArch32.TakePhysicalIRQException* on page J1-5286 |
| TakePhysicalSystemErrorException() | *aarch64/exceptions/asynch/AArch64.TakePhysicalSystemErrorException* on page J1-5224 |
| TakePrefetchAbortException() | *aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException* on page J1-5284 |
| TakeReset() | *aarch64/exceptions/exceptions/AArch64.TakeReset* on page J1-5228<br>*aarch32/exceptions/exceptions/AArch32.TakeReset* on page J1-5289 |
| TakeSMCException() | *aarch32/exceptions/syscalls/AArch32.TakeSMCException* on page J1-5291 |
| TakeSVCException() | *aarch32/exceptions/syscalls/AArch32.TakeSVCException* on page J1-5292 |
| TakeUndefInstrException() | *aarch32/exceptions/traps/AArch32.TakeUndefInstrException* on page J1-5298 |
| TakeVirtualAsynchAbortException() | *aarch32/exceptions/asynch/AArch32.TakeVirtualAsynchAbortException* on page J1-5287 |
| TakeVirtualFIQException() | *aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException* on page J1-5224<br>*aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException* on page J1-5287 |
| TakeVirtualIRQException() | *aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException* on page J1-5224<br>*aarch32/exceptions/asynch/AArch32.TakeVirtualIRQException* on page J1-5287 |
| TakeVirtualSystemErrorException() | *aarch64/exceptions/asynch/AArch64.TakeVirtualSystemErrorException* on page J1-5225 |
| ThisInstr() | *ThisInstr()* on page K11-5672<br>*shared/functions/system/ThisInstr* on page J1-5410 |
| ThisInstrAddr() | *shared/functions/registers/ThisInstrAddr* on page J1-5400 |
| ThisInstrLength() | *ThisInstrLength()* on page K11-5672<br>*shared/functions/system/ThisInstrLength* on page J1-5410 |
| TLBRecord | *Definitions required for address translation* on page D4-1784<br>*TLB operations* on page G4-4185<br>*shared/functions/memory/TLBRecord* on page J1-5399 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| TranslateAddress() | *Address translation* on page G4-4183<br>*aarch64/translation/translation/AArch64.TranslateAddress* on page J1-5266<br>*aarch32/translation/translation/AArch32.TranslateAddress* on page J1-5337 |
| TranslateAddressS1Off() | *Stage 1 translation* on page D4-1785<br>*Address translation when the stage 1 address translation is disabled* on page G4-4184<br>*aarch64/translation/attrs/AArch64.TranslateAddressS1Off* on page J1-5259<br>*aarch32/translation/attrs/AArch32.TranslateAddressS1Off* on page J1-5328 |
| TranslationFault() | *Pseudocode description of the MMU faults* on page D4-1832<br>*aarch64/translation/faults/AArch64.TranslationFault* on page J1-5264<br>*aarch32/translation/faults/AArch32.TranslationFault* on page J1-5334 |
| TranslationTableWalk() | *Translation table walk* on page D4-1788<br>*aarch64/translation/walk/AArch64.TranslationTableWalk* on page J1-5266 |
| TranslationTableWalkLD() | *Translation table walk using the Long-descriptor translation table format for stage 1* on page G4-4189<br>*aarch32/translation/walk/AArch32.TranslationTableWalkLD* on page J1-5337 |
| TranslationTableWalkSD() | *Translation table walk using the Short-descriptor translation table format for stage 1* on page G4-4186<br>*aarch32/translation/walk/AArch32.TranslationTableWalkSD* on page J1-5341 |
| UInt() | *shared/functions/common/UInt* on page J1-5373 |
| UnallocatedEncoding() | *shared/exceptions/traps/UnallocatedEncoding* on page J1-5361 |
| UndefinedFault() | *aarch64/exceptions/traps/AArch64.UndefinedFault* on page J1-5236<br>*aarch32/exceptions/traps/AArch32.UndefinedFault* on page J1-5298 |
| Unreachable() | *shared/functions/system/Unreachable* on page J1-5410 |
| UnsignedRecipEstimate() | *Floating-point reciprocal estimate and step* on page E1-2344<br>*shared/functions/vector/UnsignedRecipEstimate* on page J1-5413 |
| UnsignedRSqrtEstimate() | *Floating-point reciprocal square root estimate and step* on page E1-2347<br>*shared/functions/vector/UnsignedRSqrtEstimate* on page J1-5412 |
| UnsignedSat() | *Pseudocode description of saturation* on page E1-2315<br>*aarch32/functions/v6simd/UnsignedSat* on page J1-5324 |
| UnsignedSatQ() | *Pseudocode description of saturation* on page E1-2315<br>*shared/functions/vector/UnsignedSatQ* on page J1-5413 |
| UpdateEDSCRFields() | *Pseudocode description of entering Debug state* on page H2-4830<br>*shared/debug/halting/UpdateEDSCRFields* on page J1-5355 |
| UsingAArch32() | *shared/functions/system/UsingAArch32* on page J1-5410 |
| V[] | *aarch64/functions/registers/V* on page J1-5244 |
| VBAR[] | *aarch64/functions/sysregisters/VBAR* on page J1-5248 |
| VBitOp | *aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp* on page J1-5256 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K12-1 Pseudocode functions and procedures (continued)**

| Function | See |
|---|---|
| VCRMatch() | *Pseudocode description of Vector Catch exceptions* on page G2-3943<br>*aarch32/debug/VCRMatch/AArch32.VCRMatch* on page J1-5273 |
| VectorCatchException() | *Pseudocode description of debug exceptions* on page D2-1648<br>*Vector Catch exceptions* on page D2-1685<br>*aarch64/exceptions/debug/AArch64.VectorCatchException* on page J1-5226 |
| VFPExpandImm() | *Operation of modified immediate constants, floating-point* on page F5-2567<br>*shared/functions/float/vfpexpandimm/VFPExpandImm* on page J1-5396 |
| Vpart[] | *aarch64/functions/registers/Vpart* on page J1-5245 |
| WaitForEvent() | *shared/functions/system/WaitForEvent* on page J1-5410 |
| WaitForInterrupt() | *shared/functions/system/WaitForInterrupt* on page J1-5411 |
| WalkAttrDecode() | *Support functions* on page D4-1793<br>*shared/translation/attrs/WalkAttrDecode* on page J1-5418 |
| WatchpointByteMatch() | *Pseudocode description of Watchpoint exceptions taken from AArch64 state* on page D2-1682<br>*Pseudocode description of Watchpoint exceptions taken from AArch32 state* on page G2-3934<br>*aarch64/debug/watchpoint/AArch64.WatchpointByteMatch* on page J1-5219<br>*aarch32/debug/watchpoint/AArch32.WatchpointByteMatch* on page J1-5280 |
| WatchpointException() | *Pseudocode description of debug exceptions* on page D2-1648<br>*Pseudocode description of Watchpoint exceptions taken from AArch64 state* on page D2-1682<br>*aarch64/exceptions/debug/AArch64.WatchpointException* on page J1-5226 |
| WatchpointMatch() | *Pseudocode description of Watchpoint exceptions taken from AArch64 state* on page D2-1682<br>*Pseudocode description of Watchpoint exceptions taken from AArch32 state* on page G2-3934<br>*aarch64/debug/watchpoint/AArch64.WatchpointMatch* on page J1-5220<br>*aarch32/debug/watchpoint/AArch32.WatchpointMatch* on page J1-5281 |
| WFxTrap() | *aarch64/exceptions/traps/AArch64.WFxTrap* on page J1-5236 |
| WriteMode() | *aarch32/functions/system/AArch32.WriteMode* on page J1-5321 |
| X[] | *aarch64/functions/registers/X* on page J1-5245 |
| ZeroExtend() | *Zero-extension and sign-extension of bitstrings* on page K11-5669<br>*shared/functions/common/ZeroExtend* on page J1-5374 |
| Zeros() | *shared/functions/common/Zeros* on page J1-5374 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Appendix K13
# **Registers Index**

This appendix provides indexes to the register descriptions in this manual. It contains the following sections:


- *Introduction and register disambiguation* on page K13-5708.
- *Alphabetical index of AArch64 registers and system instructions* on page K13-5713.
- *Functional index of AArch64 registers and system instructions* on page K13-5722.
- *Alphabetical index of AArch32 registers and system instructions* on page K13-5732.
- *Functional index of AArch32 registers and system instructions* on page K13-5740.
- *Alphabetical index of memory-mapped registers* on page K13-5750.
- *Functional index of memory-mapped registers* on page K13-5755.



*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K13.1 Introduction and register disambiguation

In some sections of this manual, registers are referred to by a *general name*, where the description applies to more than one context. Generally, this is one of the following:

- The description applies to both AArch32 state and AArch64 state, and therefore the register names could apply to either AArch32 System registers or AArch64 System registers.
- The description applies to multiple Exception levels, and therefore at a particular Exception level the register names need to take the appropriate Exception. level suffix, _EL0, _EL1, _EL2, or _EL3.

The following sections disambiguate the general register names:

- *Register name disambiguation by Execution state*.
- *Register name disambiguation by Exception level* on page K13-5711.

### K13.1.1 Register name disambiguation by Execution state

Table K13-1 disambiguates the general names of the registers by Execution state.

**Table K13-1 Disambiguation of general names of registers by Execution state**

| General name | Short description | AArch64 register | AArch32 register |
|---|---|---|---|
| CONTEXTIDR | Context ID | CONTEXTIDR_EL1 | CONTEXTIDR |
| DBGBCR | Debug Breakpoint Control Registers | DBGBCR<n>_EL1 | DBGBCR<n> |
| DBGBVR | Debug Breakpoint Value Registers | DBGBVR<n>_EL1 | DBGBVR<n><br>DBGBXVR<n> |
| DBGCLAIMCLR | Debug CLAIM Tag Clear register | DBGCLAIMCLR_EL1 | DBGCLAIMCLR |
| DBGCLAIMSET | Debug CLAIM Tag Set register | DBGCLAIMSET_EL1 | DBGCLAIMSET |
| DBGDTRRX | Debug Data Transfer Register, Receive | DBGDTRRX_EL0 | DBGDTRRXint |
| DBGDTRTX | Debug Data Transfer Register, Transmit | DBGDTRTX_EL0 | DBGDTRTXint |
| DBGPRCR | Debug Power Control Register | DBGPRCR_EL1 | DBGPRCR |
| DBGVCR | Debug Vector Catch Register | DBGVCR32_EL2 | DBGVCR |
| DBGWCR | Debug Watchpoint Control Registers | DBGWCR<n>_EL1 | DBGWCR<n> |
| DBGWVR | Debug Watchpoint Value Registers | DBGWVR<n>_EL1 | DBGWVR<n> |
| DCCINT | Debug Comms Channel Interrupt Enable Register | MDCCINT_EL1 | DBGDCCINT |
| DCCSR | Debug Comms Channel Status Register | MDCCSR_EL0 | DBGDSCRint |
| DBGAUTHSTATUS | Debug Authentication Status | DBGAUTHSTATUS_EL1 | DBGAUTHSTATUS |
| DLR | Debug Link Register | DLR_EL0[31:0] | DLR |
| DSCR | Debug System Control Register | MDSCR_EL1 | DBGDSCRext |
| DSPSR | Debug Saved PE State Register | DSPSR_EL0 | DSPSR |
| FAR | Fault Address Register | FAR_EL1<br>FAR_EL2<br>FAR_EL3<br>HPFAR_EL2 | DFAR, IFAR<br>HDFAR, HIFAR<br>FAR_EL3<br>HPFAR |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-1 Disambiguation of general names of registers by Execution state (continued)**

| General name | Short description | AArch64 register | AArch32 register |
|---|---|---|---|
| HCR | Hypervisor Configuration Register | HCR_EL2 | HCR<br>HCR2 |
| HDCR | Hyp or EL2 Debug Control Register | MDCR_EL2 | HDCR |
| HSCTLR | Hypervisor System Control Register | SCTLR_EL2 | HSCTLR |
| HTTBR | EL2 Translation Table Base Register | TTBR0_EL2 | HTTBR |
| ISR | Interrupt Status Register | ISR_EL1 | ISR |
| OSDLR | OS Double-Lock Register | OSDLR_EL1 | DBGOSDLR |
| OSDTRRX | OS Lock Data Transfer Register, Receive | OSDTRRX_EL1 | DBGDTRRXext |
| OSDTRTX | OS Lock Data Transfer Register, Transmit | OSDTRTX_EL1 | DBGDTRTXext |
| OSECCR | OS Lock Exception Catch Control Register | OSECCR_EL1 | DBGOSECCR |
| OSLAR | OS Lock Access Register | OSLAR_EL1 | DBGOSLAR |
| OSLSR | OS Lock Status Register | OSLSR_EL1 | DBGOSLSR |
| SCR | Secure Configuration Register | SCR_EL3 | SCR |
| SCTLR | System Control Register | SCTLR_EL1<br>SCTLR_EL2<br>SCTLR_EL3 | SCTLR (NS)<br>HSCTLR<br>SCTLR (S) |
| SDCR | Secure or EL3 Debug Configuration Register | MDCR_EL3 | SDCR |
| SDER | Secure Debug Enable Register | SDER32_EL3 | SDER |
| SPSR | Saved Program Status Register | SPSR_EL1<br>SPSR_EL2<br>SPSR_EL3 | SPSR (general description)<br>SPSR_abt<br>SPSR_fiq<br>SPSR_hyp<br>SPSR_irq<br>SPSR_mon<br>SPSR_svc<br>SPSR_und |
| TCR | Translation Control Register | TCR_EL1<br>TCR_EL2<br>TCR_EL3<br>VTCR_EL2 | TTBCR(NS)<br>HTCR<br>TTBCR(S)<br>VTCR |
| TTBR | Translation Table Base Register | TTBR0_EL1<br>TTBR0_EL2<br>TTBR0_EL3<br>TTBR1_EL1<br>VTTBR_EL2 | TTBR0<br>TTBR1<br>HTTBR<br>VTTBR |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-1 Disambiguation of general names of registers by Execution state (continued)**

| General name | Short description | AArch64 register | AArch32 register |
|---|---|---|---|
| VCR | PL1&0 stage 2 Translation Control Register | VTCR_EL2 | VTCR |
| VBAR | Vector Base Address Register | VBAR_EL1<br>VBAR_EL2<br>VBAR_EL3 | VBAR<br>HVBAR<br>MVBAR |
| VTTBR | PL1&0 stage 2 Translation Table Base Register | VTTBR_EL2 | VTTBR |

Table K13-2 disambiguates the general names of the System registers that provide access to the Performance Monitors by Execution state.

**Table K13-2 Disambiguation of general names of the Performance Monitors System registers by Execution state**

| General name | Short description | AArch64 register | AArch32 register |
|---|---|---|---|
| PMCCFILTR | Cycle Count Filter Register | PMCCFILTR_EL0 | PMCCFILTR |
| PMCCNTR | Cycle Count Register | PMCCNTR_EL0 | PMCCNTR |
| PMCEID0 | Performance Monitors Cycle Count Filter Register 0 | PMCEID0_EL0 | PMCEID0 |
| PMCEID1 | Performance Monitors Cycle Count Filter Register 1 | PMCEID1_EL0 | PMCEID1 |
| PMCNTENCLR | Performance Monitors Count Enable Clear register | PMCNTENCLR_EL0 | PMINTENCLR |
| PMCNTENSET | Performance Monitors Count Enable Set register | PMCNTENSET_EL0 | PMCNTENSET |
| PMCR | Performance Monitors Control Register | PMCR_EL0 | PMCR |
| PMEVCNTR<n> | Performance Monitors Event Count Registers, n = 0-30 | PMEVCNTR<n>_EL0 | PMEVCNTR<n> |
| PMEVTYPER<n> | Performance Monitors Event Type Registers, n = 0-30 | PMEVTYPER<n>_EL0 | PMEVTYPER<n> |
| PMINTENCLR | Performance Monitors Interrupt Enable Clear register | PMINTENCLR_EL1 | PMINTENCLR |
| PMINTENSET | Performance Monitors Interrupt Enable Set register | PMINTENSET_EL1 | PMINTENSET |
| PMOVSCLR | Performance Monitors Overflow Flag Status Register | PMOVSCLR_EL0 | PMOVSR |
| PMOVSSET | Performance Monitors Overflow Flag Status Set register | PMOVSSET_EL0 | PMOVSSET |
| PMSELR | Performance Monitors Event Counter Selection Register | PMSELR_EL0 | PMSELR |
| PMSWINC | Performance Monitors Software Increment register | PMSWINC_EL0 | PMSWINC |
| PMUSERENR | Performance Monitors User Enable Register | PMUSERENR_EL0 | PMUSERENR |
| PMXEVCNTR | Performance Monitors Selected Event Count Register | PMXEVCNTR_EL0 | PMXEVCNTR |
| PMXEVTYPER | Performance Monitors Selected Event Type Register | PMXEVTYPER_EL0 | PMXEVTYPER |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

Table K13-3 disambiguates the general names of the System registers that provide access to the Performance Monitors by Execution state.

**Table K13-3 Disambiguation of general names of the Generic Timer System registers by Execution state**

| General name | Short description | AArch64 register | AArch32 register |
|---|---|---|---|
| CNTFRQ | Counter-timer Frequency register | CNTFRQ_EL0 | CNTFRQ |
| CNTHCTL | Counter-timer Hypervisor Control register | CNTHCTL_EL2 | CNTHCTL |
| CNTHP_CTL | Counter-timer Hypervisor Physical Timer Control register | CNTHP_CTL_EL2 | CNTHP_CTL |
| CNTHP_CVAL | Counter-timer Hypervisor Physical Timer CompareValue register | CNTHP_CVAL_EL2 | CNTHP_CVAL |
| CNTHP_TVAL | Counter-timer Hypervisor Physical Timer TimerValue register | CNTHP_TVAL_EL2 | CNTHP_TVAL |
| CNTKCTL | Counter-timer Kernel Control register | CNTKCTL_EL1 | CNTKCTL |
| CNTP_CTL | Counter-timer Physical Timer Control register | CNTP_CTL_EL0 | CNTP_CTL |
| CNTP_CVAL | Counter-timer Physical Timer CompareValue register | CNTP_CVAL_EL0 | CNTP_CVAL |
| CNTP_TVAL | Counter-timer Physical Timer TimerValue register | CNTP_TVAL_EL0 | CNTP_TVAL |
| CNTPCT | Counter-timer Physical Count register | CNTPCT_EL0 | CNTPCT |
| CNTPS_CTL | Counter-timer Physical Secure Timer Control register | CNTPS_CTL_EL1 | - |
| CNTPS_CVAL | Counter-timer Physical Secure Timer CompareValue register | CNTPS_CVAL_EL1 | - |
| CNTPS_TVAL | Counter-timer Physical Secure Timer TimerValue register | CNTPS_TVAL_EL1 | - |
| CNTV_CTL | Counter-timer Virtual Timer Control register | CNTV_CTL_EL0 | CNTV_CTL |
| CNTV_CVAL | Counter-timer Virtual Timer CompareValue register | CNTV_CVAL_EL0 | CNTV_CVAL |
| CNTV_TVAL | Counter-timer Virtual Timer TimerValue register | CNTV_TVAL_EL0 | CNTV_TVAL |
| CNTVCT | Counter-timer Virtual Count register | CNTVCT_EL0 | CNTVCT |
| CNTVOFF | Counter-timer Virtual Offset register | CNTVOFF_EL2 | CNTVOFF |

### K13.1.2 Register name disambiguation by Exception level

Table K13-4 disambiguates the general names of the AArch64 System registers by Exception level.

**Table K13-4 Disambiguation of AArch64 System registers by Exception level**

| General form | EL0 | EL1 | EL2 | EL3 |
|---|---|---|---|---|
| AFSR0_ELx | - | AFSR0_EL1 | AFSR0_EL2 | AFSR0_EL3 |
| AFSR1_ELx | - | AFSR1_EL1 | AFSR1_EL2 | AFSR1_EL3 |
| CPTR_ELx | - | - | CPTR_EL2 | CPTR_EL3 |
| ELR_ELx | - | ELR_EL1 | ELR_EL2 | ELR_EL3 |
| ESR_ELx | - | ESR_EL1 | ESR_EL2 | ESR_EL3 |
| FAR_ELx | - | FAR_EL1 | FAR_EL2 | FAR_EL3 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-4 Disambiguation of AArch64 System registers by Exception level (continued)**

| General form | EL0 | EL1 | EL2 | EL3 |
|---|---|---|---|---|
| MAIR_ELx | - | MAIR_EL1 | MAIR_EL2 | MAIR_EL3 |
| RMR_ELx | - | RMR_EL1 | RMR_EL2 | RMR_EL3 |
| RVBAR_ELx | - | RVBAR_EL1 | RVBAR_EL2 | RVBAR_EL3 |
| SCTLR_ELx | - | SCTLR_EL1 | SCTLR_EL2 | SCTLR_EL3 |
| SP_ELx | SP_EL0 | SP_EL1 | SP_EL2 | SP_EL3 |
| SPSR_ELx | - | SPSR_EL1 | SPSR_EL2 | SPSR_EL3 |
| TCR_ELx | - | TCR_EL1 | TCR_EL2 | TCR_EL3 |
| VBAR_ELx | - | VBAR_EL1 | VBAR_EL2 | VBAR_EL3 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K13.2 Alphabetical index of AArch64 registers and system instructions

This section is an index of AArch64 registers and system instructions in alphabetical order.

**Table K13-5 Alphabetical index of AArch64 Registers**

| Register | Description, see |
|---|---|
| ACTLR_EL1 | *ACTLR_EL1, Auxiliary Control Register (EL1)* on page D7-1914 |
| ACTLR_EL2 | *ACTLR_EL2, Auxiliary Control Register (EL2)* on page D7-1915 |
| ACTLR_EL3 | *ACTLR_EL3, Auxiliary Control Register (EL3)* on page D7-1916 |
| AFSR0_EL1 | *AFSR0_EL1, Auxiliary Fault Status Register 0 (EL1)* on page D7-1917 |
| AFSR0_EL2 | *AFSR0_EL2, Auxiliary Fault Status Register 0 (EL2)* on page D7-1918 |
| AFSR0_EL3 | *AFSR0_EL3, Auxiliary Fault Status Register 0 (EL3)* on page D7-1919 |
| AFSR1_EL1 | *AFSR1_EL1, Auxiliary Fault Status Register 1 (EL1)* on page D7-1920 |
| AFSR1_EL2 | *AFSR1_EL2, Auxiliary Fault Status Register 1 (EL2)* on page D7-1921 |
| AFSR1_EL3 | *AFSR1_EL3, Auxiliary Fault Status Register 1 (EL3)* on page D7-1922 |
| AIDR_EL1 | *AIDR_EL1, Auxiliary ID Register* on page D7-1923 |
| AMAIR_EL1 | *AMAIR_EL1, Auxiliary Memory Attribute Indirection Register (EL1)* on page D7-1924 |
| AMAIR_EL2 | *AMAIR_EL2, Auxiliary Memory Attribute Indirection Register (EL2)* on page D7-1926 |
| AMAIR_EL3 | *AMAIR_EL3, Auxiliary Memory Attribute Indirection Register (EL3)* on page D7-1927 |
| AT S12E0R | *AT S12E0R, Address Translate Stages 1 and 2 EL0 Read* on page C5-359 |
| AT S12E0W | *AT S12E0W, Address Translate Stages 1 and 2 EL0 Write* on page C5-360 |
| AT S12E1R | *AT S12E1R, Address Translate Stages 1 and 2 EL1 Read* on page C5-361 |
| AT S12E1W | *AT S12E1W, Address Translate Stages 1 and 2 EL1 Write* on page C5-362 |
| AT S1E0R | *AT S1E0R, Address Translate Stage 1 EL0 Read* on page C5-363 |
| AT S1E0W | *AT S1E0W, Address Translate Stage 1 EL0 Write* on page C5-364 |
| AT S1E1R | *AT S1E1R, Address Translate Stage 1 EL1 Read* on page C5-365 |
| AT S1E1W | *AT S1E1W, Address Translate Stage 1 EL1 Write* on page C5-366 |
| AT S1E2R | *AT S1E2R, Address Translate Stage 1 EL2 Read* on page C5-367 |
| AT S1E2W | *AT S1E2W, Address Translate Stage 1 EL2 Write* on page C5-368 |
| AT S1E3R | *AT S1E3R, Address Translate Stage 1 EL3 Read* on page C5-369 |
| AT S1E3W | *AT S1E3W, Address Translate Stage 1 EL3 Write* on page C5-370 |
| CCSIDR_EL1 | *CCSIDR_EL1, Current Cache Size ID Register* on page D7-1928 |
| CLIDR_EL1 | *CLIDR_EL1, Cache Level ID Register* on page D7-1930 |
| CNTFRQ_EL0 | *CNTFRQ_EL0, Counter-timer Frequency register* on page D7-2277 |
| CNTHCTL_EL2 | *CNTHCTL_EL2, Counter-timer Hypervisor Control register* on page D7-2279 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-5 Alphabetical index of AArch64 Registers (continued)**

| Register | Description, see |
|---|---|
| CNTHP_CTL_EL2 | *CNTHP_CTL_EL2, Counter-timer Hypervisor Physical Timer Control register* on page D7-2281 |
| CNTHP_CVAL_EL2 | *CNTHP_CVAL_EL2, Counter-timer Hypervisor Physical Timer CompareValue register* on page D7-2283 |
| CNTHP_TVAL_EL2 | *CNTHP_TVAL_EL2, Counter-timer Hypervisor Physical Timer TimerValue register* on page D7-2284 |
| CNTKCTL_EL1 | *CNTKCTL_EL1, Counter-timer Kernel Control register* on page D7-2285 |
| CNTP_CTL_EL0 | *CNTP_CTL_EL0, Counter-timer Physical Timer Control register* on page D7-2288 |
| CNTP_CVAL_EL0 | *CNTP_CVAL_EL0, Counter-timer Physical Timer CompareValue register* on page D7-2290 |
| CNTP_TVAL_EL0 | *CNTP_TVAL_EL0, Counter-timer Physical Timer TimerValue register* on page D7-2291 |
| CNTPCT_EL0 | *CNTPCT_EL0, Counter-timer Physical Count register* on page D7-2293 |
| CNTPS_CTL_EL1 | *CNTPS_CTL_EL1, Counter-timer Physical Secure Timer Control register* on page D7-2294 |
| CNTPS_CVAL_EL1 | *CNTPS_CVAL_EL1, Counter-timer Physical Secure Timer CompareValue register* on page D7-2296 |
| CNTPS_TVAL_EL1 | *CNTPS_TVAL_EL1, Counter-timer Physical Secure Timer TimerValue register* on page D7-2297 |
| CNTV_CTL_EL0 | *CNTV_CTL_EL0, Counter-timer Virtual Timer Control register* on page D7-2299 |
| CNTV_CVAL_EL0 | *CNTV_CVAL_EL0, Counter-timer Virtual Timer CompareValue register* on page D7-2301 |
| CNTV_TVAL_EL0 | *CNTV_TVAL_EL0, Counter-timer Virtual Timer TimerValue register* on page D7-2302 |
| CNTVCT_EL0 | *CNTVCT_EL0, Counter-timer Virtual Count register* on page D7-2304 |
| CNTVOFF_EL2 | *CNTVOFF_EL2, Counter-timer Virtual Offset register* on page D7-2305 |
| CONTEXTIDR_EL1 | *CONTEXTIDR_EL1, Context ID Register* on page D7-1932 |
| CPACR_EL1 | *CPACR_EL1, Architectural Feature Access Control Register* on page D7-1934 |
| CPTR_EL2 | *CPTR_EL2, Architectural Feature Trap Register (EL2)* on page D7-1936 |
| CPTR_EL3 | *CPTR_EL3, Architectural Feature Trap Register (EL3)* on page D7-1938 |
| CSSELR_EL1 | *CSSELR_EL1, Cache Size Selection Register* on page D7-1940 |
| CTR_EL0 | *CTR_EL0, Cache Type Register* on page D7-1942 |
| CurrentEL | *CurrentEL, Current Exception Level* on page C5-290 |
| DACR32_EL2 | *DACR32_EL2, Domain Access Control Register* on page D7-1944 |
| DAIF | *DAIF, Interrupt Mask Bits* on page C5-292 |
| DBGAUTHSTATUS_EL1 | *DBGAUTHSTATUS_EL1, Debug Authentication Status register* on page D7-2166 |
| DBGBCR<n>_EL1 | *DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15* on page D7-2168 |
| DBGBVR<n>_EL1 | *DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15* on page D7-2172 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-5 Alphabetical index of AArch64 Registers (continued)**

| Register | Description, see |
|---|---|
| DBGCLAIMCLR_EL1 | *DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register* on page D7-2175 |
| DBGCLAIMSET_EL1 | *DBGCLAIMSET_EL1, Debug CLAIM Tag Set register* on page D7-2177 |
| DBGDTR_EL0 | *DBGDTR_EL0, Debug Data Transfer Register, half-duplex* on page D7-2179 |
| DBGDTRRX_EL0 | *DBGDTRRX_EL0, Debug Data Transfer Register, Receive* on page D7-2181 |
| DBGDTRTX_EL0 | *DBGDTRTX_EL0, Debug Data Transfer Register, Transmit* on page D7-2183 |
| DBGPRCR_EL1 | *DBGPRCR_EL1, Debug Power Control Register* on page D7-2185 |
| DBGVCR32_EL2 | *DBGVCR32_EL2, Debug Vector Catch Register* on page D7-2187 |
| DBGWCR<n>_EL1 | *DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15* on page D7-2192 |
| DBGWVR<n>_EL1 | *DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15* on page D7-2196 |
| DC CISW | *DC CISW, Data or unified Cache line Clean and Invalidate by Set/Way* on page C5-343 |
| DC CIVAC | *DC CIVAC, Data or unified Cache line Clean and Invalidate by VA to PoC* on page C5-345 |
| DC CSW | *DC CSW, Data or unified Cache line Clean by Set/Way* on page C5-346 |
| DC CVAC | *DC CVAC, Data or unified Cache line Clean by VA to PoC* on page C5-348 |
| DC CVAU | *DC CVAU, Data or unified Cache line Clean by VA to PoU* on page C5-349 |
| DC ISW | *DC ISW, Data or unified Cache line Invalidate by Set/Way* on page C5-350 |
| DC IVAC | *DC IVAC, Data or unified Cache line Invalidate by VA to PoC* on page C5-352 |
| DC ZVA | *DC ZVA, Data Cache Zero by VA* on page C5-353 |
| DCZID_EL0 | *DCZID_EL0, Data Cache Zero ID register* on page D7-1946 |
| DLR_EL0 | *DLR_EL0, Debug Link Register* on page D7-2198 |
| DSPSR_EL0 | *DSPSR_EL0, Debug Saved Program Status Register* on page D7-2199 |
| ELR_EL1 | *ELR_EL1, Exception Link Register (EL1)* on page C5-296 |
| ELR_EL2 | *ELR_EL2, Exception Link Register (EL2)* on page C5-297 |
| ELR_EL3 | *ELR_EL3, Exception Link Register (EL3)* on page C5-298 |
| ESR_EL1 | *ESR_EL1, Exception Syndrome Register (EL1)* on page D7-1948 |
| ESR_EL2 | *ESR_EL2, Exception Syndrome Register (EL2)* on page D7-1949 |
| ESR_EL3 | *ESR_EL3, Exception Syndrome Register (EL3)* on page D7-1950 |
| ESR_ELx | *ESR_ELx, Exception Syndrome Register (ELx)* on page D7-1951 |
| FAR_EL1 | *FAR_EL1, Fault Address Register (EL1)* on page D7-1983 |
| FAR_EL2 | *FAR_EL2, Fault Address Register (EL2)* on page D7-1985 |
| FAR_EL3 | *FAR_EL3, Fault Address Register (EL3)* on page D7-1987 |
| FPCR | *FPCR, Floating-point Control Register* on page C5-299 |
| FPEXC32_EL2 | *FPEXC32_EL2, Floating-Point Exception Control register* on page D7-1989 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-5 Alphabetical index of AArch64 Registers (continued)**

| Register | Description, see |
|---|---|
| FPSR | *FPSR, Floating-point Status Register* on page C5-303 |
| HACR_EL2 | *HACR_EL2, Hypervisor Auxiliary Control Register* on page D7-1994 |
| HCR_EL2 | *HCR_EL2, Hypervisor Configuration Register* on page D7-1995 |
| HPFAR_EL2 | *HPFAR_EL2, Hypervisor IPA Fault Address Register* on page D7-2007 |
| HSTR_EL2 | *HSTR_EL2, Hypervisor System Trap Register* on page D7-2009 |
| IC IALLU | *IC IALLU, Instruction Cache Invalidate All to PoU* on page C5-355 |
| IC IALLUIS | *IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable* on page C5-356 |
| IC IVAU | *IC IVAU, Instruction Cache line Invalidate by VA to PoU* on page C5-357 |
| ID_AA64AFR0_EL1 | *ID_AA64AFR0_EL1, AArch64 Auxiliary Feature Register 0* on page D7-2011 |
| ID_AA64AFR1_EL1 | *ID_AA64AFR1_EL1, AArch64 Auxiliary Feature Register 1* on page D7-2013 |
| ID_AA64DFR0_EL1 | *ID_AA64DFR0_EL1, AArch64 Debug Feature Register 0* on page D7-2014 |
| ID_AA64DFR1_EL1 | *ID_AA64DFR1_EL1, AArch64 Debug Feature Register 1* on page D7-2016 |
| ID_AA64ISAR0_EL1 | *ID_AA64ISAR0_EL1, AArch64 Instruction Set Attribute Register 0* on page D7-2017 |
| ID_AA64ISAR1_EL1 | *ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1* on page D7-2019 |
| ID_AA64MMFR0_EL1 | *ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0* on page D7-2020 |
| ID_AA64MMFR1_EL1 | *ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1* on page D7-2023 |
| ID_AA64PFR0_EL1 | *ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0* on page D7-2024 |
| ID_AA64PFR1_EL1 | *ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1* on page D7-2026 |
| ID_AFR0_EL1 | *ID_AFR0_EL1, AArch32 Auxiliary Feature Register 0* on page D7-2027 |
| ID_DFR0_EL1 | *ID_DFR0_EL1, AArch32 Debug Feature Register 0* on page D7-2029 |
| ID_ISAR0_EL1 | *ID_ISAR0_EL1, AArch32 Instruction Set Attribute Register 0* on page D7-2032 |
| ID_ISAR1_EL1 | *ID_ISAR1_EL1, AArch32 Instruction Set Attribute Register 1* on page D7-2035 |
| ID_ISAR2_EL1 | *ID_ISAR2_EL1, AArch32 Instruction Set Attribute Register 2* on page D7-2038 |
| ID_ISAR3_EL1 | *ID_ISAR3_EL1, AArch32 Instruction Set Attribute Register 3* on page D7-2041 |
| ID_ISAR4_EL1 | *ID_ISAR4_EL1, AArch32 Instruction Set Attribute Register 4* on page D7-2044 |
| ID_ISAR5_EL1 | *ID_ISAR5_EL1, AArch32 Instruction Set Attribute Register 5* on page D7-2047 |
| ID_MMFR0_EL1 | *ID_MMFR0_EL1, AArch32 Memory Model Feature Register 0* on page D7-2049 |
| ID_MMFR1_EL1 | *ID_MMFR1_EL1, AArch32 Memory Model Feature Register 1* on page D7-2052 |
| ID_MMFR2_EL1 | *ID_MMFR2_EL1, AArch32 Memory Model Feature Register 2* on page D7-2056 |
| ID_MMFR3_EL1 | *ID_MMFR3_EL1, AArch32 Memory Model Feature Register 3* on page D7-2059 |
| ID_MMFR4_EL1 | *ID_MMFR4_EL1, AArch32 Memory Model Feature Register 4* on page D7-2062 |
| ID_PFR0_EL1 | *ID_PFR0_EL1, AArch32 Processor Feature Register 0* on page D7-2064 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-5 Alphabetical index of AArch64 Registers (continued)**

| Register | Description, see |
|---|---|
| ID_PFR1_EL1 | *ID_PFR1_EL1, AArch32 Processor Feature Register 1* on page D7-2066 |
| IFSR32_EL2 | *IFSR32_EL2, Instruction Fault Status Register (EL2)* on page D7-2069 |
| ISR_EL1 | *ISR_EL1, Interrupt Status Register* on page D7-2073 |
| MAIR_EL1 | *MAIR_EL1, Memory Attribute Indirection Register (EL1)* on page D7-2075 |
| MAIR_EL2 | *MAIR_EL2, Memory Attribute Indirection Register (EL2)* on page D7-2078 |
| MAIR_EL3 | *MAIR_EL3, Memory Attribute Indirection Register (EL3)* on page D7-2080 |
| MDCCINT_EL1 | *MDCCINT_EL1, Monitor DCC Interrupt Enable Register* on page D7-2204 |
| MDCCSR_EL0 | *MDCCSR_EL0, Monitor DCC Status Register* on page D7-2206 |
| MDCR_EL2 | *MDCR_EL2, Monitor Debug Configuration Register (EL2)* on page D7-2208 |
| MDCR_EL3 | *MDCR_EL3, Monitor Debug Configuration Register (EL3)* on page D7-2212 |
| MDRAR_EL1 | *MDRAR_EL1, Monitor Debug ROM Address Register* on page D7-2216 |
| MDSCR_EL1 | *MDSCR_EL1, Monitor Debug System Control Register* on page D7-2218 |
| MIDR_EL1 | *MIDR_EL1, Main ID Register* on page D7-2082 |
| MPIDR_EL1 | *MPIDR_EL1, Multiprocessor Affinity Register* on page D7-2084 |
| MVFR0_EL1 | *MVFR0_EL1, AArch32 Media and VFP Feature Register 0* on page D7-2086 |
| MVFR1_EL1 | *MVFR1_EL1, AArch32 Media and VFP Feature Register 1* on page D7-2089 |
| MVFR2_EL1 | *MVFR2_EL1, AArch32 Media and VFP Feature Register 2* on page D7-2092 |
| NZCV | *NZCV, Condition Flags* on page C5-306 |
| OSDLR_EL1 | *OSDLR_EL1, OS Double Lock Register* on page D7-2222 |
| OSDTRRX_EL1 | *OSDTRRX_EL1, OS Lock Data Transfer Register, Receive* on page D7-2224 |
| OSDTRTX_EL1 | *OSDTRTX_EL1, OS Lock Data Transfer Register, Transmit* on page D7-2226 |
| OSECCR_EL1 | *OSECCR_EL1, OS Lock Exception Catch Control Register* on page D7-2228 |
| OSLAR_EL1 | *OSLAR_EL1, OS Lock Access Register* on page D7-2230 |
| OSLSR_EL1 | *OSLSR_EL1, OS Lock Status Register* on page D7-2232 |
| PAR_EL1 | *PAR_EL1, Physical Address Register* on page D7-2094 |
| PMCCFILTR_EL0 | *PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register* on page D7-2237 |
| PMCCNTR_EL0 | *PMCCNTR_EL0, Performance Monitors Cycle Count Register* on page D7-2239 |
| PMCEID0_EL0 | *PMCEID0_EL0, Performance Monitors Common Event Identification register 0* on page D7-2241 |
| PMCEID1_EL0 | *PMCEID1_EL0, Performance Monitors Common Event Identification register 1* on page D7-2243 |
| PMCNTENCLR_EL0 | *PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register* on page D7-2245 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-5 Alphabetical index of AArch64 Registers (continued)**

| Register | Description, see |
|---|---|
| PMCNTENSET_EL0 | *PMCNTENSET_EL0, Performance Monitors Count Enable Set register* on page D7-2247 |
| PMCR_EL0 | *PMCR_EL0, Performance Monitors Control Register* on page D7-2249 |
| PMEVCNTR<n>_EL0 | *PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30* on page D7-2253 |
| PMEVTYPER<n>_EL0 | *PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30* on page D7-2255 |
| PMINTENCLR_EL1 | *PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register* on page D7-2258 |
| PMINTENSET_EL1 | *PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register* on page D7-2260 |
| PMOVSCLR_EL0 | *PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register* on page D7-2262 |
| PMOVSSET_EL0 | *PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register* on page D7-2264 |
| PMSELR_EL0 | *PMSELR_EL0, Performance Monitors Event Counter Selection Register* on page D7-2266 |
| PMSWINC_EL0 | *PMSWINC_EL0, Performance Monitors Software Increment register* on page D7-2268 |
| PMUSERENR_EL0 | *PMUSERENR_EL0, Performance Monitors User Enable Register* on page D7-2270 |
| PMXEVCNTR_EL0 | *PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register* on page D7-2272 |
| PMXEVTYPER_EL0 | *PMXEVTYPER_EL0, Performance Monitors Selected Event Type Register* on page D7-2274 |
| REVIDR_EL1 | *REVIDR_EL1, Revision ID Register* on page D7-2097 |
| RMR_EL1 | *RMR_EL1, Reset Management Register (if EL2 and EL3 not implemented)* on page D7-2098 |
| RMR_EL2 | *RMR_EL2, Reset Management Register (if EL3 not implemented)* on page D7-2100 |
| RMR_EL3 | *RMR_EL3, Reset Management Register (if EL3 implemented)* on page D7-2102 |
| RVBAR_EL1 | *RVBAR_EL1, Reset Vector Base Address Register (if EL2 and EL3 not implemented)* on page D7-2104 |
| RVBAR_EL2 | *RVBAR_EL2, Reset Vector Base Address Register (if EL3 not implemented)* on page D7-2105 |
| RVBAR_EL3 | *RVBAR_EL3, Reset Vector Base Address Register (if EL3 implemented)* on page D7-2106 |
| S3_<op1>_<Cn>_<Cm>_<op2> | *S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED registers* on page D7-2107 |
| SCR_EL3 | *SCR_EL3, Secure Configuration Register* on page D7-2108 |
| SCTLR_EL1 | *SCTLR_EL1, System Control Register (EL1)* on page D7-2112 |
| SCTLR_EL2 | *SCTLR_EL2, System Control Register (EL2)* on page D7-2119 |
| SCTLR_EL3 | *SCTLR_EL3, System Control Register (EL3)* on page D7-2123 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-5 Alphabetical index of AArch64 Registers (continued)**

| Register | Description, see |
|---|---|
| SDER32_EL3 | *SDER32_EL3, AArch32 Secure Debug Enable Register* on page D7-2234 |
| SP_EL0 | *SP_EL0, Stack Pointer (EL0)* on page C5-308 |
| SP_EL1 | *SP_EL1, Stack Pointer (EL1)* on page C5-309 |
| SP_EL2 | *SP_EL2, Stack Pointer (EL2)* on page C5-311 |
| SP_EL3 | *SP_EL3, Stack Pointer (EL3)* on page C5-313 |
| SPSel | *SPSel, Stack Pointer Select* on page C5-314 |
| SPSR_abt | *SPSR_abt, Saved Program Status Register (Abort mode)* on page C5-315 |
| SPSR_EL1 | *SPSR_EL1, Saved Program Status Register (EL1)* on page C5-318 |
| SPSR_EL2 | *SPSR_EL2, Saved Program Status Register (EL2)* on page C5-323 |
| SPSR_EL3 | *SPSR_EL3, Saved Program Status Register (EL3)* on page C5-328 |
| SPSR_fiq | *SPSR_fiq, Saved Program Status Register (FIQ mode)* on page C5-333 |
| SPSR_irq | *SPSR_irq, Saved Program Status Register (IRQ mode)* on page C5-336 |
| SPSR_und | *SPSR_und, Saved Program Status Register (Undefined mode)* on page C5-339 |
| TCR_EL1 | *TCR_EL1, Translation Control Register (EL1)* on page D7-2127 |
| TCR_EL2 | *TCR_EL2, Translation Control Register (EL2)* on page D7-2132 |
| TCR_EL3 | *TCR_EL3, Translation Control Register (EL3)* on page D7-2135 |
| TLBI ALLE1 | *TLBI ALLE1, TLB Invalidate All, EL1* on page C5-372 |
| TLBI ALLE1IS | *TLBI ALLE1IS, TLB Invalidate All, EL1, Inner Shareable* on page C5-373 |
| TLBI ALLE2 | *TLBI ALLE2, TLB Invalidate All, EL2* on page C5-374 |
| TLBI ALLE2IS | *TLBI ALLE2IS, TLB Invalidate All, EL2, Inner Shareable* on page C5-375 |
| TLBI ALLE3 | *TLBI ALLE3, TLB Invalidate All, EL3* on page C5-376 |
| TLBI ALLE3IS | *TLBI ALLE3IS, TLB Invalidate All, EL3, Inner Shareable* on page C5-377 |
| TLBI ASIDE1 | *TLBI ASIDE1, TLB Invalidate by ASID, EL1* on page C5-378 |
| TLBI ASIDE1IS | *TLBI ASIDE1IS, TLB Invalidate by ASID, EL1, Inner Shareable* on page C5-380 |
| TLBI IPAS2E1 | *TLBI IPAS2E1, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1* on page C5-382 |
| TLBI IPAS2E1IS | *TLBI IPAS2E1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable* on page C5-383 |
| TLBI IPAS2LE1 | *TLBI IPAS2LE1, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1* on page C5-385 |
| TLBI IPAS2LE1IS | *TLBI IPAS2LE1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable* on page C5-386 |
| TLBI VAAE1 | *TLBI VAAE1, TLB Invalidate by VA, All ASID, EL1* on page C5-388 |
| TLBI VAAE1IS | *TLBI VAAE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable* on page C5-390 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-5 Alphabetical index of AArch64 Registers (continued)**

| Register | Description, see |
|---|---|
| TLBI VAALE1 | *TLBI VAALE1, TLB Invalidate by VA, All ASID, Last level, EL1* on page C5-392 |
| TLBI VAALE1IS | *TLBI VAALE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable* on page C5-394 |
| TLBI VAE1 | *TLBI VAE1, TLB Invalidate by VA, EL1* on page C5-396 |
| TLBI VAE1IS | *TLBI VAE1IS, TLB Invalidate by VA, EL1, Inner Shareable* on page C5-398 |
| TLBI VAE2 | *TLBI VAE2, TLB Invalidate by VA, EL2* on page C5-400 |
| TLBI VAE2IS | *TLBI VAE2IS, TLB Invalidate by VA, EL2, Inner Shareable* on page C5-402 |
| TLBI VAE3 | *TLBI VAE3, TLB Invalidate by VA, EL3* on page C5-404 |
| TLBI VAE3IS | *TLBI VAE3IS, TLB Invalidate by VA, EL3, Inner Shareable* on page C5-406 |
| TLBI VALE1 | *TLBI VALE1, TLB Invalidate by VA, Last level, EL1* on page C5-408 |
| TLBI VALE1IS | *TLBI VALE1IS, TLB Invalidate by VA, Last level, EL1, Inner Shareable* on page C5-410 |
| TLBI VALE2 | *TLBI VALE2, TLB Invalidate by VA, Last level, EL2* on page C5-412 |
| TLBI VALE2IS | *TLBI VALE2IS, TLB Invalidate by VA, Last level, EL2, Inner Shareable* on page C5-414 |
| TLBI VALE3 | *TLBI VALE3, TLB Invalidate by VA, Last level, EL3* on page C5-416 |
| TLBI VALE3IS | *TLBI VALE3IS, TLB Invalidate by VA, Last level, EL3, Inner Shareable* on page C5-418 |
| TLBI VMALLE1 | *TLBI VMALLE1, TLB Invalidate by VMID, All at stage 1, EL1* on page C5-420 |
| TLBI VMALLE1IS | *TLBI VMALLE1IS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable* on page C5-421 |
| TLBI VMALLS12E1 | *TLBI VMALLS12E1, TLB Invalidate by VMID, All at Stage 1 and 2, EL1* on page C5-422 |
| TLBI VMALLS12E1IS | *TLBI VMALLS12E1IS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable* on page C5-423 |
| TPIDR_EL0 | *TPIDR_EL0, EL0 Read/Write Software Thread ID Register* on page D7-2138 |
| TPIDR_EL1 | *TPIDR_EL1, EL1 Software Thread ID Register* on page D7-2139 |
| TPIDR_EL2 | *TPIDR_EL2, EL2 Software Thread ID Register* on page D7-2140 |
| TPIDR_EL3 | *TPIDR_EL3, EL3 Software Thread ID Register* on page D7-2141 |
| TPIDRRO_EL0 | *TPIDRRO_EL0, EL0 Read-Only Software Thread ID Register* on page D7-2142 |
| TTBR0_EL1 | *TTBR0_EL1, Translation Table Base Register 0 (EL1)* on page D7-2143 |
| TTBR0_EL2 | *TTBR0_EL2, Translation Table Base Register 0 (EL2)* on page D7-2145 |
| TTBR0_EL3 | *TTBR0_EL3, Translation Table Base Register 0 (EL3)* on page D7-2147 |
| TTBR1_EL1 | *TTBR1_EL1, Translation Table Base Register 1* on page D7-2149 |
| VBAR_EL1 | *VBAR_EL1, Vector Base Address Register (EL1)* on page D7-2151 |
| VBAR_EL2 | *VBAR_EL2, Vector Base Address Register (EL2)* on page D7-2153 |
| VBAR_EL3 | *VBAR_EL3, Vector Base Address Register (EL3)* on page D7-2155 |
| VMPIDR_EL2 | *VMPIDR_EL2, Virtualization Multiprocessor ID Register* on page D7-2156 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-5 Alphabetical index of AArch64 Registers (continued)**

| Register | Description, see |
|---|---|
| VPIDR_EL2 | *VPIDR_EL2, Virtualization Processor ID Register* on page D7-2158 |
| VTCR_EL2 | *VTCR_EL2, Virtualization Translation Control Register* on page D7-2160 |
| VTTBR_EL2 | *VTTBR_EL2, Virtualization Translation Table Base Register* on page D7-2163 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## K13.3 Functional index of AArch64 registers and system instructions

This section is an index of the AArch64 registers and system instructions, divided by functional group.

### K13.3.1 Special-purpose registers

This section is an index to the registers in the Special-purpose registers functional group.

**Table K13-6 Special-purpose registers**

| Register | Description, see |
|---|---|
| DLR_EL0 | *DLR_EL0, Debug Link Register* on page D7-2198 |
| DSPSR_EL0 | *DSPSR_EL0, Debug Saved Program Status Register* on page D7-2199 |
| ELR_EL1 | *ELR_EL1, Exception Link Register (EL1)* on page C5-296 |
| ELR_EL2 | *ELR_EL2, Exception Link Register (EL2)* on page C5-297 |
| ELR_EL3 | *ELR_EL3, Exception Link Register (EL3)* on page C5-298 |
| FPCR | *FPCR, Floating-point Control Register* on page C5-299 |
| FPSR | *FPSR, Floating-point Status Register* on page C5-303 |
| SP_EL0 | *SP_EL0, Stack Pointer (EL0)* on page C5-308 |
| SP_EL1 | *SP_EL1, Stack Pointer (EL1)* on page C5-309 |
| SP_EL2 | *SP_EL2, Stack Pointer (EL2)* on page C5-311 |
| SP_EL3 | *SP_EL3, Stack Pointer (EL3)* on page C5-313 |
| SPSR_abt | *SPSR_abt, Saved Program Status Register (Abort mode)* on page C5-315 |
| SPSR_EL1 | *SPSR_EL1, Saved Program Status Register (EL1)* on page C5-318 |
| SPSR_EL2 | *SPSR_EL2, Saved Program Status Register (EL2)* on page C5-323 |
| SPSR_EL3 | *SPSR_EL3, Saved Program Status Register (EL3)* on page C5-328 |
| SPSR_fiq | *SPSR_fiq, Saved Program Status Register (FIQ mode)* on page C5-333 |
| SPSR_irq | *SPSR_irq, Saved Program Status Register (IRQ mode)* on page C5-336 |
| SPSR_und | *SPSR_und, Saved Program Status Register (Undefined mode)* on page C5-339 |

### K13.3.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

**Table K13-7 VMSA-specific registers**

| Register | Description, see |
|---|---|
| AMAIR_EL1 | *AMAIR_EL1, Auxiliary Memory Attribute Indirection Register (EL1)* on page D7-1924 |
| AMAIR_EL2 | *AMAIR_EL2, Auxiliary Memory Attribute Indirection Register (EL2)* on page D7-1926 |
| AMAIR_EL3 | *AMAIR_EL3, Auxiliary Memory Attribute Indirection Register (EL3)* on page D7-1927 |
| CONTEXTIDR_EL1 | *CONTEXTIDR_EL1, Context ID Register* on page D7-1932 |
| DACR32_EL2 | *DACR32_EL2, Domain Access Control Register* on page D7-1944 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-7 VMSA-specific registers (continued)**

| Register | Description, see |
|---|---|
| MAIR_EL1 | *MAIR_EL1, Memory Attribute Indirection Register (EL1)* on page D7-2075 |
| MAIR_EL2 | *MAIR_EL2, Memory Attribute Indirection Register (EL2)* on page D7-2078 |
| MAIR_EL3 | *MAIR_EL3, Memory Attribute Indirection Register (EL3)* on page D7-2080 |
| TCR_EL1 | *TCR_EL1, Translation Control Register (EL1)* on page D7-2127 |
| TCR_EL2 | *TCR_EL2, Translation Control Register (EL2)* on page D7-2132 |
| TCR_EL3 | *TCR_EL3, Translation Control Register (EL3)* on page D7-2135 |
| TTBR0_EL1 | *TTBR0_EL1, Translation Table Base Register 0 (EL1)* on page D7-2143 |
| TTBR0_EL2 | *TTBR0_EL2, Translation Table Base Register 0 (EL2)* on page D7-2145 |
| TTBR0_EL3 | *TTBR0_EL3, Translation Table Base Register 0 (EL3)* on page D7-2147 |
| TTBR1_EL1 | *TTBR1_EL1, Translation Table Base Register 1* on page D7-2149 |
| VTCR_EL2 | *VTCR_EL2, Virtualization Translation Control Register* on page D7-2160 |
| VTTBR_EL2 | *VTTBR_EL2, Virtualization Translation Table Base Register* on page D7-2163 |

### K13.3.3 ID registers

This section is an index to the registers in the Identification registers functional group.

**Table K13-8 ID registers**

| Register | Description, see |
|---|---|
| AIDR_EL1 | *AIDR_EL1, Auxiliary ID Register* on page D7-1923 |
| CCSIDR_EL1 | *CCSIDR_EL1, Current Cache Size ID Register* on page D7-1928 |
| CLIDR_EL1 | *CLIDR_EL1, Cache Level ID Register* on page D7-1930 |
| CSSELR_EL1 | *CSSELR_EL1, Cache Size Selection Register* on page D7-1940 |
| CTR_EL0 | *CTR_EL0, Cache Type Register* on page D7-1942 |
| DCZID_EL0 | *DCZID_EL0, Data Cache Zero ID register* on page D7-1946 |
| ID_AA64AFR0_EL1 | *ID_AA64AFR0_EL1, AArch64 Auxiliary Feature Register 0* on page D7-2011 |
| ID_AA64AFR1_EL1 | *ID_AA64AFR1_EL1, AArch64 Auxiliary Feature Register 1* on page D7-2013 |
| ID_AA64DFR0_EL1 | *ID_AA64DFR0_EL1, AArch64 Debug Feature Register 0* on page D7-2014 |
| ID_AA64DFR1_EL1 | *ID_AA64DFR1_EL1, AArch64 Debug Feature Register 1* on page D7-2016 |
| ID_AA64ISAR0_EL1 | *ID_AA64ISAR0_EL1, AArch64 Instruction Set Attribute Register 0* on page D7-2017 |
| ID_AA64ISAR1_EL1 | *ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1* on page D7-2019 |
| ID_AA64MMFR0_EL1 | *ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0* on page D7-2020 |
| ID_AA64MMFR1_EL1 | *ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1* on page D7-2023 |
| ID_AA64PFR0_EL1 | *ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0* on page D7-2024 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-8 ID registers (continued)**

| Register | Description, see |
|---|---|
| ID_AA64PFR1_EL1 | *ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1* on page D7-2026 |
| ID_AFR0_EL1 | *ID_AFR0_EL1, AArch32 Auxiliary Feature Register 0* on page D7-2027 |
| ID_DFR0_EL1 | *ID_DFR0_EL1, AArch32 Debug Feature Register 0* on page D7-2029 |
| ID_ISAR0_EL1 | *ID_ISAR0_EL1, AArch32 Instruction Set Attribute Register 0* on page D7-2032 |
| ID_ISAR1_EL1 | *ID_ISAR1_EL1, AArch32 Instruction Set Attribute Register 1* on page D7-2035 |
| ID_ISAR2_EL1 | *ID_ISAR2_EL1, AArch32 Instruction Set Attribute Register 2* on page D7-2038 |
| ID_ISAR3_EL1 | *ID_ISAR3_EL1, AArch32 Instruction Set Attribute Register 3* on page D7-2041 |
| ID_ISAR4_EL1 | *ID_ISAR4_EL1, AArch32 Instruction Set Attribute Register 4* on page D7-2044 |
| ID_ISAR5_EL1 | *ID_ISAR5_EL1, AArch32 Instruction Set Attribute Register 5* on page D7-2047 |
| ID_MMFR0_EL1 | *ID_MMFR0_EL1, AArch32 Memory Model Feature Register 0* on page D7-2049 |
| ID_MMFR1_EL1 | *ID_MMFR1_EL1, AArch32 Memory Model Feature Register 1* on page D7-2052 |
| ID_MMFR2_EL1 | *ID_MMFR2_EL1, AArch32 Memory Model Feature Register 2* on page D7-2056 |
| ID_MMFR3_EL1 | *ID_MMFR3_EL1, AArch32 Memory Model Feature Register 3* on page D7-2059 |
| ID_MMFR4_EL1 | *ID_MMFR4_EL1, AArch32 Memory Model Feature Register 4* on page D7-2062 |
| ID_PFR0_EL1 | *ID_PFR0_EL1, AArch32 Processor Feature Register 0* on page D7-2064 |
| ID_PFR1_EL1 | *ID_PFR1_EL1, AArch32 Processor Feature Register 1* on page D7-2066 |
| MIDR_EL1 | *MIDR_EL1, Main ID Register* on page D7-2082 |
| MPIDR_EL1 | *MPIDR_EL1, Multiprocessor Affinity Register* on page D7-2084 |
| MVFR0_EL1 | *MVFR0_EL1, AArch32 Media and VFP Feature Register 0* on page D7-2086 |
| MVFR1_EL1 | *MVFR1_EL1, AArch32 Media and VFP Feature Register 1* on page D7-2089 |
| MVFR2_EL1 | *MVFR2_EL1, AArch32 Media and VFP Feature Register 2* on page D7-2092 |
| REVIDR_EL1 | *REVIDR_EL1, Revision ID Register* on page D7-2097 |
| VMPIDR_EL2 | *VMPIDR_EL2, Virtualization Multiprocessor ID Register* on page D7-2156 |
| VPIDR_EL2 | *VPIDR_EL2, Virtualization Processor ID Register* on page D7-2158 |

### K13.3.4 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

**Table K13-9 Performance monitors registers**

| Register | Description, see |
|---|---|
| PMCCFILTR_EL0 | *PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register* on page D7-2237 |
| PMCCNTR_EL0 | *PMCCNTR_EL0, Performance Monitors Cycle Count Register* on page D7-2239 |
| PMCEID0_EL0 | *PMCEID0_EL0, Performance Monitors Common Event Identification register 0* on page D7-2241 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-9 Performance monitors registers (continued)**

| Register | Description, see |
|---|---|
| PMCEID1_EL0 | *PMCEID1_EL0, Performance Monitors Common Event Identification register 1* on page D7-2243 |
| PMCNTENCLR_EL0 | *PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register* on page D7-2245 |
| PMCNTENSET_EL0 | *PMCNTENSET_EL0, Performance Monitors Count Enable Set register* on page D7-2247 |
| PMCR_EL0 | *PMCR_EL0, Performance Monitors Control Register* on page D7-2249 |
| PMEVCNTR<n>_EL0 | *PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30* on page D7-2253 |
| PMEVTYPER<n>_EL0 | *PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30* on page D7-2255 |
| PMINTENCLR_EL1 | *PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register* on page D7-2258 |
| PMINTENSET_EL1 | *PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register* on page D7-2260 |
| PMOVSCLR_EL0 | *PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register* on page D7-2262 |
| PMOVSSET_EL0 | *PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register* on page D7-2264 |
| PMSELR_EL0 | *PMSELR_EL0, Performance Monitors Event Counter Selection Register* on page D7-2266 |
| PMSWINC_EL0 | *PMSWINC_EL0, Performance Monitors Software Increment register* on page D7-2268 |
| PMUSERENR_EL0 | *PMUSERENR_EL0, Performance Monitors User Enable Register* on page D7-2270 |
| PMXEVCNTR_EL0 | *PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register* on page D7-2272 |
| PMXEVTYPER_EL0 | *PMXEVTYPER_EL0, Performance Monitors Selected Event Type Register* on page D7-2274 |

### K13.3.5 Debug registers

This section is an index to the registers in the Debug registers functional group.

**Table K13-10 Debug registers**

| Register | Description, see |
|---|---|
| DBGAUTHSTATUS_EL1 | *DBGAUTHSTATUS_EL1, Debug Authentication Status register* on page D7-2166 |
| DBGBCR<n>_EL1 | *DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15* on page D7-2168 |
| DBGBVR<n>_EL1 | *DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15* on page D7-2172 |
| DBGCLAIMCLR_EL1 | *DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register* on page D7-2175 |
| DBGCLAIMSET_EL1 | *DBGCLAIMSET_EL1, Debug CLAIM Tag Set register* on page D7-2177 |
| DBGDTR_EL0 | *DBGDTR_EL0, Debug Data Transfer Register, half-duplex* on page D7-2179 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-10 Debug registers (continued)**

| Register | Description, see |
|---|---|
| DBGDTRRX_EL0 | *DBGDTRRX_EL0, Debug Data Transfer Register, Receive* on page D7-2181 |
| DBGDTRTX_EL0 | *DBGDTRTX_EL0, Debug Data Transfer Register, Transmit* on page D7-2183 |
| DBGPRCR_EL1 | *DBGPRCR_EL1, Debug Power Control Register* on page D7-2185 |
| DBGVCR32_EL2 | *DBGVCR32_EL2, Debug Vector Catch Register* on page D7-2187 |
| DBGWCR<n>_EL1 | *DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15* on page D7-2192 |
| DBGWVR<n>_EL1 | *DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15* on page D7-2196 |
| DLR_EL0 | *DLR_EL0, Debug Link Register* on page D7-2198 |
| DSPSR_EL0 | *DSPSR_EL0, Debug Saved Program Status Register* on page D7-2199 |
| MDCCINT_EL1 | *MDCCINT_EL1, Monitor DCC Interrupt Enable Register* on page D7-2204 |
| MDCCSR_EL0 | *MDCCSR_EL0, Monitor DCC Status Register* on page D7-2206 |
| MDCR_EL2 | *MDCR_EL2, Monitor Debug Configuration Register (EL2)* on page D7-2208 |
| MDCR_EL3 | *MDCR_EL3, Monitor Debug Configuration Register (EL3)* on page D7-2212 |
| MDRAR_EL1 | *MDRAR_EL1, Monitor Debug ROM Address Register* on page D7-2216 |
| MDSCR_EL1 | *MDSCR_EL1, Monitor Debug System Control Register* on page D7-2218 |
| OSDLR_EL1 | *OSDLR_EL1, OS Double Lock Register* on page D7-2222 |
| OSDTRRX_EL1 | *OSDTRRX_EL1, OS Lock Data Transfer Register, Receive* on page D7-2224 |
| OSDTRTX_EL1 | *OSDTRTX_EL1, OS Lock Data Transfer Register, Transmit* on page D7-2226 |
| OSECCR_EL1 | *OSECCR_EL1, OS Lock Exception Catch Control Register* on page D7-2228 |
| OSLAR_EL1 | *OSLAR_EL1, OS Lock Access Register* on page D7-2230 |
| OSLSR_EL1 | *OSLSR_EL1, OS Lock Status Register* on page D7-2232 |
| SDER32_EL3 | *SDER32_EL3, AArch32 Secure Debug Enable Register* on page D7-2234 |

### K13.3.6 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

**Table K13-11 Generic timer registers**

| Register | Description, see |
|---|---|
| CNTFRQ_EL0 | *CNTFRQ_EL0, Counter-timer Frequency register* on page D7-2277 |
| CNTHCTL_EL2 | *CNTHCTL_EL2, Counter-timer Hypervisor Control register* on page D7-2279 |
| CNTHP_CTL_EL2 | *CNTHP_CTL_EL2, Counter-timer Hypervisor Physical Timer Control register* on page D7-2281 |
| CNTHP_CVAL_EL2 | *CNTHP_CVAL_EL2, Counter-timer Hypervisor Physical Timer CompareValue register* on page D7-2283 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-11 Generic timer registers (continued)**

| Register | Description, see |
|---|---|
| CNTHP_TVAL_EL2 | *CNTHP_TVAL_EL2, Counter-timer Hypervisor Physical Timer TimerValue register* on page D7-2284 |
| CNTKCTL_EL1 | *CNTKCTL_EL1, Counter-timer Kernel Control register* on page D7-2285 |
| CNTP_CTL_EL0 | *CNTP_CTL_EL0, Counter-timer Physical Timer Control register* on page D7-2288 |
| CNTP_CVAL_EL0 | *CNTP_CVAL_EL0, Counter-timer Physical Timer CompareValue register* on page D7-2290 |
| CNTP_TVAL_EL0 | *CNTP_TVAL_EL0, Counter-timer Physical Timer TimerValue register* on page D7-2291 |
| CNTPCT_EL0 | *CNTPCT_EL0, Counter-timer Physical Count register* on page D7-2293 |
| CNTPS_CTL_EL1 | *CNTPS_CTL_EL1, Counter-timer Physical Secure Timer Control register* on page D7-2294 |
| CNTPS_CVAL_EL1 | *CNTPS_CVAL_EL1, Counter-timer Physical Secure Timer CompareValue register* on page D7-2296 |
| CNTPS_TVAL_EL1 | *CNTPS_TVAL_EL1, Counter-timer Physical Secure Timer TimerValue register* on page D7-2297 |
| CNTV_CTL_EL0 | *CNTV_CTL_EL0, Counter-timer Virtual Timer Control register* on page D7-2299 |
| CNTV_CVAL_EL0 | *CNTV_CVAL_EL0, Counter-timer Virtual Timer CompareValue register* on page D7-2301 |
| CNTV_TVAL_EL0 | *CNTV_TVAL_EL0, Counter-timer Virtual Timer TimerValue register* on page D7-2302 |
| CNTVCT_EL0 | *CNTVCT_EL0, Counter-timer Virtual Count register* on page D7-2304 |
| CNTVOFF_EL2 | *CNTVOFF_EL2, Counter-timer Virtual Offset register* on page D7-2305 |

### K13.3.7 Cache maintenance system instructions

This section is an index to the registers in the Cache maintenance instructions functional group.

**Table K13-12 Cache maintenance system instructions**

| Register | Description, see |
|---|---|
| DC CISW | *DC CISW, Data or unified Cache line Clean and Invalidate by Set/Way* on page C5-343 |
| DC CIVAC | *DC CIVAC, Data or unified Cache line Clean and Invalidate by VA to PoC* on page C5-345 |
| DC CSW | *DC CSW, Data or unified Cache line Clean by Set/Way* on page C5-346 |
| DC CVAC | *DC CVAC, Data or unified Cache line Clean by VA to PoC* on page C5-348 |
| DC CVAU | *DC CVAU, Data or unified Cache line Clean by VA to PoU* on page C5-349 |
| DC ISW | *DC ISW, Data or unified Cache line Invalidate by Set/Way* on page C5-350 |
| DC IVAC | *DC IVAC, Data or unified Cache line Invalidate by VA to PoC* on page C5-352 |
| DC ZVA | *DC ZVA, Data Cache Zero by VA* on page C5-353 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-12 Cache maintenance system instructions (continued)**

| Register | Description, see |
|---|---|
| IC IALLU | *IC IALLU, Instruction Cache Invalidate All to PoU* on page C5-355 |
| IC IALLUIS | *IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable* on page C5-356 |
| IC IVAU | *IC IVAU, Instruction Cache line Invalidate by VA to PoU* on page C5-357 |

### K13.3.8 Address translation system instructions

This section is an index to the registers in the Address translation instructions functional group.

**Table K13-13 Address translation system instructions**

| Register | Description, see |
|---|---|
| AT S12E0R | *AT S12E0R, Address Translate Stages 1 and 2 EL0 Read* on page C5-359 |
| AT S12E0W | *AT S12E0W, Address Translate Stages 1 and 2 EL0 Write* on page C5-360 |
| AT S12E1R | *AT S12E1R, Address Translate Stages 1 and 2 EL1 Read* on page C5-361 |
| AT S12E1W | *AT S12E1W, Address Translate Stages 1 and 2 EL1 Write* on page C5-362 |
| AT S1E0R | *AT S1E0R, Address Translate Stage 1 EL0 Read* on page C5-363 |
| AT S1E0W | *AT S1E0W, Address Translate Stage 1 EL0 Write* on page C5-364 |
| AT S1E1R | *AT S1E1R, Address Translate Stage 1 EL1 Read* on page C5-365 |
| AT S1E1W | *AT S1E1W, Address Translate Stage 1 EL1 Write* on page C5-366 |
| AT S1E2R | *AT S1E2R, Address Translate Stage 1 EL2 Read* on page C5-367 |
| AT S1E2W | *AT S1E2W, Address Translate Stage 1 EL2 Write* on page C5-368 |
| AT S1E3R | *AT S1E3R, Address Translate Stage 1 EL3 Read* on page C5-369 |
| AT S1E3W | *AT S1E3W, Address Translate Stage 1 EL3 Write* on page C5-370 |

### K13.3.9 TLB maintenance system instructions

This section is an index to the registers in the TLB maintenance instructions functional group.

**Table K13-14 TLB maintenance system instructions**

| Register | Description, see |
|---|---|
| TLBI ALLE1 | *TLBI ALLE1, TLB Invalidate All, EL1* on page C5-372 |
| TLBI ALLE1IS | *TLBI ALLE1IS, TLB Invalidate All, EL1, Inner Shareable* on page C5-373 |
| TLBI ALLE2 | *TLBI ALLE2, TLB Invalidate All, EL2* on page C5-374 |
| TLBI ALLE2IS | *TLBI ALLE2IS, TLB Invalidate All, EL2, Inner Shareable* on page C5-375 |
| TLBI ALLE3 | *TLBI ALLE3, TLB Invalidate All, EL3* on page C5-376 |
| TLBI ALLE3IS | *TLBI ALLE3IS, TLB Invalidate All, EL3, Inner Shareable* on page C5-377 |
| TLBI ASIDE1 | *TLBI ASIDE1, TLB Invalidate by ASID, EL1* on page C5-378 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-14 TLB maintenance system instructions (continued)**

| Register | Description, see |
|---|---|
| TLBI ASIDE1IS | *TLBI ASIDE1IS, TLB Invalidate by ASID, EL1, Inner Shareable* on page C5-380 |
| TLBI IPAS2E1 | *TLBI IPAS2E1, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1* on page C5-382 |
| TLBI IPAS2E1IS | *TLBI IPAS2E1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable* on page C5-383 |
| TLBI IPAS2LE1 | *TLBI IPAS2LE1, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1* on page C5-385 |
| TLBI IPAS2LE1IS | *TLBI IPAS2LE1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable* on page C5-386 |
| TLBI VAAE1 | *TLBI VAAE1, TLB Invalidate by VA, All ASID, EL1* on page C5-388 |
| TLBI VAAE1IS | *TLBI VAAE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable* on page C5-390 |
| TLBI VAALE1 | *TLBI VAALE1, TLB Invalidate by VA, All ASID, Last level, EL1* on page C5-392 |
| TLBI VAALE1IS | *TLBI VAALE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable* on page C5-394 |
| TLBI VAE1 | *TLBI VAE1, TLB Invalidate by VA, EL1* on page C5-396 |
| TLBI VAE1IS | *TLBI VAE1IS, TLB Invalidate by VA, EL1, Inner Shareable* on page C5-398 |
| TLBI VAE2 | *TLBI VAE2, TLB Invalidate by VA, EL2* on page C5-400 |
| TLBI VAE2IS | *TLBI VAE2IS, TLB Invalidate by VA, EL2, Inner Shareable* on page C5-402 |
| TLBI VAE3 | *TLBI VAE3, TLB Invalidate by VA, EL3* on page C5-404 |
| TLBI VAE3IS | *TLBI VAE3IS, TLB Invalidate by VA, EL3, Inner Shareable* on page C5-406 |
| TLBI VALE1 | *TLBI VALE1, TLB Invalidate by VA, Last level, EL1* on page C5-408 |
| TLBI VALE1IS | *TLBI VALE1IS, TLB Invalidate by VA, Last level, EL1, Inner Shareable* on page C5-410 |
| TLBI VALE2 | *TLBI VALE2, TLB Invalidate by VA, Last level, EL2* on page C5-412 |
| TLBI VALE2IS | *TLBI VALE2IS, TLB Invalidate by VA, Last level, EL2, Inner Shareable* on page C5-414 |
| TLBI VALE3 | *TLBI VALE3, TLB Invalidate by VA, Last level, EL3* on page C5-416 |
| TLBI VALE3IS | *TLBI VALE3IS, TLB Invalidate by VA, Last level, EL3, Inner Shareable* on page C5-418 |
| TLBI VMALLE1 | *TLBI VMALLE1, TLB Invalidate by VMID, All at stage 1, EL1* on page C5-420 |
| TLBI VMALLE1IS | *TLBI VMALLE1IS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable* on page C5-421 |
| TLBI VMALLS12E1 | *TLBI VMALLS12E1, TLB Invalidate by VMID, All at Stage 1 and 2, EL1* on page C5-422 |
| TLBI VMALLS12E1IS | *TLBI VMALLS12E1IS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable* on page C5-423 |

### K13.3.10 Base System registers

This section is an index to the registers in the functional group.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-15 Base System registers**

| Register | Description, see |
|---|---|
| ACTLR_EL1 | *ACTLR_EL1, Auxiliary Control Register (EL1)* on page D7-1914 |
| ACTLR_EL2 | *ACTLR_EL2, Auxiliary Control Register (EL2)* on page D7-1915 |
| ACTLR_EL3 | *ACTLR_EL3, Auxiliary Control Register (EL3)* on page D7-1916 |
| AFSR0_EL1 | *AFSR0_EL1, Auxiliary Fault Status Register 0 (EL1)* on page D7-1917 |
| AFSR0_EL2 | *AFSR0_EL2, Auxiliary Fault Status Register 0 (EL2)* on page D7-1918 |
| AFSR0_EL3 | *AFSR0_EL3, Auxiliary Fault Status Register 0 (EL3)* on page D7-1919 |
| AFSR1_EL1 | *AFSR1_EL1, Auxiliary Fault Status Register 1 (EL1)* on page D7-1920 |
| AFSR1_EL2 | *AFSR1_EL2, Auxiliary Fault Status Register 1 (EL2)* on page D7-1921 |
| AFSR1_EL3 | *AFSR1_EL3, Auxiliary Fault Status Register 1 (EL3)* on page D7-1922 |
| CPACR_EL1 | *CPACR_EL1, Architectural Feature Access Control Register* on page D7-1934 |
| CPTR_EL2 | *CPTR_EL2, Architectural Feature Trap Register (EL2)* on page D7-1936 |
| CPTR_EL3 | *CPTR_EL3, Architectural Feature Trap Register (EL3)* on page D7-1938 |
| CurrentEL | *CurrentEL, Current Exception Level* on page C5-290 |
| DAIF | *DAIF, Interrupt Mask Bits* on page C5-292 |
| ESR_EL1 | *ESR_EL1, Exception Syndrome Register (EL1)* on page D7-1948 |
| ESR_EL2 | *ESR_EL2, Exception Syndrome Register (EL2)* on page D7-1949 |
| ESR_EL3 | *ESR_EL3, Exception Syndrome Register (EL3)* on page D7-1950 |
| ESR_ELx | *ESR_ELx, Exception Syndrome Register (ELx)* on page D7-1951 |
| FAR_EL1 | *FAR_EL1, Fault Address Register (EL1)* on page D7-1983 |
| FAR_EL2 | *FAR_EL2, Fault Address Register (EL2)* on page D7-1985 |
| FAR_EL3 | *FAR_EL3, Fault Address Register (EL3)* on page D7-1987 |
| FPEXC32_EL2 | *FPEXC32_EL2, Floating-Point Exception Control register* on page D7-1989 |
| HACR_EL2 | *HACR_EL2, Hypervisor Auxiliary Control Register* on page D7-1994 |
| HCR_EL2 | *HCR_EL2, Hypervisor Configuration Register* on page D7-1995 |
| HPFAR_EL2 | *HPFAR_EL2, Hypervisor IPA Fault Address Register* on page D7-2007 |
| HSTR_EL2 | *HSTR_EL2, Hypervisor System Trap Register* on page D7-2009 |
| IFSR32_EL2 | *IFSR32_EL2, Instruction Fault Status Register (EL2)* on page D7-2069 |
| ISR_EL1 | *ISR_EL1, Interrupt Status Register* on page D7-2073 |
| NZCV | *NZCV, Condition Flags* on page C5-306 |
| PAR_EL1 | *PAR_EL1, Physical Address Register* on page D7-2094 |
| RMR_EL1 | *RMR_EL1, Reset Management Register (if EL2 and EL3 not implemented)* on page D7-2098 |
| RMR_EL2 | *RMR_EL2, Reset Management Register (if EL3 not implemented)* on page D7-2100 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-15 Base System registers (continued)**

| Register | Description, see |
|---|---|
| RMR_EL3 | *RMR_EL3, Reset Management Register (if EL3 implemented)* on page D7-2102 |
| RVBAR_EL1 | *RVBAR_EL1, Reset Vector Base Address Register (if EL2 and EL3 not implemented)* on page D7-2104 |
| RVBAR_EL2 | *RVBAR_EL2, Reset Vector Base Address Register (if EL3 not implemented)* on page D7-2105 |
| RVBAR_EL3 | *RVBAR_EL3, Reset Vector Base Address Register (if EL3 implemented)* on page D7-2106 |
| S3_<op1>_<Cn>_<Cm>_<op2> | *S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED registers* on page D7-2107 |
| SCR_EL3 | *SCR_EL3, Secure Configuration Register* on page D7-2108 |
| SCTLR_EL1 | *SCTLR_EL1, System Control Register (EL1)* on page D7-2112 |
| SCTLR_EL2 | *SCTLR_EL2, System Control Register (EL2)* on page D7-2119 |
| SCTLR_EL3 | *SCTLR_EL3, System Control Register (EL3)* on page D7-2123 |
| SPSel | *SPSel, Stack Pointer Select* on page C5-314 |
| TPIDR_EL0 | *TPIDR_EL0, EL0 Read/Write Software Thread ID Register* on page D7-2138 |
| TPIDR_EL1 | *TPIDR_EL1, EL1 Software Thread ID Register* on page D7-2139 |
| TPIDR_EL2 | *TPIDR_EL2, EL2 Software Thread ID Register* on page D7-2140 |
| TPIDR_EL3 | *TPIDR_EL3, EL3 Software Thread ID Register* on page D7-2141 |
| TPIDRRO_EL0 | *TPIDRRO_EL0, EL0 Read-Only Software Thread ID Register* on page D7-2142 |
| VBAR_EL1 | *VBAR_EL1, Vector Base Address Register (EL1)* on page D7-2151 |
| VBAR_EL2 | *VBAR_EL2, Vector Base Address Register (EL2)* on page D7-2153 |
| VBAR_EL3 | *VBAR_EL3, Vector Base Address Register (EL3)* on page D7-2155 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## K13.4 Alphabetical index of AArch32 registers and system instructions

This section is an index of AArch32 registers and system instructions in alphabetical order.

**Table K13-16 Alphabetical index of AArch32 Registers**

| Register | Description, see |
|---|---|
| ACTLR | *ACTLR, Auxiliary Control Register* on page G6-4214 |
| ACTLR2 | *ACTLR2, Auxiliary Control Register 2* on page G6-4216 |
| ADFSR | *ADFSR, Auxiliary Data Fault Status Register* on page G6-4218 |
| AIDR | *AIDR, Auxiliary ID Register* on page G6-4220 |
| AIFSR | *AIFSR, Auxiliary Instruction Fault Status Register* on page G6-4222 |
| AMAIR0 | *AMAIR0, Auxiliary Memory Attribute Indirection Register 0* on page G6-4224 |
| AMAIR1 | *AMAIR1, Auxiliary Memory Attribute Indirection Register 1* on page G6-4226 |
| APSR | *APSR, Application Program Status Register* on page G6-4228 |
| ATS12NSOPR | *ATS12NSOPR, Address Translate Stages 1 and 2 Non-secure Only PL1 Read* on page G6-4230 |
| ATS12NSOPW | *ATS12NSOPW, Address Translate Stages 1 and 2 Non-secure Only PL1 Write* on page G6-4232 |
| ATS12NSOUR | *ATS12NSOUR, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read* on page G6-4234 |
| ATS12NSOUW | *ATS12NSOUW, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write* on page G6-4236 |
| ATS1CPR | *ATS1CPR, Address Translate Stage 1 Current state PL1 Read* on page G6-4238 |
| ATS1CPW | *ATS1CPW, Address Translate Stage 1 Current state PL1 Write* on page G6-4240 |
| ATS1CUR | *ATS1CUR, Address Translate Stage 1 Current state Unprivileged Read* on page G6-4242 |
| ATS1CUW | *ATS1CUW, Address Translate Stage 1 Current state Unprivileged Write* on page G6-4244 |
| ATS1HR | *ATS1HR, Address Translate Stage 1 Hyp mode Read* on page G6-4246 |
| ATS1HW | *ATS1HW, Address Translate Stage 1 Hyp mode Write* on page G6-4248 |
| BPIALL | *BPIALL, Branch Predictor Invalidate All* on page G6-4250 |
| BPIALLIS | *BPIALLIS, Branch Predictor Invalidate All, Inner Shareable* on page G6-4251 |
| BPIMVA | *BPIMVA, Branch Predictor Invalidate by VA* on page G6-4252 |
| CCSIDR | *CCSIDR, Current Cache Size ID Register* on page G6-4254 |
| CLIDR | *CLIDR, Cache Level ID Register* on page G6-4257 |
| CNTFRQ | *CNTFRQ, Counter-timer Frequency register* on page G6-4781 |
| CNTHCTL | *CNTHCTL, Counter-timer Hyp Control register* on page G6-4783 |
| CNTHP_CTL | *CNTHP_CTL, Counter-timer Hyp Physical Timer Control register* on page G6-4785 |
| CNTHP_CVAL | *CNTHP_CVAL, Counter-timer Hyp Physical CompareValue register* on page G6-4787 |
| CNTHP_TVAL | *CNTHP_TVAL, Counter-timer Hyp Physical Timer TimerValue register* on page G6-4789 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-16 Alphabetical index of AArch32 Registers (continued)**

| Register | Description, see |
|---|---|
| CNTKCTL | *CNTKCTL, Counter-timer Kernel Control register* on page G6-4791 |
| CNTP_CTL | *CNTP_CTL, Counter-timer Physical Timer Control register* on page G6-4794 |
| CNTP_CVAL | *CNTP_CVAL, Counter-timer Physical Timer CompareValue register* on page G6-4797 |
| CNTP_TVAL | *CNTP_TVAL, Counter-timer Physical Timer TimerValue register* on page G6-4799 |
| CNTPCT | *CNTPCT, Counter-timer Physical Count register* on page G6-4801 |
| CNTV_CTL | *CNTV_CTL, Counter-timer Virtual Timer Control register* on page G6-4803 |
| CNTV_CVAL | *CNTV_CVAL, Counter-timer Virtual Timer CompareValue register* on page G6-4805 |
| CNTV_TVAL | *CNTV_TVAL, Counter-timer Virtual Timer TimerValue register* on page G6-4807 |
| CNTVCT | *CNTVCT, Counter-timer Virtual Count register* on page G6-4809 |
| CNTVOFF | *CNTVOFF, Counter-timer Virtual Offset register* on page G6-4811 |
| CONTEXTIDR | *CONTEXTIDR, Context ID Register* on page G6-4259 |
| CP15DMB | *CP15DMB, CP15 Data Memory Barrier operation* on page G6-4261 |
| CP15DSB | *CP15DSB, CP15 Data Synchronization Barrier operation* on page G6-4263 |
| CP15ISB | *CP15ISB, CP15 Instruction Synchronization Barrier operation* on page G6-4265 |
| CPACR | *CPACR, Architectural Feature Access Control Register* on page G6-4267 |
| CPSR | *CPSR, Current Program Status Register* on page G6-4271 |
| CSSELR | *CSSELR, Cache Size Selection Register* on page G6-4274 |
| CTR | *CTR, Cache Type Register* on page G6-4276 |
| DACR | *DACR, Domain Access Control Register* on page G6-4278 |
| DBGAUTHSTATUS | *DBGAUTHSTATUS, Debug Authentication Status register* on page G6-4649 |
| DBGBCR<n> | *DBGBCR<n>, Debug Breakpoint Control Registers, n = 0 - 15* on page G6-4651 |
| DBGBVR<n> | *DBGBVR<n>, Debug Breakpoint Value Registers, n = 0 - 15* on page G6-4655 |
| DBGBXVR<n> | *DBGBXVR<n>, Debug Breakpoint Extended Value Registers, n = 0 - 15* on page G6-4658 |
| DBGCLAIMCLR | *DBGCLAIMCLR, Debug CLAIM Tag Clear register* on page G6-4660 |
| DBGCLAIMSET | *DBGCLAIMSET, Debug CLAIM Tag Set register* on page G6-4662 |
| DBGDCCINT | *DBGDCCINT, DCC Interrupt Enable Register* on page G6-4664 |
| DBGDEVID | *DBGDEVID, Debug Device ID register 0* on page G6-4666 |
| DBGDEVID1 | *DBGDEVID1, Debug Device ID register 1* on page G6-4669 |
| DBGDEVID2 | *DBGDEVID2, Debug Device ID register 2* on page G6-4671 |
| DBGDIDR | *DBGDIDR, Debug ID Register* on page G6-4673 |
| DBGDRAR | *DBGDRAR, Debug ROM Address Register* on page G6-4676 |
| DBGDSAR | *DBGDSAR, Debug Self Address Register* on page G6-4679 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-16 Alphabetical index of AArch32 Registers (continued)**

| Register | Description, see |
|---|---|
| DBGDSCRext | *DBGDSCRext, Debug Status and Control Register, External View* on page G6-4681 |
| DBGDSCRint | *DBGDSCRint, Debug Status and Control Register, Internal View* on page G6-4685 |
| DBGDTRRXext | *DBGDTRRXext, Debug OS Lock Data Transfer Register, Receive, External View* on page G6-4688 |
| DBGDTRRXint | *DBGDTRRXint, Debug Data Transfer Register, Receive* on page G6-4690 |
| DBGDTRTXext | *DBGDTRTXext, Debug OS Lock Data Transfer Register, Transmit* on page G6-4692 |
| DBGDTRTXint | *DBGDTRTXint, Debug Data Transfer Register, Transmit* on page G6-4694 |
| DBGOSDLR | *DBGOSDLR, Debug OS Double Lock Register* on page G6-4696 |
| DBGOSECCR | *DBGOSECCR, Debug OS Lock Exception Catch Control Register* on page G6-4698 |
| DBGOSLAR | *DBGOSLAR, Debug OS Lock Access Register* on page G6-4700 |
| DBGOSLSR | *DBGOSLSR, Debug OS Lock Status Register* on page G6-4702 |
| DBGPRCR | *DBGPRCR, Debug Power Control Register* on page G6-4704 |
| DBGVCR | *DBGVCR, Debug Vector Catch Register* on page G6-4706 |
| DBGWCR<n> | *DBGWCR<n>, Debug Watchpoint Control Registers, n = 0 - 15* on page G6-4714 |
| DBGWFAR | *DBGWFAR, Debug Watchpoint Fault Address Register* on page G6-4718 |
| DBGWVR<n> | *DBGWVR<n>, Debug Watchpoint Value Registers, n = 0 - 15* on page G6-4720 |
| DCCIMVAC | *DCCIMVAC, Data Cache line Clean and Invalidate by VA to PoC* on page G6-4280 |
| DCCISW | *DCCISW, Data Cache line Clean and Invalidate by Set/Way* on page G6-4282 |
| DCCMVAC | *DCCMVAC, Data Cache line Clean by VA to PoC* on page G6-4284 |
| DCCMVAU | *DCCMVAU, Data Cache line Clean by VA to PoU* on page G6-4286 |
| DCCSW | *DCCSW, Data Cache line Clean by Set/Way* on page G6-4288 |
| DCIMVAC | *DCIMVAC, Data Cache line Invalidate by VA to PoC* on page G6-4290 |
| DCISW | *DCISW, Data Cache line Invalidate by Set/Way* on page G6-4292 |
| DFAR | *DFAR, Data Fault Address Register* on page G6-4294 |
| DFSR | *DFSR, Data Fault Status Register* on page G6-4296 |
| DLR | *DLR, Debug Link Register* on page G6-4722 |
| DSPSR | *DSPSR, Debug Saved Program Status Register* on page G6-4724 |
| DTLBIALL | *DTLBIALL, Data TLB Invalidate All* on page G6-4302 |
| DTLBIASID | *DTLBIASID, Data TLB Invalidate by ASID match* on page G6-4304 |
| DTLBIMVA | *DTLBIMVA, Data TLB Invalidate by VA* on page G6-4306 |
| ELR_hyp | *ELR_hyp, Exception Link Register (Hyp mode)* on page G6-4308 |
| FCSEIDR | *FCSEIDR, FCSE Process ID register* on page G6-4310 |
| FPEXC | *FPEXC, Floating-Point Exception Control register* on page G6-4312 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-16 Alphabetical index of AArch32 Registers (continued)**

| Register | Description, see |
|---|---|
| FPSCR | *FPSCR, Floating-Point Status and Control Register* on page G6-4317 |
| FPSID | *FPSID, Floating-Point System ID register* on page G6-4323 |
| HACR | *HACR, Hyp Auxiliary Configuration Register* on page G6-4326 |
| HACTLR | *HACTLR, Hyp Auxiliary Control Register* on page G6-4328 |
| HACTLR2 | *HACTLR2, Hyp Auxiliary Control Register 2* on page G6-4330 |
| HADFSR | *HADFSR, Hyp Auxiliary Data Fault Status Register* on page G6-4332 |
| HAIFSR | *HAIFSR, Hyp Auxiliary Instruction Fault Status Register* on page G6-4334 |
| HAMAIR0 | *HAMAIR0, Hyp Auxiliary Memory Attribute Indirection Register 0* on page G6-4336 |
| HAMAIR1 | *HAMAIR1, Hyp Auxiliary Memory Attribute Indirection Register 1* on page G6-4338 |
| HCPTR | *HCPTR, Hyp Architectural Feature Trap Register* on page G6-4340 |
| HCR | *HCR, Hyp Configuration Register* on page G6-4344 |
| HCR2 | *HCR2, Hyp Configuration Register 2* on page G6-4354 |
| HDCR | *HDCR, Hyp Debug Control Register* on page G6-4728 |
| HDFAR | *HDFAR, Hyp Data Fault Address Register* on page G6-4357 |
| HIFAR | *HIFAR, Hyp Instruction Fault Address Register* on page G6-4359 |
| HMAIR0 | *HMAIR0, Hyp Memory Attribute Indirection Register 0* on page G6-4361 |
| HMAIR1 | *HMAIR1, Hyp Memory Attribute Indirection Register 1* on page G6-4364 |
| HPFAR | *HPFAR, Hyp IPA Fault Address Register* on page G6-4367 |
| HRMR | *HRMR, Hyp Reset Management Register* on page G6-4369 |
| HSCTLR | *HSCTLR, Hyp System Control Register* on page G6-4371 |
| HSR | *HSR, Hyp Syndrome Register* on page G6-4376 |
| HSTR | *HSTR, Hyp System Trap Register* on page G6-4395 |
| HTCR | *HTCR, Hyp Translation Control Register* on page G6-4397 |
| HTPIDR | *HTPIDR, Hyp Software Thread ID Register* on page G6-4400 |
| HTTBR | *HTTBR, Hyp Translation Table Base Register* on page G6-4402 |
| HVBAR | *HVBAR, Hyp Vector Base Address Register* on page G6-4404 |
| ICIALLU | *ICIALLU, Instruction Cache Invalidate All to PoU* on page G6-4406 |
| ICIALLUIS | *ICIALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable* on page G6-4408 |
| ICIMVAU | *ICIMVAU, Instruction Cache line Invalidate by VA to PoU* on page G6-4410 |
| ID_AFR0 | *ID_AFR0, Auxiliary Feature Register 0* on page G6-4412 |
| ID_DFR0 | *ID_DFR0, Debug Feature Register 0* on page G6-4414 |
| ID_ISAR0 | *ID_ISAR0, Instruction Set Attribute Register 0* on page G6-4417 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-16 Alphabetical index of AArch32 Registers (continued)**

| Register | Description, see |
|---|---|
| ID_ISAR1 | *ID_ISAR1, Instruction Set Attribute Register 1* on page G6-4420 |
| ID_ISAR2 | *ID_ISAR2, Instruction Set Attribute Register 2* on page G6-4423 |
| ID_ISAR3 | *ID_ISAR3, Instruction Set Attribute Register 3* on page G6-4426 |
| ID_ISAR4 | *ID_ISAR4, Instruction Set Attribute Register 4* on page G6-4429 |
| ID_ISAR5 | *ID_ISAR5, Instruction Set Attribute Register 5* on page G6-4432 |
| ID_MMFR0 | *ID_MMFR0, Memory Model Feature Register 0* on page G6-4434 |
| ID_MMFR1 | *ID_MMFR1, Memory Model Feature Register 1* on page G6-4437 |
| ID_MMFR2 | *ID_MMFR2, Memory Model Feature Register 2* on page G6-4441 |
| ID_MMFR3 | *ID_MMFR3, Memory Model Feature Register 3* on page G6-4445 |
| ID_MMFR4 | *ID_MMFR4, Memory Model Feature Register 4* on page G6-4448 |
| ID_PFR0 | *ID_PFR0, Processor Feature Register 0* on page G6-4450 |
| ID_PFR1 | *ID_PFR1, Processor Feature Register 1* on page G6-4452 |
| IFAR | *IFAR, Instruction Fault Address Register* on page G6-4455 |
| IFSR | *IFSR, Instruction Fault Status Register* on page G6-4457 |
| ISR | *ISR, Interrupt Status Register* on page G6-4462 |
| ITLBIALL | *ITLBIALL, Instruction TLB Invalidate All* on page G6-4464 |
| ITLBIASID | *ITLBIASID, Instruction TLB Invalidate by ASID match* on page G6-4466 |
| ITLBIMVA | *ITLBIMVA, Instruction TLB Invalidate by VA* on page G6-4468 |
| JIDR | *JIDR, Jazelle ID Register* on page G6-4470 |
| JMCR | *JMCR, Jazelle Main Configuration Register* on page G6-4472 |
| JOSCR | *JOSCR, Jazelle OS Control Register* on page G6-4473 |
| MAIR0 | *MAIR0, Memory Attribute Indirection Register 0* on page G6-4475 |
| MAIR1 | *MAIR1, Memory Attribute Indirection Register 1* on page G6-4478 |
| MIDR | *MIDR, Main ID Register* on page G6-4481 |
| MPIDR | *MPIDR, Multiprocessor Affinity Register* on page G6-4484 |
| MVBAR | *MVBAR, Monitor Vector Base Address Register* on page G6-4486 |
| MVFR0 | *MVFR0, Media and VFP Feature Register 0* on page G6-4488 |
| MVFR1 | *MVFR1, Media and VFP Feature Register 1* on page G6-4492 |
| MVFR2 | *MVFR2, Media and VFP Feature Register 2* on page G6-4495 |
| NMRR | *NMRR, Normal Memory Remap Register* on page G6-4497 |
| NSACR | *NSACR, Non-Secure Access Control Register* on page G6-4500 |
| PAR | *PAR, Physical Address Register* on page G6-4504 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-16 Alphabetical index of AArch32 Registers (continued)**

| Register | Description, see |
|---|---|
| PMCCFILTR | *PMCCFILTR, Performance Monitors Cycle Count Filter Register* on page G6-4738 |
| PMCCNTR | *PMCCNTR, Performance Monitors Cycle Count Register* on page G6-4741 |
| PMCEID0 | *PMCEID0, Performance Monitors Common Event Identification register 0* on page G6-4744 |
| PMCEID1 | *PMCEID1, Performance Monitors Common Event Identification register 1* on page G6-4746 |
| PMCNTENCLR | *PMCNTENCLR, Performance Monitors Count Enable Clear register* on page G6-4748 |
| PMCNTENSET | *PMCNTENSET, Performance Monitors Count Enable Set register* on page G6-4750 |
| PMCR | *PMCR, Performance Monitors Control Register* on page G6-4752 |
| PMEVCNTR<n> | *PMEVCNTR<n>, Performance Monitors Event Count Registers, n = 0 - 30* on page G6-4756 |
| PMEVTYPER<n> | *PMEVTYPER<n>, Performance Monitors Event Type Registers, n = 0 - 30* on page G6-4758 |
| PMINTENCLR | *PMINTENCLR, Performance Monitors Interrupt Enable Clear register* on page G6-4761 |
| PMINTENSET | *PMINTENSET, Performance Monitors Interrupt Enable Set register* on page G6-4763 |
| PMOVSR | *PMOVSR, Performance Monitors Overflow Flag Status Register* on page G6-4765 |
| PMOVSSET | *PMOVSSET, Performance Monitors Overflow Flag Status Set register* on page G6-4767 |
| PMSELR | *PMSELR, Performance Monitors Event Counter Selection Register* on page G6-4769 |
| PMSWINC | *PMSWINC, Performance Monitors Software Increment register* on page G6-4771 |
| PMUSERENR | *PMUSERENR, Performance Monitors User Enable Register* on page G6-4773 |
| PMXEVCNTR | *PMXEVCNTR, Performance Monitors Selected Event Count Register* on page G6-4776 |
| PMXEVTYPER | *PMXEVTYPER, Performance Monitors Selected Event Type Register* on page G6-4778 |
| PRRR | *PRRR, Primary Region Remap Register* on page G6-4511 |
| REVIDR | *REVIDR, Revision ID Register* on page G6-4514 |
| RMR (at EL1) | *RMR (at EL1), Reset Management Register* on page G6-4516 |
| RMR (at EL3) | *RMR (at EL3), Reset Management Register* on page G6-4518 |
| RVBAR | *RVBAR, Reset Vector Base Address Register* on page G6-4520 |
| SCR | *SCR, Secure Configuration Register* on page G6-4522 |
| SCTLR | *SCTLR, System Control Register* on page G6-4527 |
| SDCR | *SDCR, Secure Debug Configuration Register* on page G6-4732 |
| SDER | *SDER, Secure Debug Enable Register* on page G6-4735 |
| SPSR | *SPSR, Saved Program Status Register* on page G6-4534 |
| SPSR_abt | *SPSR_abt, Saved Program Status Register (Abort mode)* on page G6-4537 |
| SPSR_fiq | *SPSR_fiq, Saved Program Status Register (FIQ mode)* on page G6-4541 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-16 Alphabetical index of AArch32 Registers (continued)**

| Register | Description, see |
|---|---|
| SPSR_hyp | *SPSR_hyp, Saved Program Status Register (Hyp mode)* on page G6-4545 |
| SPSR_irq | *SPSR_irq, Saved Program Status Register (IRQ mode)* on page G6-4549 |
| SPSR_mon | *SPSR_mon, Saved Program Status Register (Monitor mode)* on page G6-4553 |
| SPSR_svc | *SPSR_svc, Saved Program Status Register (Sup. Call mode)* on page G6-4557 |
| SPSR_und | *SPSR_und, Saved Program Status Register (Undefined mode)* on page G6-4561 |
| TCMTR | *TCMTR, TCM Type Register* on page G6-4565 |
| TLBIALL | *TLBIALL, TLB Invalidate All* on page G6-4567 |
| TLBIALLH | *TLBIALLH, TLB Invalidate All, Hyp mode* on page G6-4569 |
| TLBIALLHIS | *TLBIALLHIS, TLB Invalidate All, Hyp mode, Inner Shareable* on page G6-4571 |
| TLBIALLIS | *TLBIALLIS, TLB Invalidate All, Inner Shareable* on page G6-4573 |
| TLBIALLNSNH | *TLBIALLNSNH, TLB Invalidate All, Non-Secure Non-Hyp* on page G6-4575 |
| TLBIALLNSNHIS | *TLBIALLNSNHIS, TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable* on page G6-4577 |
| TLBIASID | *TLBIASID, TLB Invalidate by ASID match* on page G6-4579 |
| TLBIASIDIS | *TLBIASIDIS, TLB Invalidate by ASID match, Inner Shareable* on page G6-4581 |
| TLBIIPAS2 | *TLBIIPAS2, TLB Invalidate by Intermediate Physical Address, Stage 2* on page G6-4583 |
| TLBIIPAS2IS | *TLBIIPAS2IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable* on page G6-4585 |
| TLBIIPAS2L | *TLBIIPAS2L, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level* on page G6-4587 |
| TLBIIPAS2LIS | *TLBIIPAS2LIS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable* on page G6-4589 |
| TLBIMVA | *TLBIMVA, TLB Invalidate by VA* on page G6-4591 |
| TLBIMVAA | *TLBIMVAA, TLB Invalidate by VA, All ASID* on page G6-4593 |
| TLBIMVAAIS | *TLBIMVAAIS, TLB Invalidate by VA, All ASID, Inner Shareable* on page G6-4595 |
| TLBIMVAAL | *TLBIMVAAL, TLB Invalidate by VA, All ASID, Last level* on page G6-4597 |
| TLBIMVAALIS | *TLBIMVAALIS, TLB Invalidate by VA, All ASID, Last level, Inner Shareable* on page G6-4599 |
| TLBIMVAH | *TLBIMVAH, TLB Invalidate by VA, Hyp mode* on page G6-4601 |
| TLBIMVAHIS | *TLBIMVAHIS, TLB Invalidate by VA, Hyp mode, Inner Shareable* on page G6-4603 |
| TLBIMVAIS | *TLBIMVAIS, TLB Invalidate by VA, Inner Shareable* on page G6-4605 |
| TLBIMVAL | *TLBIMVAL, TLB Invalidate by VA, Last level* on page G6-4607 |
| TLBIMVALH | *TLBIMVALH, TLB Invalidate by VA, Last level, Hyp mode* on page G6-4609 |
| TLBIMVALHIS | *TLBIMVALHIS, TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable* on page G6-4611 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-16 Alphabetical index of AArch32 Registers (continued)**

| Register | Description, see |
|---|---|
| TLBIMVALIS | *TLBIMVALIS, TLB Invalidate by VA, Last level, Inner Shareable* on page G6-4613 |
| TLBTR | *TLBTR, TLB Type Register* on page G6-4615 |
| TPIDRPRW | *TPIDRPRW, PL1 Software Thread ID Register* on page G6-4617 |
| TPIDRURO | *TPIDRURO, PL0 Read-Only Software Thread ID Register* on page G6-4619 |
| TPIDRURW | *TPIDRURW, PL0 Read/Write Software Thread ID Register* on page G6-4621 |
| TTBCR | *TTBCR, Translation Table Base Control Register* on page G6-4623 |
| TTBR0 | *TTBR0, Translation Table Base Register 0* on page G6-4628 |
| TTBR1 | *TTBR1, Translation Table Base Register 1* on page G6-4632 |
| VBAR | *VBAR, Vector Base Address Register* on page G6-4636 |
| VMPIDR | *VMPIDR, Virtualization Multiprocessor ID Register* on page G6-4638 |
| VPIDR | *VPIDR, Virtualization Processor ID Register* on page G6-4640 |
| VTCR | *VTCR, Virtualization Translation Control Register* on page G6-4643 |
| VTTBR | *VTTBR, Virtualization Translation Table Base Register* on page G6-4646 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## K13.5 Functional index of AArch32 registers and system instructions

This section is an index of the AArch32 registers and system instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- *Special-purpose registers*.
- *VMSA-specific registers* on page K13-5741.
- *ID registers* on page K13-5741.
- *Performance monitors registers* on page K13-5743.
- *Debug registers* on page K13-5743.
- *Generic timer registers* on page K13-5745.
- *Cache maintenance system instructions* on page K13-5745.
- *Address translation system instructions* on page K13-5746.
- *TLB maintenance system instructions* on page K13-5746.
- *Legacy feature registers and system instructions* on page K13-5748.
- *Base System registers* on page K13-5748.

### K13.5.1 Special-purpose registers

This section is an index to the registers in the Processor state registers functional group.

**Table K13-17 Special-purpose registers**

| Register | Description, see |
|---|---|
| DLR | *DLR, Debug Link Register* on page G6-4722 |
| DSPSR | *DSPSR, Debug Saved Program Status Register* on page G6-4724 |
| ELR_hyp | *ELR_hyp, Exception Link Register (Hyp mode)* on page G6-4308 |
| FPSCR | *FPSCR, Floating-Point Status and Control Register* on page G6-4317 |
| SPSR | *SPSR, Saved Program Status Register* on page G6-4534 |
| SPSR_abt | *SPSR_abt, Saved Program Status Register (Abort mode)* on page G6-4537 |
| SPSR_fiq | *SPSR_fiq, Saved Program Status Register (FIQ mode)* on page G6-4541 |
| SPSR_hyp | *SPSR_hyp, Saved Program Status Register (Hyp mode)* on page G6-4545 |
| SPSR_irq | *SPSR_irq, Saved Program Status Register (IRQ mode)* on page G6-4549 |
| SPSR_mon | *SPSR_mon, Saved Program Status Register (Monitor mode)* on page G6-4553 |
| SPSR_svc | *SPSR_svc, Saved Program Status Register (Sup. Call mode)* on page G6-4557 |
| SPSR_und | *SPSR_und, Saved Program Status Register (Undefined mode)* on page G6-4561 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K13.5.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

**Table K13-18 VMSA-specific registers**

| Register | Description, see |
|---|---|
| AMAIR0 | *AMAIR0, Auxiliary Memory Attribute Indirection Register 0* on page G6-4224 |
| AMAIR1 | *AMAIR1, Auxiliary Memory Attribute Indirection Register 1* on page G6-4226 |
| CONTEXTIDR | *CONTEXTIDR, Context ID Register* on page G6-4259 |
| DACR | *DACR, Domain Access Control Register* on page G6-4278 |
| HAMAIR0 | *HAMAIR0, Hyp Auxiliary Memory Attribute Indirection Register 0* on page G6-4336 |
| HAMAIR1 | *HAMAIR1, Hyp Auxiliary Memory Attribute Indirection Register 1* on page G6-4338 |
| HMAIR0 | *HMAIR0, Hyp Memory Attribute Indirection Register 0* on page G6-4361 |
| HMAIR1 | *HMAIR1, Hyp Memory Attribute Indirection Register 1* on page G6-4364 |
| HTCR | *HTCR, Hyp Translation Control Register* on page G6-4397 |
| HTTBR | *HTTBR, Hyp Translation Table Base Register* on page G6-4402 |
| MAIR0 | *MAIR0, Memory Attribute Indirection Register 0* on page G6-4475 |
| MAIR1 | *MAIR1, Memory Attribute Indirection Register 1* on page G6-4478 |
| NMRR | *NMRR, Normal Memory Remap Register* on page G6-4497 |
| PRRR | *PRRR, Primary Region Remap Register* on page G6-4511 |
| TTBCR | *TTBCR, Translation Table Base Control Register* on page G6-4623 |
| TTBR0 | *TTBR0, Translation Table Base Register 0* on page G6-4628 |
| TTBR1 | *TTBR1, Translation Table Base Register 1* on page G6-4632 |
| VTCR | *VTCR, Virtualization Translation Control Register* on page G6-4643 |
| VTTBR | *VTTBR, Virtualization Translation Table Base Register* on page G6-4646 |

### K13.5.3 ID registers

This section is an index to the registers in the Identification registers functional group.

**Table K13-19 ID registers**

| Register | Description, see |
|---|---|
| AIDR | *AIDR, Auxiliary ID Register* on page G6-4220 |
| CCSIDR | *CCSIDR, Current Cache Size ID Register* on page G6-4254 |
| CLIDR | *CLIDR, Cache Level ID Register* on page G6-4257 |
| CSSELR | *CSSELR, Cache Size Selection Register* on page G6-4274 |
| CTR | *CTR, Cache Type Register* on page G6-4276 |
| FPSID | *FPSID, Floating-Point System ID register* on page G6-4323 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-19 ID registers (continued)**

| Register | Description, see |
|---|---|
| ID_AFR0 | *ID_AFR0, Auxiliary Feature Register 0* on page G6-4412 |
| ID_DFR0 | *ID_DFR0, Debug Feature Register 0* on page G6-4414 |
| ID_ISAR0 | *ID_ISAR0, Instruction Set Attribute Register 0* on page G6-4417 |
| ID_ISAR1 | *ID_ISAR1, Instruction Set Attribute Register 1* on page G6-4420 |
| ID_ISAR2 | *ID_ISAR2, Instruction Set Attribute Register 2* on page G6-4423 |
| ID_ISAR3 | *ID_ISAR3, Instruction Set Attribute Register 3* on page G6-4426 |
| ID_ISAR4 | *ID_ISAR4, Instruction Set Attribute Register 4* on page G6-4429 |
| ID_ISAR5 | *ID_ISAR5, Instruction Set Attribute Register 5* on page G6-4432 |
| ID_MMFR0 | *ID_MMFR0, Memory Model Feature Register 0* on page G6-4434 |
| ID_MMFR1 | *ID_MMFR1, Memory Model Feature Register 1* on page G6-4437 |
| ID_MMFR2 | *ID_MMFR2, Memory Model Feature Register 2* on page G6-4441 |
| ID_MMFR3 | *ID_MMFR3, Memory Model Feature Register 3* on page G6-4445 |
| ID_MMFR4 | *ID_MMFR4, Memory Model Feature Register 4* on page G6-4448 |
| ID_PFR0 | *ID_PFR0, Processor Feature Register 0* on page G6-4450 |
| ID_PFR1 | *ID_PFR1, Processor Feature Register 1* on page G6-4452 |
| MIDR | *MIDR, Main ID Register* on page G6-4481 |
| MPIDR | *MPIDR, Multiprocessor Affinity Register* on page G6-4484 |
| MVFR0 | *MVFR0, Media and VFP Feature Register 0* on page G6-4488 |
| MVFR1 | *MVFR1, Media and VFP Feature Register 1* on page G6-4492 |
| MVFR2 | *MVFR2, Media and VFP Feature Register 2* on page G6-4495 |
| REVIDR | *REVIDR, Revision ID Register* on page G6-4514 |
| TCMTR | *TCMTR, TCM Type Register* on page G6-4565 |
| TLBTR | *TLBTR, TLB Type Register* on page G6-4615 |
| VMPIDR | *VMPIDR, Virtualization Multiprocessor ID Register* on page G6-4638 |
| VPIDR | *VPIDR, Virtualization Processor ID Register* on page G6-4640 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

### K13.5.4 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

**Table K13-20 Performance monitors registers**

| Register | Description, see |
|---|---|
| PMCCFILTR | *PMCCFILTR, Performance Monitors Cycle Count Filter Register* on page G6-4738 |
| PMCCNTR | *PMCCNTR, Performance Monitors Cycle Count Register* on page G6-4741 |
| PMCEID0 | *PMCEID0, Performance Monitors Common Event Identification register 0* on page G6-4744 |
| PMCEID1 | *PMCEID1, Performance Monitors Common Event Identification register 1* on page G6-4746 |
| PMCNTENCLR | *PMCNTENCLR, Performance Monitors Count Enable Clear register* on page G6-4748 |
| PMCNTENSET | *PMCNTENSET, Performance Monitors Count Enable Set register* on page G6-4750 |
| PMCR | *PMCR, Performance Monitors Control Register* on page G6-4752 |
| PMEVCNTR<n> | *PMEVCNTR<n>, Performance Monitors Event Count Registers, n = 0 - 30* on page G6-4756 |
| PMEVTYPER<n> | *PMEVTYPER<n>, Performance Monitors Event Type Registers, n = 0 - 30* on page G6-4758 |
| PMINTENCLR | *PMINTENCLR, Performance Monitors Interrupt Enable Clear register* on page G6-4761 |
| PMINTENSET | *PMINTENSET, Performance Monitors Interrupt Enable Set register* on page G6-4763 |
| PMOVSR | *PMOVSR, Performance Monitors Overflow Flag Status Register* on page G6-4765 |
| PMOVSSET | *PMOVSSET, Performance Monitors Overflow Flag Status Set register* on page G6-4767 |
| PMSELR | *PMSELR, Performance Monitors Event Counter Selection Register* on page G6-4769 |
| PMSWINC | *PMSWINC, Performance Monitors Software Increment register* on page G6-4771 |
| PMUSERENR | *PMUSERENR, Performance Monitors User Enable Register* on page G6-4773 |
| PMXEVCNTR | *PMXEVCNTR, Performance Monitors Selected Event Count Register* on page G6-4776 |
| PMXEVTYPER | *PMXEVTYPER, Performance Monitors Selected Event Type Register* on page G6-4778 |

### K13.5.5 Debug registers

This section is an index to the registers in the Debug registers functional group.

**Table K13-21 Debug registers**

| Register | Description, see |
|---|---|
| DBGAUTHSTATUS | *DBGAUTHSTATUS, Debug Authentication Status register* on page G6-4649 |
| DBGBCR<n> | *DBGBCR<n>, Debug Breakpoint Control Registers, n = 0 - 15* on page G6-4651 |
| DBGBVR<n> | *DBGBVR<n>, Debug Breakpoint Value Registers, n = 0 - 15* on page G6-4655 |
| DBGBXVR<n> | *DBGBXVR<n>, Debug Breakpoint Extended Value Registers, n = 0 - 15* on page G6-4658 |
| DBGCLAIMCLR | *DBGCLAIMCLR, Debug CLAIM Tag Clear register* on page G6-4660 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-21 Debug registers (continued)**

| Register | Description, see |
|---|---|
| DBGCLAIMSET | *DBGCLAIMSET, Debug CLAIM Tag Set register* on page G6-4662 |
| DBGDCCINT | *DBGDCCINT, DCC Interrupt Enable Register* on page G6-4664 |
| DBGDEVID | *DBGDEVID, Debug Device ID register 0* on page G6-4666 |
| DBGDEVID1 | *DBGDEVID1, Debug Device ID register 1* on page G6-4669 |
| DBGDEVID2 | *DBGDEVID2, Debug Device ID register 2* on page G6-4671 |
| DBGDIDR | *DBGDIDR, Debug ID Register* on page G6-4673 |
| DBGDRAR | *DBGDRAR, Debug ROM Address Register* on page G6-4676 |
| DBGDSAR | *DBGDSAR, Debug Self Address Register* on page G6-4679 |
| DBGDSCRext | *DBGDSCRext, Debug Status and Control Register, External View* on page G6-4681 |
| DBGDSCRint | *DBGDSCRint, Debug Status and Control Register, Internal View* on page G6-4685 |
| DBGDTRRXext | *DBGDTRRXext, Debug OS Lock Data Transfer Register, Receive, External View* on page G6-4688 |
| DBGDTRRXint | *DBGDTRRXint, Debug Data Transfer Register, Receive* on page G6-4690 |
| DBGDTRTXext | *DBGDTRTXext, Debug OS Lock Data Transfer Register, Transmit* on page G6-4692 |
| DBGDTRTXint | *DBGDTRTXint, Debug Data Transfer Register, Transmit* on page G6-4694 |
| DBGOSDLR | *DBGOSDLR, Debug OS Double Lock Register* on page G6-4696 |
| DBGOSECCR | *DBGOSECCR, Debug OS Lock Exception Catch Control Register* on page G6-4698 |
| DBGOSLAR | *DBGOSLAR, Debug OS Lock Access Register* on page G6-4700 |
| DBGOSLSR | *DBGOSLSR, Debug OS Lock Status Register* on page G6-4702 |
| DBGPRCR | *DBGPRCR, Debug Power Control Register* on page G6-4704 |
| DBGVCR | *DBGVCR, Debug Vector Catch Register* on page G6-4706 |
| DBGWCR<n> | *DBGWCR<n>, Debug Watchpoint Control Registers, n = 0 - 15* on page G6-4714 |
| DBGWFAR | *DBGWFAR, Debug Watchpoint Fault Address Register* on page G6-4718 |
| DBGWVR<n> | *DBGWVR<n>, Debug Watchpoint Value Registers, n = 0 - 15* on page G6-4720 |
| DLR | *DLR, Debug Link Register* on page G6-4722 |
| DSPSR | *DSPSR, Debug Saved Program Status Register* on page G6-4724 |
| HDCR | *HDCR, Hyp Debug Control Register* on page G6-4728 |
| SDCR | *SDCR, Secure Debug Configuration Register* on page G6-4732 |
| SDER | *SDER, Secure Debug Enable Register* on page G6-4735 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K13.5.6 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

**Table K13-22 Generic timer registers**

| Register | Description, see |
|---|---|
| CNTFRQ | *CNTFRQ, Counter-timer Frequency register* on page G6-4781 |
| CNTHCTL | *CNTHCTL, Counter-timer Hyp Control register* on page G6-4783 |
| CNTHP_CTL | *CNTHP_CTL, Counter-timer Hyp Physical Timer Control register* on page G6-4785 |
| CNTHP_CVAL | *CNTHP_CVAL, Counter-timer Hyp Physical CompareValue register* on page G6-4787 |
| CNTHP_TVAL | *CNTHP_TVAL, Counter-timer Hyp Physical Timer TimerValue register* on page G6-4789 |
| CNTKCTL | *CNTKCTL, Counter-timer Kernel Control register* on page G6-4791 |
| CNTP_CTL | *CNTP_CTL, Counter-timer Physical Timer Control register* on page G6-4794 |
| CNTP_CVAL | *CNTP_CVAL, Counter-timer Physical Timer CompareValue register* on page G6-4797 |
| CNTP_TVAL | *CNTP_TVAL, Counter-timer Physical Timer TimerValue register* on page G6-4799 |
| CNTPCT | *CNTPCT, Counter-timer Physical Count register* on page G6-4801 |
| CNTV_CTL | *CNTV_CTL, Counter-timer Virtual Timer Control register* on page G6-4803 |
| CNTV_CVAL | *CNTV_CVAL, Counter-timer Virtual Timer CompareValue register* on page G6-4805 |
| CNTV_TVAL | *CNTV_TVAL, Counter-timer Virtual Timer TimerValue register* on page G6-4807 |
| CNTVCT | *CNTVCT, Counter-timer Virtual Count register* on page G6-4809 |
| CNTVOFF | *CNTVOFF, Counter-timer Virtual Offset register* on page G6-4811 |

### K13.5.7 Cache maintenance system instructions

This section is an index to the registers in the Cache maintenance instructions functional group.

**Table K13-23 Cache maintenance system instructions**

| Register | Description, see |
|---|---|
| BPIALL | *BPIALL, Branch Predictor Invalidate All* on page G6-4250 |
| BPIALLIS | *BPIALLIS, Branch Predictor Invalidate All, Inner Shareable* on page G6-4251 |
| BPIMVA | *BPIMVA, Branch Predictor Invalidate by VA* on page G6-4252 |
| DCCIMVAC | *DCCIMVAC, Data Cache line Clean and Invalidate by VA to PoC* on page G6-4280 |
| DCCISW | *DCCISW, Data Cache line Clean and Invalidate by Set/Way* on page G6-4282 |
| DCCMVAC | *DCCMVAC, Data Cache line Clean by VA to PoC* on page G6-4284 |
| DCCMVAU | *DCCMVAU, Data Cache line Clean by VA to PoU* on page G6-4286 |
| DCCSW | *DCCSW, Data Cache line Clean by Set/Way* on page G6-4288 |
| DCIMVAC | *DCIMVAC, Data Cache line Invalidate by VA to PoC* on page G6-4290 |
| DCISW | *DCISW, Data Cache line Invalidate by Set/Way* on page G6-4292 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-23 Cache maintenance system instructions (continued)**

| Register | Description, see |
|---|---|
| ICIALLU | *ICIALLU, Instruction Cache Invalidate All to PoU* on page G6-4406 |
| ICIALLUIS | *ICIALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable* on page G6-4408 |
| ICIMVAU | *ICIMVAU, Instruction Cache line Invalidate by VA to PoU* on page G6-4410 |

### K13.5.8 Address translation system instructions

This section is an index to the registers in the Address translation instructions functional group.

**Table K13-24 Address translation system instructions**

| Register | Description, see |
|---|---|
| ATS12NSOPR | *ATS12NSOPR, Address Translate Stages 1 and 2 Non-secure Only PL1 Read* on page G6-4230 |
| ATS12NSOPW | *ATS12NSOPW, Address Translate Stages 1 and 2 Non-secure Only PL1 Write* on page G6-4232 |
| ATS12NSOUR | *ATS12NSOUR, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read* on page G6-4234 |
| ATS12NSOUW | *ATS12NSOUW, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write* on page G6-4236 |
| ATS1CPR | *ATS1CPR, Address Translate Stage 1 Current state PL1 Read* on page G6-4238 |
| ATS1CPW | *ATS1CPW, Address Translate Stage 1 Current state PL1 Write* on page G6-4240 |
| ATS1CUR | *ATS1CUR, Address Translate Stage 1 Current state Unprivileged Read* on page G6-4242 |
| ATS1CUW | *ATS1CUW, Address Translate Stage 1 Current state Unprivileged Write* on page G6-4244 |
| ATS1HR | *ATS1HR, Address Translate Stage 1 Hyp mode Read* on page G6-4246 |
| ATS1HW | *ATS1HW, Address Translate Stage 1 Hyp mode Write* on page G6-4248 |

### K13.5.9 TLB maintenance system instructions

This section is an index to the registers in the TLB maintenance instructions functional group.

**Table K13-25 TLB maintenance system instructions**

| Register | Description, see |
|---|---|
| DTLBIALL | *DTLBIALL, Data TLB Invalidate All* on page G6-4302 |
| DTLBIASID | *DTLBIASID, Data TLB Invalidate by ASID match* on page G6-4304 |
| DTLBIMVA | *DTLBIMVA, Data TLB Invalidate by VA* on page G6-4306 |
| ITLBIALL | *ITLBIALL, Instruction TLB Invalidate All* on page G6-4464 |
| ITLBIASID | *ITLBIASID, Instruction TLB Invalidate by ASID match* on page G6-4466 |
| ITLBIMVA | *ITLBIMVA, Instruction TLB Invalidate by VA* on page G6-4468 |
| TLBIALL | *TLBIALL, TLB Invalidate All* on page G6-4567 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-25 TLB maintenance system instructions (continued)**

| Register | Description, see |
|---|---|
| TLBIALLH | *TLBIALLH, TLB Invalidate All, Hyp mode* on page G6-4569 |
| TLBIALLHIS | *TLBIALLHIS, TLB Invalidate All, Hyp mode, Inner Shareable* on page G6-4571 |
| TLBIALLIS | *TLBIALLIS, TLB Invalidate All, Inner Shareable* on page G6-4573 |
| TLBIALLNSNH | *TLBIALLNSNH, TLB Invalidate All, Non-Secure Non-Hyp* on page G6-4575 |
| TLBIALLNSNHIS | *TLBIALLNSNHIS, TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable* on page G6-4577 |
| TLBIASID | *TLBIASID, TLB Invalidate by ASID match* on page G6-4579 |
| TLBIASIDIS | *TLBIASIDIS, TLB Invalidate by ASID match, Inner Shareable* on page G6-4581 |
| TLBIIPAS2 | *TLBIIPAS2, TLB Invalidate by Intermediate Physical Address, Stage 2* on page G6-4583 |
| TLBIIPAS2IS | *TLBIIPAS2IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable* on page G6-4585 |
| TLBIIPAS2L | *TLBIIPAS2L, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level* on page G6-4587 |
| TLBIIPAS2LIS | *TLBIIPAS2LIS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable* on page G6-4589 |
| TLBIMVA | *TLBIMVA, TLB Invalidate by VA* on page G6-4591 |
| TLBIMVAA | *TLBIMVAA, TLB Invalidate by VA, All ASID* on page G6-4593 |
| TLBIMVAAIS | *TLBIMVAAIS, TLB Invalidate by VA, All ASID, Inner Shareable* on page G6-4595 |
| TLBIMVAAL | *TLBIMVAAL, TLB Invalidate by VA, All ASID, Last level* on page G6-4597 |
| TLBIMVAALIS | *TLBIMVAALIS, TLB Invalidate by VA, All ASID, Last level, Inner Shareable* on page G6-4599 |
| TLBIMVAH | *TLBIMVAH, TLB Invalidate by VA, Hyp mode* on page G6-4601 |
| TLBIMVAHIS | *TLBIMVAHIS, TLB Invalidate by VA, Hyp mode, Inner Shareable* on page G6-4603 |
| TLBIMVAIS | *TLBIMVAIS, TLB Invalidate by VA, Inner Shareable* on page G6-4605 |
| TLBIMVAL | *TLBIMVAL, TLB Invalidate by VA, Last level* on page G6-4607 |
| TLBIMVALH | *TLBIMVALH, TLB Invalidate by VA, Last level, Hyp mode* on page G6-4609 |
| TLBIMVALHIS | *TLBIMVALHIS, TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable* on page G6-4611 |
| TLBIMVALIS | *TLBIMVALIS, TLB Invalidate by VA, Last level, Inner Shareable* on page G6-4613 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

### K13.5.10 Legacy feature registers and system instructions

This section is an index to the registers in the Legacy feature registers functional group.

**Table K13-26 Legacy feature registers and system instructions**

| Register | Description, see |
|---|---|
| CP15DMB | *CP15DMB, CP15 Data Memory Barrier operation* on page G6-4261 |
| CP15DSB | *CP15DSB, CP15 Data Synchronization Barrier operation* on page G6-4263 |
| CP15ISB | *CP15ISB, CP15 Instruction Synchronization Barrier operation* on page G6-4265 |
| FCSEIDR | *FCSEIDR, FCSE Process ID register* on page G6-4310 |
| JIDR | *JIDR, Jazelle ID Register* on page G6-4470 |
| JMCR | *JMCR, Jazelle Main Configuration Register* on page G6-4472 |
| JOSCR | *JOSCR, Jazelle OS Control Register* on page G6-4473 |

### K13.5.11 Base System registers

This section is an index to the registers in the functional group.

**Table K13-27 Base System registers**

| Register | Description, see |
|---|---|
| ACTLR | *ACTLR, Auxiliary Control Register* on page G6-4214 |
| ACTLR2 | *ACTLR2, Auxiliary Control Register 2* on page G6-4216 |
| ADFSR | *ADFSR, Auxiliary Data Fault Status Register* on page G6-4218 |
| AIFSR | *AIFSR, Auxiliary Instruction Fault Status Register* on page G6-4222 |
| APSR | *APSR, Application Program Status Register* on page G6-4228 |
| CPACR | *CPACR, Architectural Feature Access Control Register* on page G6-4267 |
| CPSR | *CPSR, Current Program Status Register* on page G6-4271 |
| DFAR | *DFAR, Data Fault Address Register* on page G6-4294 |
| DFSR | *DFSR, Data Fault Status Register* on page G6-4296 |
| FPEXC | *FPEXC, Floating-Point Exception Control register* on page G6-4312 |
| HACR | *HACR, Hyp Auxiliary Configuration Register* on page G6-4326 |
| HACTLR | *HACTLR, Hyp Auxiliary Control Register* on page G6-4328 |
| HACTLR2 | *HACTLR2, Hyp Auxiliary Control Register 2* on page G6-4330 |
| HADFSR | *HADFSR, Hyp Auxiliary Data Fault Status Register* on page G6-4332 |
| HAIFSR | *HAIFSR, Hyp Auxiliary Instruction Fault Status Register* on page G6-4334 |
| HCPTR | *HCPTR, Hyp Architectural Feature Trap Register* on page G6-4340 |
| HCR | *HCR, Hyp Configuration Register* on page G6-4344 |
| HCR2 | *HCR2, Hyp Configuration Register 2* on page G6-4354 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Table K13-27 Base System registers (continued)**

| Register | Description, see |
|---|---|
| HDFAR | *HDFAR, Hyp Data Fault Address Register* on page G6-4357 |
| HIFAR | *HIFAR, Hyp Instruction Fault Address Register* on page G6-4359 |
| HPFAR | *HPFAR, Hyp IPA Fault Address Register* on page G6-4367 |
| HRMR | *HRMR, Hyp Reset Management Register* on page G6-4369 |
| HSCTLR | *HSCTLR, Hyp System Control Register* on page G6-4371 |
| HSR | *HSR, Hyp Syndrome Register* on page G6-4376 |
| HSTR | *HSTR, Hyp System Trap Register* on page G6-4395 |
| HTPIDR | *HTPIDR, Hyp Software Thread ID Register* on page G6-4400 |
| HVBAR | *HVBAR, Hyp Vector Base Address Register* on page G6-4404 |
| IFAR | *IFAR, Instruction Fault Address Register* on page G6-4455 |
| IFSR | *IFSR, Instruction Fault Status Register* on page G6-4457 |
| ISR | *ISR, Interrupt Status Register* on page G6-4462 |
| MVBAR | *MVBAR, Monitor Vector Base Address Register* on page G6-4486 |
| NSACR | *NSACR, Non-Secure Access Control Register* on page G6-4500 |
| PAR | *PAR, Physical Address Register* on page G6-4504 |
| RMR (at EL1) | *RMR (at EL1), Reset Management Register* on page G6-4516 |
| RMR (at EL3) | *RMR (at EL3), Reset Management Register* on page G6-4518 |
| RVBAR | *RVBAR, Reset Vector Base Address Register* on page G6-4520 |
| SCR | *SCR, Secure Configuration Register* on page G6-4522 |
| SCTLR | *SCTLR, System Control Register* on page G6-4527 |
| TPIDRPRW | *TPIDRPRW, PL1 Software Thread ID Register* on page G6-4617 |
| TPIDRURO | *TPIDRURO, PL0 Read-Only Software Thread ID Register* on page G6-4619 |
| TPIDRURW | *TPIDRURW, PL0 Read/Write Software Thread ID Register* on page G6-4621 |
| VBAR | *VBAR, Vector Base Address Register* on page G6-4636 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

## K13.6 Alphabetical index of memory-mapped registers

This section is an index of memory-mapped registers in alphabetical order.

**Table K13-28 Alphabetical index of Memory-Mapped Registers**

| Register | Description, see |
|---|---|
| ASICCTL | *ASICCTL, CTI External Multiplexer Control register* on page H9-5055 |
| CNTACR<n> | *CNTACR<n>, Counter-timer Access Control Registers, n = 0 - 7* on page I3-5174 |
| CNTCR | *CNTCR, Counter Control Register* on page I3-5176 |
| CNTCV | *CNTCV, Counter Count Value register* on page I3-5178 |
| CNTEL0ACR | *CNTEL0ACR, Counter-timer EL0 Access Control Register* on page I3-5180 |
| CNTFID0 | *CNTFID0, Counter Frequency ID* on page I3-5182 |
| CNTFID<n> | *CNTFID<n>, Counter Frequency IDs, n = 1 - 23* on page I3-5183 |
| CNTFRQ | *CNTFRQ, Counter-timer Frequency* on page I3-5184 |
| CNTNSAR | *CNTNSAR, Counter-timer Non-secure Access Register* on page I3-5186 |
| CNTP_CTL | *CNTP_CTL, Counter-timer Physical Timer Control* on page I3-5188 |
| CNTP_CVAL | *CNTP_CVAL, Counter-timer Physical Timer CompareValue* on page I3-5190 |
| CNTP_TVAL | *CNTP_TVAL, Counter-timer Physical Timer TimerValue* on page I3-5192 |
| CNTPCT | *CNTPCT, Counter-timer Physical Count* on page I3-5193 |
| CNTSR | *CNTSR, Counter Status Register* on page I3-5195 |
| CNTTIDR | *CNTTIDR, Counter-timer Timer ID Register* on page I3-5197 |
| CNTV_CTL | *CNTV_CTL, Counter-timer Virtual Timer Control* on page I3-5199 |
| CNTV_CVAL | *CNTV_CVAL, Counter-timer Virtual Timer CompareValue* on page I3-5201 |
| CNTV_TVAL | *CNTV_TVAL, Counter-timer Virtual Timer TimerValue* on page I3-5203 |
| CNTVCT | *CNTVCT, Counter-timer Virtual Count* on page I3-5205 |
| CNTVOFF{<n>} | *CNTVOFF{<n>}, Counter-timer Virtual Offset* on page I3-5207 |
| CounterID<n> | *CounterID<n>, Counter ID registers, n = 0 - 11* on page I3-5209 |
| CTIAPPCLEAR | *CTIAPPCLEAR, CTI Application Trigger Clear register* on page H9-5056 |
| CTIAPPPULSE | *CTIAPPPULSE, CTI Application Pulse register* on page H9-5057 |
| CTIAPPSET | *CTIAPPSET, CTI Application Trigger Set register* on page H9-5058 |
| CTIAUTHSTATUS | *CTIAUTHSTATUS, CTI Authentication Status register* on page H9-5059 |
| CTICHINSTATUS | *CTICHINSTATUS, CTI Channel In Status register* on page H9-5061 |
| CTICHOUTSTATUS | *CTICHOUTSTATUS, CTI Channel Out Status register* on page H9-5062 |
| CTICIDR0 | *CTICIDR0, CTI Component Identification Register 0* on page H9-5063 |
| CTICIDR1 | *CTICIDR1, CTI Component Identification Register 1* on page H9-5064 |
| CTICIDR2 | *CTICIDR2, CTI Component Identification Register 2* on page H9-5065 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-28 Alphabetical index of Memory-Mapped Registers (continued)**

| Register | Description, see |
|---|---|
| CTICIDR3 | *CTICIDR3, CTI Component Identification Register 3* on page H9-5066 |
| CTICLAIMCLR | *CTICLAIMCLR, CTI CLAIM Tag Clear register* on page H9-5067 |
| CTICLAIMSET | *CTICLAIMSET, CTI CLAIM Tag Set register* on page H9-5068 |
| CTICONTROL | *CTICONTROL, CTI Control register* on page H9-5069 |
| CTIDEVAFF0 | *CTIDEVAFF0, CTI Device Affinity register 0* on page H9-5070 |
| CTIDEVAFF1 | *CTIDEVAFF1, CTI Device Affinity register 1* on page H9-5071 |
| CTIDEVARCH | *CTIDEVARCH, CTI Device Architecture register* on page H9-5072 |
| CTIDEVID | *CTIDEVID, CTI Device ID register 0* on page H9-5074 |
| CTIDEVID1 | *CTIDEVID1, CTI Device ID register 1* on page H9-5076 |
| CTIDEVID2 | *CTIDEVID2, CTI Device ID register 2* on page H9-5077 |
| CTIDEVTYPE | *CTIDEVTYPE, CTI Device Type register* on page H9-5078 |
| CTIGATE | *CTIGATE, CTI Channel Gate Enable register* on page H9-5079 |
| CTIINEN<n> | *CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31* on page H9-5080 |
| CTIINTACK | *CTIINTACK, CTI Output Trigger Acknowledge register* on page H9-5081 |
| CTIITCTRL | *CTIITCTRL, CTI Integration mode Control register* on page H9-5083 |
| CTILAR | *CTILAR, CTI Lock Access Register* on page H9-5084 |
| CTILSR | *CTILSR, CTI Lock Status Register* on page H9-5085 |
| CTIOUTEN<n> | *CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31* on page H9-5087 |
| CTIPIDR0 | *CTIPIDR0, CTI Peripheral Identification Register 0* on page H9-5088 |
| CTIPIDR1 | *CTIPIDR1, CTI Peripheral Identification Register 1* on page H9-5089 |
| CTIPIDR2 | *CTIPIDR2, CTI Peripheral Identification Register 2* on page H9-5090 |
| CTIPIDR3 | *CTIPIDR3, CTI Peripheral Identification Register 3* on page H9-5091 |
| CTIPIDR4 | *CTIPIDR4, CTI Peripheral Identification Register 4* on page H9-5092 |
| CTITRIGINSTATUS | *CTITRIGINSTATUS, CTI Trigger In Status register* on page H9-5093 |
| CTITRIGOUTSTATUS | *CTITRIGOUTSTATUS, CTI Trigger Out Status register* on page H9-5094 |
| DBGAUTHSTATUS_EL1 | *DBGAUTHSTATUS_EL1, Debug Authentication Status register* on page H9-4968 |
| DBGBCR<n>_EL1 | *DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15* on page H9-4970 |
| DBGBVR<n>_EL1 | *DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15* on page H9-4973 |
| DBGCLAIMCLR_EL1 | *DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register* on page H9-4976 |
| DBGCLAIMSET_EL1 | *DBGCLAIMSET_EL1, Debug CLAIM Tag Set register* on page H9-4978 |
| DBGDTRRX_EL0 | *DBGDTRRX_EL0, Debug Data Transfer Register, Receive* on page H9-4980 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-28 Alphabetical index of Memory-Mapped Registers (continued)**

| Register | Description, see |
|---|---|
| DBGDTRTX_EL0 | *DBGDTRTX_EL0, Debug Data Transfer Register, Transmit* on page H9-4982 |
| DBGWCR<n>_EL1 | *DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15* on page H9-4984 |
| DBGWVR<n>_EL1 | *DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15* on page H9-4987 |
| EDAA32PFR | *EDAA32PFR, External Debug AArch32 Processor Feature Register* on page H9-4989 |
| EDACR | *EDACR, External Debug Auxiliary Control Register* on page H9-4991 |
| EDCIDR0 | *EDCIDR0, External Debug Component Identification Register 0* on page H9-4992 |
| EDCIDR1 | *EDCIDR1, External Debug Component Identification Register 1* on page H9-4993 |
| EDCIDR2 | *EDCIDR2, External Debug Component Identification Register 2* on page H9-4994 |
| EDCIDR3 | *EDCIDR3, External Debug Component Identification Register 3* on page H9-4995 |
| EDCIDSR | *EDCIDSR, External Debug Context ID Sample Register* on page H9-4996 |
| EDDEVAFF0 | *EDDEVAFF0, External Debug Device Affinity register 0* on page H9-4997 |
| EDDEVAFF1 | *EDDEVAFF1, External Debug Device Affinity register 1* on page H9-4998 |
| EDDEVARCH | *EDDEVARCH, External Debug Device Architecture register* on page H9-4999 |
| EDDEVID | *EDDEVID, External Debug Device ID register 0* on page H9-5001 |
| EDDEVID1 | *EDDEVID1, External Debug Device ID register 1* on page H9-5003 |
| EDDEVID2 | *EDDEVID2, External Debug Device ID register 2* on page H9-5004 |
| EDDEVTYPE | *EDDEVTYPE, External Debug Device Type register* on page H9-5005 |
| EDDFR | *EDDFR, External Debug Feature Register* on page H9-5006 |
| EDECCR | *EDECCR, External Debug Exception Catch Control Register* on page H9-5008 |
| EDECR | *EDECR, External Debug Execution Control Register* on page H9-5010 |
| EDESR | *EDESR, External Debug Event Status Register* on page H9-5012 |
| EDITCTRL | *EDITCTRL, External Debug Integration mode Control register* on page H9-5014 |
| EDITR | *EDITR, External Debug Instruction Transfer Register* on page H9-5016 |
| EDLAR | *EDLAR, External Debug Lock Access Register* on page H9-5018 |
| EDLSR | *EDLSR, External Debug Lock Status Register* on page H9-5019 |
| EDPCSR | *EDPCSR, External Debug Program Counter Sample Register* on page H9-5021 |
| EDPFR | *EDPFR, External Debug Processor Feature Register* on page H9-5023 |
| EDPIDR0 | *EDPIDR0, External Debug Peripheral Identification Register 0* on page H9-5025 |
| EDPIDR1 | *EDPIDR1, External Debug Peripheral Identification Register 1* on page H9-5026 |
| EDPIDR2 | *EDPIDR2, External Debug Peripheral Identification Register 2* on page H9-5027 |
| EDPIDR3 | *EDPIDR3, External Debug Peripheral Identification Register 3* on page H9-5028 |
| EDPIDR4 | *EDPIDR4, External Debug Peripheral Identification Register 4* on page H9-5029 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-28 Alphabetical index of Memory-Mapped Registers (continued)**

| Register | Description, see |
|---|---|
| EDPRCR | *EDPRCR, External Debug Power/Reset Control Register* on page H9-5030 |
| EDPRSR | *EDPRSR, External Debug Processor Status Register* on page H9-5033 |
| EDRCR | *EDRCR, External Debug Reserve Control Register* on page H9-5041 |
| EDSCR | *EDSCR, External Debug Status and Control Register* on page H9-5043 |
| EDVIDSR | *EDVIDSR, External Debug Virtual Context Sample Register* on page H9-5048 |
| EDWAR | *EDWAR, External Debug Watchpoint Address Register* on page H9-5050 |
| MIDR_EL1 | *MIDR_EL1, Main ID Register* on page H9-5051 |
| OSLAR_EL1 | *OSLAR_EL1, OS Lock Access Register* on page H9-5053 |
| PMAUTHSTATUS | *PMAUTHSTATUS, Performance Monitors Authentication Status register* on page I3-5122 |
| PMCCFILTR_EL0 | *PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register* on page I3-5124 |
| PMCCNTR_EL0 | *PMCCNTR_EL0, Performance Monitors Cycle Counter* on page I3-5126 |
| PMCEID0_EL0 | *PMCEID0_EL0, Performance Monitors Common Event Identification register 0* on page I3-5128 |
| PMCEID1_EL0 | *PMCEID1_EL0, Performance Monitors Common Event Identification register 1* on page I3-5129 |
| PMCFGR | *PMCFGR, Performance Monitors Configuration Register* on page I3-5130 |
| PMCIDR0 | *PMCIDR0, Performance Monitors Component Identification Register 0* on page I3-5132 |
| PMCIDR1 | *PMCIDR1, Performance Monitors Component Identification Register 1* on page I3-5133 |
| PMCIDR2 | *PMCIDR2, Performance Monitors Component Identification Register 2* on page I3-5134 |
| PMCIDR3 | *PMCIDR3, Performance Monitors Component Identification Register 3* on page I3-5135 |
| PMCNTENCLR_EL0 | *PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register* on page I3-5136 |
| PMCNTENSET_EL0 | *PMCNTENSET_EL0, Performance Monitors Count Enable Set register* on page I3-5138 |
| PMCR_EL0 | *PMCR_EL0, Performance Monitors Control Register* on page I3-5140 |
| PMDEVAFF0 | *PMDEVAFF0, Performance Monitors Device Affinity register 0* on page I3-5143 |
| PMDEVAFF1 | *PMDEVAFF1, Performance Monitors Device Affinity register 1* on page I3-5144 |
| PMDEVARCH | *PMDEVARCH, Performance Monitors Device Architecture register* on page I3-5145 |
| PMDEVTYPE | *PMDEVTYPE, Performance Monitors Device Type register* on page I3-5147 |
| PMEVCNTR<n>_EL0 | *PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30* on page I3-5148 |
| PMEVTYPER<n>_EL0 | *PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30* on page I3-5149 |
| PMINTENCLR_EL1 | *PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register* on page I3-5152 |
| PMINTENSET_EL1 | *PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register* on page I3-5154 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-28 Alphabetical index of Memory-Mapped Registers (continued)**

| Register | Description, see |
|---|---|
| PMITCTRL | *PMITCTRL, Performance Monitors Integration mode Control register* on page I3-5156 |
| PMLAR | *PMLAR, Performance Monitors Lock Access Register* on page I3-5158 |
| PMLSR | *PMLSR, Performance Monitors Lock Status Register* on page I3-5159 |
| PMOVSCLR_EL0 | *PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear register* on page I3-5161 |
| PMOVSSET_EL0 | *PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register* on page I3-5163 |
| PMPIDR0 | *PMPIDR0, Performance Monitors Peripheral Identification Register 0* on page I3-5165 |
| PMPIDR1 | *PMPIDR1, Performance Monitors Peripheral Identification Register 1* on page I3-5166 |
| PMPIDR2 | *PMPIDR2, Performance Monitors Peripheral Identification Register 2* on page I3-5167 |
| PMPIDR3 | *PMPIDR3, Performance Monitors Peripheral Identification Register 3* on page I3-5168 |
| PMPIDR4 | *PMPIDR4, Performance Monitors Peripheral Identification Register 4* on page I3-5169 |
| PMSWINC_EL0 | *PMSWINC_EL0, Performance Monitors Software Increment register* on page I3-5170 |


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

## K13.7 Functional index of memory-mapped registers

This section is an index of the memory-mapped registers, divided by functional group.

### K13.7.1 ID registers

This section is an index to the registers in the Identification registers functional group.

**Table K13-29 ID registers**

| Register | Description, see |
|---|---|
| EDAA32PFR | *EDAA32PFR, External Debug AArch32 Processor Feature Register* on page H9-4989 |
| EDDFR | *EDDFR, External Debug Feature Register* on page H9-5006 |
| EDPFR | *EDPFR, External Debug Processor Feature Register* on page H9-5023 |
| MIDR_EL1 | *MIDR_EL1, Main ID Register* on page H9-5051 |

### K13.7.2 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

**Table K13-30 Performance monitors registers**

| Register | Description, see |
|---|---|
| PMAUTHSTATUS | *PMAUTHSTATUS, Performance Monitors Authentication Status register* on page I3-5122 |
| PMCCFILTR_EL0 | *PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register* on page I3-5124 |
| PMCCNTR_EL0 | *PMCCNTR_EL0, Performance Monitors Cycle Counter* on page I3-5126 |
| PMCEID0_EL0 | *PMCEID0_EL0, Performance Monitors Common Event Identification register 0* on page I3-5128 |
| PMCEID1_EL0 | *PMCEID1_EL0, Performance Monitors Common Event Identification register 1* on page I3-5129 |
| PMCFGR | *PMCFGR, Performance Monitors Configuration Register* on page I3-5130 |
| PMCIDR0 | *PMCIDR0, Performance Monitors Component Identification Register 0* on page I3-5132 |
| PMCIDR1 | *PMCIDR1, Performance Monitors Component Identification Register 1* on page I3-5133 |
| PMCIDR2 | *PMCIDR2, Performance Monitors Component Identification Register 2* on page I3-5134 |
| PMCIDR3 | *PMCIDR3, Performance Monitors Component Identification Register 3* on page I3-5135 |
| PMCNTENCLR_EL0 | *PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register* on page I3-5136 |
| PMCNTENSET_EL0 | *PMCNTENSET_EL0, Performance Monitors Count Enable Set register* on page I3-5138 |
| PMCR_EL0 | *PMCR_EL0, Performance Monitors Control Register* on page I3-5140 |
| PMDEVAFF0 | *PMDEVAFF0, Performance Monitors Device Affinity register 0* on page I3-5143 |
| PMDEVAFF1 | *PMDEVAFF1, Performance Monitors Device Affinity register 1* on page I3-5144 |
| PMDEVARCH | *PMDEVARCH, Performance Monitors Device Architecture register* on page I3-5145 |
| PMDEVTYPE | *PMDEVTYPE, Performance Monitors Device Type register* on page I3-5147 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-30 Performance monitors registers (continued)**

| Register | Description, see |
|---|---|
| PMEVCNTR<n>_EL0 | *PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30* on page I3-5148 |
| PMEVTYPER<n>_EL0 | *PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30* on page I3-5149 |
| PMINTENCLR_EL1 | *PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register* on page I3-5152 |
| PMINTENSET_EL1 | *PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register* on page I3-5154 |
| PMITCTRL | *PMITCTRL, Performance Monitors Integration mode Control register* on page I3-5156 |
| PMLAR | *PMLAR, Performance Monitors Lock Access Register* on page I3-5158 |
| PMLSR | *PMLSR, Performance Monitors Lock Status Register* on page I3-5159 |
| PMOVSCLR_EL0 | *PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear register* on page I3-5161 |
| PMOVSSET_EL0 | *PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register* on page I3-5163 |
| PMPIDR0 | *PMPIDR0, Performance Monitors Peripheral Identification Register 0* on page I3-5165 |
| PMPIDR1 | *PMPIDR1, Performance Monitors Peripheral Identification Register 1* on page I3-5166 |
| PMPIDR2 | *PMPIDR2, Performance Monitors Peripheral Identification Register 2* on page I3-5167 |
| PMPIDR3 | *PMPIDR3, Performance Monitors Peripheral Identification Register 3* on page I3-5168 |
| PMPIDR4 | *PMPIDR4, Performance Monitors Peripheral Identification Register 4* on page I3-5169 |
| PMSWINC_EL0 | *PMSWINC_EL0, Performance Monitors Software Increment register* on page I3-5170 |

### K13.7.3 Debug registers

This section is an index to the registers in the Debug registers functional group.

**Table K13-31 Debug registers**

| Register | Description, see |
|---|---|
| DBGAUTHSTATUS_EL1 | *DBGAUTHSTATUS_EL1, Debug Authentication Status register* on page H9-4968 |
| DBGBCR<n>_EL1 | *DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15* on page H9-4970 |
| DBGBVR<n>_EL1 | *DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15* on page H9-4973 |
| DBGCLAIMCLR_EL1 | *DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register* on page H9-4976 |
| DBGCLAIMSET_EL1 | *DBGCLAIMSET_EL1, Debug CLAIM Tag Set register* on page H9-4978 |
| DBGDTRRX_EL0 | *DBGDTRRX_EL0, Debug Data Transfer Register, Receive* on page H9-4980 |
| DBGDTRTX_EL0 | *DBGDTRTX_EL0, Debug Data Transfer Register, Transmit* on page H9-4982 |
| DBGWCR<n>_EL1 | *DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15* on page H9-4984 |
| DBGWVR<n>_EL1 | *DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15* on page H9-4987 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-31 Debug registers (continued)**

| Register | Description, see |
|---|---|
| EDACR | *EDACR, External Debug Auxiliary Control Register* on page H9-4991 |
| EDCIDR0 | *EDCIDR0, External Debug Component Identification Register 0* on page H9-4992 |
| EDCIDR1 | *EDCIDR1, External Debug Component Identification Register 1* on page H9-4993 |
| EDCIDR2 | *EDCIDR2, External Debug Component Identification Register 2* on page H9-4994 |
| EDCIDR3 | *EDCIDR3, External Debug Component Identification Register 3* on page H9-4995 |
| EDCIDSR | *EDCIDSR, External Debug Context ID Sample Register* on page H9-4996 |
| EDDEVAFF0 | *EDDEVAFF0, External Debug Device Affinity register 0* on page H9-4997 |
| EDDEVAFF1 | *EDDEVAFF1, External Debug Device Affinity register 1* on page H9-4998 |
| EDDEVARCH | *EDDEVARCH, External Debug Device Architecture register* on page H9-4999 |
| EDDEVID | *EDDEVID, External Debug Device ID register 0* on page H9-5001 |
| EDDEVID1 | *EDDEVID1, External Debug Device ID register 1* on page H9-5003 |
| EDDEVID2 | *EDDEVID2, External Debug Device ID register 2* on page H9-5004 |
| EDDEVTYPE | *EDDEVTYPE, External Debug Device Type register* on page H9-5005 |
| EDECCR | *EDECCR, External Debug Exception Catch Control Register* on page H9-5008 |
| EDECR | *EDECR, External Debug Execution Control Register* on page H9-5010 |
| EDESR | *EDESR, External Debug Event Status Register* on page H9-5012 |
| EDITCTRL | *EDITCTRL, External Debug Integration mode Control register* on page H9-5014 |
| EDITR | *EDITR, External Debug Instruction Transfer Register* on page H9-5016 |
| EDLAR | *EDLAR, External Debug Lock Access Register* on page H9-5018 |
| EDLSR | *EDLSR, External Debug Lock Status Register* on page H9-5019 |
| EDPCSR | *EDPCSR, External Debug Program Counter Sample Register* on page H9-5021 |
| EDPIDR0 | *EDPIDR0, External Debug Peripheral Identification Register 0* on page H9-5025 |
| EDPIDR1 | *EDPIDR1, External Debug Peripheral Identification Register 1* on page H9-5026 |
| EDPIDR2 | *EDPIDR2, External Debug Peripheral Identification Register 2* on page H9-5027 |
| EDPIDR3 | *EDPIDR3, External Debug Peripheral Identification Register 3* on page H9-5028 |
| EDPIDR4 | *EDPIDR4, External Debug Peripheral Identification Register 4* on page H9-5029 |
| EDPRCR | *EDPRCR, External Debug Power/Reset Control Register* on page H9-5030 |
| EDPRSR | *EDPRSR, External Debug Processor Status Register* on page H9-5033 |
| EDRCR | *EDRCR, External Debug Reserve Control Register* on page H9-5041 |
| EDSCR | *EDSCR, External Debug Status and Control Register* on page H9-5043 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-31 Debug registers (continued)**

| Register | Description, see |
|---|---|
| EDVIDSR | *EDVIDSR, External Debug Virtual Context Sample Register* on page H9-5048 |
| EDWAR | *EDWAR, External Debug Watchpoint Address Register* on page H9-5050 |
| OSLAR_EL1 | *OSLAR_EL1, OS Lock Access Register* on page H9-5053 |

### K13.7.4 Cross-trigger interface registers

This section is an index to the registers in the Cross-Trigger Interface registers functional group.

**Table K13-32 Cross-trigger interface registers**

| Register | Description, see |
|---|---|
| ASICCTL | *ASICCTL, CTI External Multiplexer Control register* on page H9-5055 |
| CTIAPPCLEAR | *CTIAPPCLEAR, CTI Application Trigger Clear register* on page H9-5056 |
| CTIAPPPULSE | *CTIAPPPULSE, CTI Application Pulse register* on page H9-5057 |
| CTIAPPSET | *CTIAPPSET, CTI Application Trigger Set register* on page H9-5058 |
| CTIAUTHSTATUS | *CTIAUTHSTATUS, CTI Authentication Status register* on page H9-5059 |
| CTICHINSTATUS | *CTICHINSTATUS, CTI Channel In Status register* on page H9-5061 |
| CTICHOUTSTATUS | *CTICHOUTSTATUS, CTI Channel Out Status register* on page H9-5062 |
| CTICIDR0 | *CTICIDR0, CTI Component Identification Register 0* on page H9-5063 |
| CTICIDR1 | *CTICIDR1, CTI Component Identification Register 1* on page H9-5064 |
| CTICIDR2 | *CTICIDR2, CTI Component Identification Register 2* on page H9-5065 |
| CTICIDR3 | *CTICIDR3, CTI Component Identification Register 3* on page H9-5066 |
| CTICLAIMCLR | *CTICLAIMCLR, CTI CLAIM Tag Clear register* on page H9-5067 |
| CTICLAIMSET | *CTICLAIMSET, CTI CLAIM Tag Set register* on page H9-5068 |
| CTICONTROL | *CTICONTROL, CTI Control register* on page H9-5069 |
| CTIDEVAFF0 | *CTIDEVAFF0, CTI Device Affinity register 0* on page H9-5070 |
| CTIDEVAFF1 | *CTIDEVAFF1, CTI Device Affinity register 1* on page H9-5071 |
| CTIDEVARCH | *CTIDEVARCH, CTI Device Architecture register* on page H9-5072 |
| CTIDEVID | *CTIDEVID, CTI Device ID register 0* on page H9-5074 |
| CTIDEVID1 | *CTIDEVID1, CTI Device ID register 1* on page H9-5076 |
| CTIDEVID2 | *CTIDEVID2, CTI Device ID register 2* on page H9-5077 |
| CTIDEVTYPE | *CTIDEVTYPE, CTI Device Type register* on page H9-5078 |
| CTIGATE | *CTIGATE, CTI Channel Gate Enable register* on page H9-5079 |
| CTIINEN<n> | *CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31* on page H9-5080 |
| CTIINTACK | *CTIINTACK, CTI Output Trigger Acknowledge register* on page H9-5081 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Table K13-32 Cross-trigger interface registers (continued)**

| Register | Description, see |
|---|---|
| CTIITCTRL | *CTIITCTRL, CTI Integration mode Control register* on page H9-5083 |
| CTILAR | *CTILAR, CTI Lock Access Register* on page H9-5084 |
| CTILSR | *CTILSR, CTI Lock Status Register* on page H9-5085 |
| CTIOUTEN<n> | *CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31* on page H9-5087 |
| CTIPIDR0 | *CTIPIDR0, CTI Peripheral Identification Register 0* on page H9-5088 |
| CTIPIDR1 | *CTIPIDR1, CTI Peripheral Identification Register 1* on page H9-5089 |
| CTIPIDR2 | *CTIPIDR2, CTI Peripheral Identification Register 2* on page H9-5090 |
| CTIPIDR3 | *CTIPIDR3, CTI Peripheral Identification Register 3* on page H9-5091 |
| CTIPIDR4 | *CTIPIDR4, CTI Peripheral Identification Register 4* on page H9-5092 |
| CTITRIGINSTATUS | *CTITRIGINSTATUS, CTI Trigger In Status register* on page H9-5093 |
| CTITRIGOUTSTATUS | *CTITRIGOUTSTATUS, CTI Trigger Out Status register* on page H9-5094 |


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

# Glossary

**Note**

The update of the Glossary from ARMv7 to ARMv8 remains work-in-progress.

**A32 instruction** A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in A32 state. A32 instructions must be word-aligned.

A32 instructions were previously called ARM instructions.

*See also* A32 state, A64 instruction, T32 instruction.

**A32 state** The AArch32 Instruction set state in which the PE executes A32 instructions.

A32 state was previously called ARM state.

*See also* T32 instruction, T32 state.

**A64 instruction** A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch64. A64 instructions must be word-aligned.

*See also* A32 instruction, T32 instruction.

**AArch32** The 32-bit Execution state. In AArch32 state, addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets

*See also* AArch64, A32 instruction, T32 instruction.

**AArch64** The 64-bit Execution state. In AArch64 state, addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

*See also* AArch32, A64 instruction.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Abort** An exception caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU.

**Addressing mode**
Means a method for generating the memory address used by a load/store instruction.

**Advanced SIMD** A feature of the ARM architecture that provides SIMD operations on a register file of SIMD and floating-point registers. Where an implementation supports both Advanced SIMD and floating-point instructions, these instructions operate on the same register file.

**Aligned** A data item stored at an address that is exactly divisible by the highest power of 2 that divides exactly into its size in bytes. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of each element of the access.

**Architecturally executed**
An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired it has been *architecturally executed*. Any instruction that, in a simple sequential execution of a program, is treated as a `NOP` because it fails its condition code check, is an architecturally executed instruction.

In a PE that performs speculative execution, an instruction is not architecturally executed if the PE discards the results of a speculative execution.

*See also* Condition code check, Simple sequential execution.

**Architecturally mapped**
Where this manual describes a register as being *architecturally mapped* to another register, this indicates that, in an implementation that supports both of the registers, the two registers access the same state.

**Architecturally UNKNOWN**
An architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of UNKNOWN. Implementations can define the value of the field, but are not required to do so.

*See also* IMPLEMENTATION DEFINED.

**ARM core registers**
Some older documentation uses *ARM core registers* to refer to the following set of registers for execution in AArch32 state:

- The 13 general-purpose registers, R0-R12, that software can use for processing.
- SP, the *stack pointer*, that can also be referred to as R13.
- LR, the *link register*, that can also be referred to as R14.
- PC, the *program counter*, that can also be referred to as R15.

*See also* General-purpose registers.

**ARM instruction**
*See* A32 instruction.

**Associativity** *See* Cache associativity.

**Atomicity** Describes either single-copy atomicity or multi-copy atomicity. *Atomicity in the ARM architecture* on page B2-80 defines these forms of atomicity for the ARM architecture.

*See also* Multi-copy atomicity, Single-copy atomicity.

**Banked register** A register that has multiple instances, with the instance that is in use depending on the PE mode, Security state, or other PE state.

**Base register** A register specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

**Base register writeback**
Describes writing back a modified value to the base register used in an address calculation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Behaves as if** Where this manual indicates that a PE *behaves as if* a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.

**Big-endian memory**

Means that, for example:

- A byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the most significant byte in the halfword at that address.

*See also* Endianness, Little-endian memory.

**Blocking** Describes an operation that does not permit following instructions to be executed before the operation completes.

A non-blocking operation can permit following instructions to be executed before the operation completes, and in the event of encountering an exception does not signal an exception to the PE. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise PE state.

**Branch prediction**

Is where a PE selects a future execution path to fetch along. For example, after a branch instruction, the PE can choose to speculatively fetch either the instruction following the branch or the instruction at the branch target.

*See also* Prefetching.

**Breakpoint** A debug event triggered by the execution of a particular instruction, specified by one or both of the address of the instruction and the state of the PE when the instruction is executed.

**Byte** An 8-bit data item.

**Cache associativity**

The number of locations in a cache set to which an address can be assigned. Each location is identified by its *way* value.

**Cache level** The position of a cache in the cache hierarchy. In the ARM architecture, the lower numbered levels are those closest to the PE. For more information, see *Terms used in describing the maintenance instructions* on page D3-1712.

**Cache line** The basic unit of storage in a cache. Its size in words is always a power of two, usually 4 or 8 words. A cache line must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely called cache lines.

**Cache lockdown**

Enables critical software and data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. It alleviates the delays caused by accessing a cache in a worst-case situation. This ensures that all subsequent accesses to the software and data concerned are cache hits and so complete quickly.

**Cache miss** A memory access that cannot be processed at high speed because the data it addresses is not in the cache.

**Cache sets** Areas of a cache, divided up to simplify and speed up the process of determining whether a cache hit occurs. The number of cache sets is always a power of two.

**Cache way** A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to (Associativity-1). Each cache line in a cache way is chosen to have the same index as the cache way. For example, cache way *n* consists of the cache line with index *n* from each cache set.

**Coherence order**

*See* Coherent.

**Coherent** Data accesses from a set of observers to a byte in memory are coherent if accesses to that byte in memory by the members of that set of observers are consistent with there being a single total order of all writes to that byte in memory by all members of the set of observers. This single total order of all to writes to that memory location is the *coherence order* for that byte in memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Condition code check**
The process of determining whether a conditional instruction executes normally or is treated as a `NOP`. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally. For a T32 instruction in an IT block, the value of PSTATE.IT determines whether the instruction is executed normally.

*See also* Condition code field, Condition flags, Conditional execution.

**Condition code field**
A 4-bit field in an instruction that specifies the condition under which the instruction executes.

*See also* Condition code check.

**Condition flags** The N, Z, C, and V bits of PSTATE, an SPSR, or FPSCR. See the register descriptions for more information.

*See also* Condition code check, PSTATE.

**Conditional execution**
When a conditional instruction starts executing, if the condition code check returns TRUE, the instruction executes normally. Otherwise, it is treated as a `NOP`.

*See also* Condition code check.

**CONSTRAINED UNPREDICTABLE**
Where an instruction can result in UNPREDICTABLE behavior, the ARMv8 architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

In body text, the term CONSTRAINED UNPREDICTABLE is shown in SMALL CAPITALS.

*See also* UNPREDICTABLE.

**Context switch** The saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch describes any situation where the context is switched by an operating system and might or might not include changes to the address space.

**Context synchronization operation**
One of:

- Performing an ISB operation. An ISB operation is performed when an `ISB` instruction is executed and does not fail its condition code check.
- Taking an exception.
- Returning from an exception.
- Exit from Debug state.
- Executing a `DCPS` instruction.
- Executing a `DRPS` instruction.

The architecture requires a context synchronization operation to guarantee visibility of any change to a System register.

**Debugger** In most of this manual, *debugger* refers to any agent that is performing debug. However, some parts of the manual require a more rigorous definition, and define debugger locally. See:

- *Definition of a debugger* on page D2-1636 for the definition that applies to Chapter D2.
- *Definition of a debugger* on page G2-3882 for the definition that applies to Chapter G2.
- *Definition of a debugger* on page H1-4816 for the definition that applies to Part H.

**Deprecated** Something that is present in the ARM architecture for backwards compatibility. Whenever possible software must avoid using deprecated features. Features that are deprecated but are not optional are present in current implementations of the ARM architecture, but might not be present, or might be deprecated and OPTIONAL, in future versions of the ARM architecture.

*See also* OPTIONAL.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Digital signal processing (DSP)**
Algorithms for processing signals that have been sampled and converted to digital form. DSP algorithms often use saturated arithmetic.

**Direct Memory Access (DMA)**
An operation that accesses main memory directly, without the PE performing any accesses to the data concerned.

**Direct read** A direct read of a System register is a read performed by a System register access instruction.

For more information, see *Direct read* on page D7-1910.

*See also* Direct write, Indirect read, Indirect write.

**Direct write** A direct write of a System register is a write performed by a System register access instruction.

For more information, see *Direct write* on page D7-1910.

*See also* Direct read, Indirect read, Indirect write.

**DMA** *See* Direct Memory Access (DMA).

**DNM** *See* Do-Not-Modify (DNM).

**Domain** In the ARM architecture, *domain* is used in the following contexts.

**Shareability domain** Defines a set of observers for which the shareability attributes make the data or unified caches transparent for data accesses.

**Power domain** Defines a block of logic with a single, common, power supply.

**Memory regions domain**
When using the Short-descriptor translation table format, defines a collection of Sections, Large pages and Small pages of memory, that can have their access permissions switched rapidly by writing to the *Domain Access Control Register* (DACR). ARM deprecates any use of memory regions domains.

**Do-Not-Modify (DNM)**
Means the value must not be altered by software. DNM fields read as UNKNOWN values, and must only be written with the value read from the same field on the same PE.

**Double-precision value**
Consists of two consecutive 32-bit words that are interpreted as a basic double-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

**Doubleword** A 64-bit data item. Doublewords are normally at least word-aligned in ARM systems.

**Doubleword-aligned**
Means that the address is divisible by 8.

**DSP** *See* Digital signal processing (DSP).

**Endianness** An aspect of the system memory mapping.

*See also* Big-endian memory and Little-endian memory.

**Exception** Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

**Exception vector**
A fixed address that contains the address of the first instruction of the corresponding exception handler.

**Execution stream**
The stream of instructions that would have been executed by sequential execution of the program.

**Explicit access** A read from memory, or a write to memory, generated by a load or store instruction executed by the PE. Reads and writes generated by hardware translation table accesses are not explicit accesses.

**External abort** An abort that is generated by the external memory system.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Fast Context Switch Extension (FCSE)**
Modifies the behavior of an ARM memory system to enable multiple programs running on the ARM PE to use identical address ranges, while ensuring that the addresses they present to the rest of the memory system differ. From ARMv6, ARM deprecates any use of the FCSE. The FCSE is:

- Optional in an ARMv7 implementation that does not include the Multiprocessing Extensions.
- Obsolete from the introduction of the Multiprocessing Extensions.

**FCSE** *See* Fast Context Switch Extension (FCSE).

**Flat address mapping**
Is where the physical address for every access is equal to its virtual address.

**Flush-to-zero mode**
A processing mode that optimizes the performance of some floating-point algorithms by replacing the denormalized operands and intermediate results with zeros, without significantly affecting the accuracy of their final results.

**General-purpose registers**
The registers that the base instructions use for processing:

- In AArch32 state the general-purpose registers are R0-R14, that can also be described as R0-R12, SP, LR.

  **Note**

  Older documentation defines the AArch32 general-purpose registers as R0-R12, and the ARM core registers as R0-R12, SP, LR, and PC.

- In AArch64 state the general-purpose registers are:
  - W0-W30 when accessed as 32-bit registers.
  - X0-X30 when accessed as 64-bit registers.

*See also* High registers, Low registers.

**Halfword** A 16-bit data item. Halfwords are normally halfword-aligned in ARM systems.

**Halfword-aligned**
Means that the address is divisible by 2.

**High registers** In AArch32 state, the general-purpose registers R8-R14. Most 16-bit T32 instructions cannot access the high registers.

**Note**

- In some contexts, *high registers* refers to R8-R15, meaning R8-R14 and the PC.

*See also* General-purpose registers, Low registers.

**High vectors** An alternative location for the exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

**IGNORED** Indicates that the architecture guarantees that the bit or field is not interpreted or modified by hardware.

In body text, the term IGNORED is shown in SMALL CAPITALS.

**Immediate and offset fields**
Are unsigned unless otherwise stated.

**Immediate value**
A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many A64, A32, and T32 instructions can be used with an immediate argument.

**IMP** An abbreviation used in diagrams to indicate that one or more bits have IMPLEMENTATION DEFINED behavior.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**IMPLEMENTATION DEFINED**
Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

In body text, the term IMPLEMENTATION DEFINED is shown in SMALL CAPITALS.

**Index register** A register specified in some load and store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address that is sent to memory. Some instruction forms permit the index register value to be shifted before the addition or subtraction.

**Indirect read** When an instruction uses a System register value to establish operating conditions, that use of the System register is an indirect read of the System register.

For more information, including additional examples of indirect reads, see *Indirect read* on page D7-1910.

*See also* Direct read, Direct write, Indirect write.

**Indirect write** An indirect write of a System register occurs when the contents of a register are updated by some mechanism other than a *Direct write* to that register. For example, an indirect write to a register might occur as a side-effect of executing an instruction that does not perform a direct write to the register, or because of some operation performed by an external agent.

For more information, see *Indirect write* on page D7-1910.

*See also* Direct read, Direct write, Indirect read.

**Inline literals** These are constant addresses and other data items held in the same area as the software itself. They are automatically generated by compilers, and can also appear in assembler code.

**Intermediate physical address (IPA)**
An implementation of virtualization, the address to which a Guest OS maps a VA. A hypervisor might then map the IPA to a PA. Typically, the Guest OS is unaware of the translation from IPA to PA.

*See also* Physical address (PA), Virtual address (VA).

**Interworking** A method of working that permits branches between software using the A32 and T32 instruction sets.

**IPA** *See* Intermediate physical address (IPA).

**Level** *See* Cache level.

**Level of Coherence (LoC)**
The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of coherency. For more information, see *Terms used in describing the maintenance instructions* on page D3-1712.

*See also* Cache level, Point of coherency (PoC).

**Level of Unification, Inner Shareable (LoUIS)**
The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for the Inner Shareable shareability domain. For more information, see *Terms used in describing the maintenance instructions* on page D3-1712.

*See also* Cache level, Point of unification (PoU).

**Level of Unification, uniprocessor (LoUU)**
For a PE, the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for that PE. For more information, see *Terms used in describing the maintenance instructions* on page D3-1712.

*See also* Cache level, Point of unification (PoU).

**Line** *See* Cache line.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Little-endian memory**
Means that, for example:

- A byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the least significant byte in the halfword at that address.

*See also* Big-endian memory, .Endianness.

**Load/Store architecture**
An architecture where data-processing operations only operate on register contents, not directly on memory contents.

**LoC** *See* Level of Coherence (LoC).

**LoUIS** *See* Level of Unification, Inner Shareable (LoUIS).

**LoUU** *See* Level of Unification, uniprocessor (LoUU).

**Lockdown** *See* Cache lockdown.

**Low registers** InAArch32 state, general-purpose registers R0-R7. Unlike the high registers, all T32 instructions can access the Low registers.

*See also* General-purpose registers, High registers.

**Memory barrier** *See Memory barriers* on page B2-86.

**Memory coherency**
The problem of ensuring that when a memory location is read, either by a data read or an instruction fetch, the value actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache.

**Memory Management Unit (MMU)**
Provides detailed control of the part of a memory system that provides a single stage of address translation. Most of the control is provided using translation tables that are held in memory, and define the attributes of different regions of the physical memory map.

**Memory Protection Unit (MPU)**
A hardware unit whose registers provide simple control of a limited number of protection regions in memory.

**Miss** *See* Cache miss.

**MMU** *See* Memory Management Unit (MMU).

**MPU** *See* Memory Protection Unit (MPU).

**Multi-copy atomicity**
The form of atomicity described in *Requirements for multi-copy atomicity* on page B2-81.

*See also* Atomicity, Single-copy atomicity.

**NaN**

- Not a Number. A floating-point value that can be used when neither a numeric value nor an infinity is appropriate. A NaN can be a *quiet* NaN, that propagate through most floating-point operations, or a *signaling* NaN, that causes an Invalid Operation floating-point exception when used. For more information, see the *IEEE Standard for Floating-point Arithmetic*.

**Natural eviction** A natural eviction is an eviction that occurs in the course of the normal operation of the memory system, rather than because of an operation that explicitly causes an eviction from the cache, such as the execution of a cache maintenance instruction. Typically, a natural eviction occurs when the caching algorithm requires data to be cached but the cache does not have room for that data.

**Observer** A PE or mechanism in the system, such as a peripheral device, that can generate reads from or writes to memory.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Obsolete** Obsolete indicates something that is no longer supported by ARM. When an architectural feature is described as obsolete, this indicates that the architecture has no support for that feature, although an earlier version of the architecture did support it.

**Offset addressing**
Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

**OPTIONAL** When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the ARM architecture:

- If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture. ARM expects such a features to be included in a new implementation only if there is a known backwards-compatibility reason for the inclusion of the feature.

  A feature that is OPTIONAL and deprecated might not be present in future versions of the architecture.

- A feature that is OPTIONAL but not deprecated is, typically, a feature added to a version of the ARM architecture after the initial release of that version of the architecture. ARM recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term OPTIONAL are shown in SMALL CAPITALS.

**Note**: Do not confuse these ARM-specific uses of OPTIONAL with other uses of *optional*, where it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
- Behavior determined by an implementation choice, for example the optional byte order reversal in an ARMv7-R implementation, where the SCTLR.IE bit indicates the implemented option.

*See also* Deprecated.

**PA** *See* Physical address (PA).

**PE** *See* Processing element (PE).

**Physical address (PA)**
An address that identifies a location in the physical memory map.

*See also* Intermediate physical address (IPA), Virtual address (VA).

**PoC** *See* Point of coherency (PoC).

**PoU** *See* Point of unification (PoU).

**Point of coherency (PoC)**
For a particular MVA, the point at which all agents that can access memory are guaranteed to see the same copy of a memory location. For more information, see *Terms used in describing the maintenance instructions* on page D3-1712.

**Point of unification (PoU)**
For a particular PE, the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. For more information, see *Terms used in describing the maintenance instructions* on page D3-1712.

**Post-indexed addressing**
Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

**Prefetching** Prefetching refers to speculatively fetching instructions or data from the memory system. In particular, instruction prefetching is the process of fetching instructions from memory before the instructions that precede them, in simple sequential execution of the program, have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

In this manual, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Note**

The Prefetch Abort exception can be generated on any instruction fetch, and is not limited to speculative instruction fetches.

*See also* Simple sequential execution.

**Pre-indexed addressing**
Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

**Processing element (PE)**
The abstract machine defined in the ARM architecture, as documented in an ARM Architecture Reference Manual. A PE implementation compliant with the ARM architecture must conform with the behaviors described in the corresponding ARM Architecture Reference Manual.

**Protection region**
A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

**Protection Unit** *See* Memory Protection Unit (MPU).

**Pseudo-instruction**
UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV <Rd>, <Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL <Rd>, <Rm>, #<n>`.

**PSTATE** An abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

*See also* Condition flags.

**Quadword** A 128-bit data item. Quadwords are normally at least word-aligned in ARM systems.

**Quadword-aligned**
Means that the address is divisible by 16.

**Quiet NaN** A NaN that propagates unchanged through most floating-point operations.

**RAO** *See* Read-As-One (RAO).

**RAZ** *See* Read-As-Zero (RAZ).

**RAO/SBOP** In versions of the ARM architecture before ARMv8, Read-As-One, Should-Be-One-or-Preserved on writes.

In ARMv8, RES1 replaces this description.

*See also* UNK/SBOP, Read-As-One (RAO), RES1, Should-Be-One-or-Preserved (SBOP).

**RAO/WI** Read-As-One, Writes Ignored.

Hardware must implement the field as Read-as-One, and must ignore writes to the field.

Software can rely on the field reading as all 1s, and on writes being ignored.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

*See also* Read-As-One (RAO).

**RAZ/SBZP** In versions of the ARM architecture before ARMv8, Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

In ARMv8, RES0 replaces this description.

*See also* UNK/SBZP, Read-As-Zero (RAZ), RES0, Should-Be-Zero-or-Preserved (SBZP).

**RAZ/WI** Read-As-Zero, Writes Ignored.

Hardware must implement the field as Read-as-Zero, and must ignore writes to the field.

Software can rely on the field reading as all 0s, and on writes being ignored.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

*See also* Read-As-Zero (RAZ).

**Read-allocate cache**

A cache in which a cache miss on reading data causes a cache line to be allocated into the cache.

**Read-As-One (RAO)**

Hardware must implement the field as reading as all 1s.

Software:

- Can rely on the field reading as all 1s.
- Must use a SBOP policy to write to the field.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

**Read-As-Zero (RAZ)**

Hardware must implement the field as reading as all 0s.

Software:

- Can rely on the field reading as all 0s
- Must use a SBZP policy to write to the field.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

**Read, modify, write**

In a read, modify, write instruction sequence, a value is read to a general-purpose register, the relevant fields updated in that register, and the new value written back.

**RES0**

A reserved bit or field with *Should-Be-Zero-or-Preserved (SBZP)* behavior. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or bitfield:

- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

**Note**

- RES0 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is *Architecturally mapped* to an AArch64 System register, and a bit or field in that register is RES0 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES0 for register fields is:

**If a bit is RES0 in all contexts**

It is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 0. In this case:
   - Reads of the bit always return 0.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 0.
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

- A direct write to the bit must update a storage location associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

**If a bit is RES0 only in some contexts**

When the bit is described as RES0:

- An indirect write to the register sets the bit to 0.
- A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

A bit that is RES0 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 0.
- Must use an SBZP policy to write to the bit.

The RES0 description can apply to bits or bitfields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as 0, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as SBZ.

This RES0 description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

In body text, the term RES0 is shown in SMALL CAPITALS.

*See also* Read-As-Zero (RAZ), RES1, Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

**RES1**

A reserved bit or field with *Should-Be-One-or-Preserved (SBOP)* behavior. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or bitfield:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

**Note**

- RES1 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is *Architecturally mapped* to an AArch64 System register, and a bit or field in that register is RES1 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES1 for register fields is:

**If a bit is RES1 in all contexts**

It is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
   - Reads of the bit always return 1.
   - Writes to the bit are ignored.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 1.
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location associated with the bit.
   - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

   Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

**If a bit is RES1 only in some contexts**

When the bit is described as RES1:
- An indirect write to the register sets the bit to 1.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  **Note**

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:
- Must not rely on the bit reading as 1.
- Must use an SBOP policy to write to the bit.

The RES1 description can apply to bits or bitfields that are read-only, or are write-only:
- For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as SBO.

This RES1 description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

In body text, the term RES1 is shown in SMALL CAPITALS.

*See also* Read-As-One (RAO), RES0, Should-Be-One-or-Preserved (SBOP), UNKNOWN.

**Reserved**

Unless otherwise stated:
- Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- Bit positions described as reserved are:
  - In an RW or WO register, RES0.
  - In an RO register, UNK.

*See also* RES0, RES1, UNK.

**RISC**

Reduced Instruction Set Computer.

**Rounding error**

The value of the rounded result of an arithmetic operation minus the exact result of the operation.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Rounding mode** Specifies how the exact result of a floating-point operation is rounded to a value that is representable in the destination format. The rounding modes are defined by the *IEEE Standard for Floating-point Arithmetic*, see *Floating-point standards, and terminology* on page A1-48.

**Saturated arithmetic**
Integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in ARM processors, in which overflowing results wrap around from $+2^{31}–1$ to $–2^{31}$ or vice versa.

**SBO** *See* Should-Be-One (SBO).

**SBOP** *See* Should-Be-One-or-Preserved (SBOP).

**SBZ** *See* Should-Be-Zero (SBZ).

**SBZP** *See* Should-Be-Zero-or-Preserved (SBZP).

**Security hole** A mechanism by which execution at the current level of privilege can achieve an outcome that cannot be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE. The ARM architecture forbids security holes.

**Self-modifying code**
Code that writes one or more instructions to memory and then executes them. When using self-modifying code, you must use cache maintenance and barrier instructions to ensure synchronization. For more information, see *Caches and memory hierarchy* on page B2-70.

**Set** *See* Cache sets.

**Should-Be-One (SBO)**
Hardware must ignore writes to the field.

ARM strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.

**Should-Be-One-or-Preserved (SBOP)**
From the introduction of the ARMv8 architecture, the description *Should-Be-One-or-Preserved (SBOP)* is superseded by *RES1*.

**Note**

The ARMv7 Large Physical Address Extension modified the definition of SBOP for register bits that are SBOP in some but not all contexts. The behavior of these bits is covered by the *RES1* definition, but not by the generic definition of SBOP given here.

Hardware must ignore writes to the field.

If software has read the field since the PE implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 1s.

If software writes a value to the field that is not a value previously read for the field and is not all 1s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Should-Be-Zero (SBZ)**
Hardware must ignore writes to the field.

ARM strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.

**Should-Be-Zero-or-Preserved (SBZP)**
From the introduction of the ARMv8 architecture, the description *Should-Be-Zero-or-Preserved (SBZP)* is superseded by *RES0*.

**Note**

The ARMv7 Large Physical Address Extension modified the definition of SBZP for register bits that are SBZP in some but not all contexts. The behavior of these bits is covered by the *RES0* definition, but not by the generic definition of SBZP given here.

Hardware must ignore writes to the field.

If software has read the field since the PE implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 0s.

If software writes a value to the field that is not a value previously read for the field and is not all 0s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

**Signaling NaNs** Cause an Invalid Operation exception whenever any floating-point operation receives a signaling NaN as an operand. Signaling NaNs can be used in debugging, to track down some uses of uninitialized variables.

**Signed immediate and offset fields**
Are encoded in two's complement notation unless otherwise stated.

**SIMD** Single-Instruction, Multiple-Data.

The SIMD instructions in AArch32 state are:

- The instructions summarized in *Parallel addition and subtraction instructions* on page F1-2418.
- The Advanced SIMD instructions summarized in *Advanced SIMD and floating-point instructions* on page E1-2325, when operating on vectors.

  **Note**

  In ARMv7, some VFP instructions can operate on vectors. However, ARM deprecates those instruction uses, and strongly recommends that Advanced SIMD instructions are always used for vector operations.

**Simple sequential execution**
The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and ARM does not expect this model to correspond to a realistic implementation of the architecture.

**Single-copy atomicity**
The form of atomicity described in *Properties of single-copy atomic accesses* on page B2-81.

*See also* Atomicity, Multi-copy atomicity.

**Single-precision value**
A 32-bit word that is interpreted as a basic single-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.


Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.
Non-Confidential - Beta

**Spatial locality** The observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

**Special-purpose register**
One of a specified set of registers for which all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization:

- *Special-purpose registers* on page C5-289 specifies the AArch64 Special-purpose registers.
- *AArch32 Special-purpose registers* on page G1-3764 lists the AArch32 Special-purpose registers.

**T32 instruction** One or two halfwords that specify an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in T32 state. T32 instructions must be halfword-aligned.

T32 instructions were previously called Thumb instructions.

*See also* A32 instruction, A64 instruction, T32 state.

**T32 state** The AArch32 Instruction set state in which the PE executes T32 instructions.

T32 state was previously called Thumb state.

*See also* A32 state, T32 instruction.

**Temporal locality**
The observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

**Thumb instruction**
*See* T32 instruction.

**TLB** *See* Translation Lookaside Buffer (TLB).

**TLB lockdown** A way to prevent specific translation table walk results being accessed. This ensures that accesses to the associated memory areas never cause a translation table walk.

**Translation Lookaside Buffer (TLB)**
A memory structure containing the results of translation table walks. They help to reduce the average cost of a memory access. Usually, there is a TLB for each memory interface of the ARM implementation.

**Translation table**
A table held in memory that defines the properties of memory areas of various sizes from 1KB to 1MB.

**Translation table walk**
The process of doing a full translation table lookup. It is performed automatically by hardware.

**Trap enable bits** In VFPv2, VFPv3U, and VFPv4U, determine whether trapped or untrapped exception handling is selected. If trapped exception handling is selected, the way it is carried out is IMPLEMENTATION DEFINED.

**Unaligned** An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

**Unaligned memory accesses**
Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

**Unallocated** Except where otherwise stated in this manual, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE, or an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

**UNDEFINED** Indicates an instruction that generates an Undefined Instruction exception.

In body text, the term UNDEFINED is shown in SMALL CAPITALS.

*See also Undefined Instruction exception* on page G1-3811.

**Unified cache** Is a cache used for both processing instruction fetches and processing data loads and stores.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Unindexed addressing**
Means addressing in which the base register value is used directly as the virtual address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0. The `LDC`, `LDC2`, `STC`, and `STC2` instructions have an explicit unindexed addressing mode that permits the offset field in the instruction to specify additional coprocessor options.

**UNK**
An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Hardware must implement the bit as read as 0, or all 0s for a multi-bit field. Software must not rely on the field reading as zero.

*See also* UNKNOWN.

**UNK/SBOP**
Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

*See also* Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

**UNK/SBZP**
Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.

Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

*See also* Read-As-Zero (RAZ), Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

**UNKNOWN**
An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and do not return UNKNOWN values.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALL CAPITALS.

*See also* UNK.

**UNPREDICTABLE**
Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

Execution in a Non-secure EL1 or EL0 mode of an instruction that is UNPREDICTABLE can be implemented as generating a Hyp Trap exception, provided that at least one instruction that is not UNPREDICTABLE causes a Hyp Trap exception.

In body text, the term UNPREDICTABLE is shown in SMALL CAPITALS.

*See also* CONSTRAINED UNPREDICTABLE.

**VA**
*See* Virtual address (VA).

**VFP**
In ARMv7, an extension to the ARM architecture, that provides single-precision and double-precision floating-point arithmetic.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*

**Virtual address (VA)**
An address generated by an ARM PE. This means it is an address that might be held in the program counter of the PE. For a PMSA implementation, the virtual address is identical to the physical address.

*See also* Intermediate physical address (IPA), Physical address (PA).

**Watchpoint** A debug event triggered by an access to memory, specified in terms of the address of the location in memory being accessed.

**Way** *See* Cache way.

**WI** Writes Ignored. In a register that software can write to, a WI attribute applied to a bit or field indicates that the bit or field ignores the value written by software and retains the value it had before that write.

*See also* RAO/WI, RAZ/WI, RES0, RES1.

**Word** A 32-bit data item. Words are normally word-aligned in ARM systems.

**Word-aligned** Means that the address is divisible by 4.

**Write-allocate cache**
A cache in which a cache miss on storing data causes a cache line to be allocated into the cache.

**Write-back cache**
A cache in which when a cache hit occurs on a store access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in main memory. Any such data is written back to main memory when the cache line is cleaned or reallocated. Another common term for a write-back cache is a *copy-back cache*.

**Write-through cache**
A cache in which when a cache hit occurs on a store access, the data is written both to the cache and to main memory. This is normally done via a write buffer, to avoid slowing down the PE.

**Write buffer** A block of high-speed memory that optimizes stores to main memory.


*Copyright © 2013-2015 ARM Limited or its affiliates. All rights reserved.*
*Non-Confidential - Beta*